Chapter 4
Petri Nets

4.1 INTRODUCTION

An alternative to automata for untimed models of DES is provided by Petri nets. These models were first developed by C. A. Petri in the early 1960s. As we will see, Petri nets are related to automata in the sense that they also explicitly represent the transition function of DES. Like an automaton, a Petri net is a device that manipulates events according to certain rules. One of its features is that it includes explicit conditions under which an event can be enabled; this allows the representation of very general DES whose operation depends on potentially complex control schemes. This representation is conveniently described graphically, at least for small systems, resulting in *Petri net graphs*; Petri net graphs are intuitive and capture a lot of structural information about the system. We will see that an automaton can always be represented as a Petri net; on the other hand, not all Petri nets can be represented as *finite-state* automata. Consequently, Petri nets can represent a larger class of languages than the class of regular languages, $\mathcal{R}$. Another motivation for considering Petri net models of DES is the body of analysis techniques that have been developed for studying them. These techniques include reachability analysis, similarly to the case of automata, as well as linear-algebraic techniques. Such techniques cover not only untimed Petri net models but timed Petri net models as well; in this regard, we will see in the next chapter that there is a well-developed theory, called the "max-plus algebra," for a certain class of timed Petri nets (cf. Sect. 5.4). Finally, we mention that control of Petri nets is an active research area and there are controller synthesis techniques that exploit the structural properties of Petri nets.

4.2 PETRI NET BASICS

The process of defining a Petri net involves two steps. First, we define the *Petri net graph*, also called *Petri net structure*, which is analogous to the state transition diagram of an automaton; this is the focus of Sect. 4.2.1. Then we adjoin to this graph an initial state, a set of marked states, and a transition labeling function, resulting in the complete Petri net model, its associated dynamics, and the languages that it generates and marks; this is treated in Sects. 4.2.2–4.2.4.

4.2.1 Petri Net Notation and Definitions

In Petri nets, events are associated with *transitions*. In order for a transition to occur, several conditions may have to be satisfied. Information related to these conditions is contained in *places*. Some such places are viewed as the "input" to a transition; they are associated with the conditions required for this transition to occur. Other places are viewed as the output of a transition; they are associated with conditions that are affected by the occurrence of this transition. Transitions, places, and certain relationships between them define the basic components of a *Petri net graph*. A Petri net graph has two types of nodes, places and transitions, and arcs connecting these. It is a *bipartite graph* in the sense that arcs cannot directly connect nodes of the same type; rather, arcs connect place nodes to transition nodes and transition nodes to place nodes. The precise definition of Petri net graph is as follows.

Definition. (Petri net graph)
A *Petri net graph* (or *Petri net structure*) is a weighted bipartite graph

$$(P, T, A, w)$$

where

P is the finite set of *places* (one type of node in the graph)

T is the finite set of *transitions* (the other type of node in the graph)

$A \subseteq (P \times T) \cup (T \times P)$ is the set of arcs from places to transitions and from transitions to places in the graph

$w : A \to \{1, 2, 3, \ldots\}$ is the *weight function* on the arcs.

We assume that (P, T, A, w) has no isolated places or transitions. ♦

We will normally represent the set of places by $P = \{p_1, p_2, \ldots, p_n\}$, and the set of transitions by $T = \{t_1, t_2, \ldots, t_m\}$; thus, in the remainder of this chapter, $|P| = n$ and $|T| = m$. A typical arc is of the form (p_i, t_j) or (t_j, p_i), and the weight related to an arc is a positive integer. Note that we could allow P and T to be countable, rather than finite sets, as is the case for the state set in automata. It turns out, however, that a finite number of transitions and places is almost always perfectly adequate in modeling DES of interest.

We can see that a Petri net graph is somewhat more elaborate than the state transition diagram of an automaton. First, the nodes of a state transition diagram correspond to states selected from a single set X. In a Petri net graph, the nodes are either places, selected from the set P, or transitions, selected from the set T. Second, in a state transition diagram there is a single arc for each event causing a state transition. In a Petri net graph, we

allow multiple arcs to connect two nodes, or, equivalently, we assign a weight to each arc representing the number of arcs. This is why we call this structure a *multigraph*.

In describing a Petri net graph, it is convenient to use $I(t_j)$ to represent the set of input places to transition t_j. Similarly, $O(t_j)$ represents the set of output places from transition t_j. Thus, we have

$$I(t_j) = \{p_i \in P : (p_i, t_j) \in A\}, \qquad O(t_j) = \{p_i \in P : (t_j, p_i) \in A\}$$

Similar notation can be used to describe input and output transitions for a given place p_i: $I(p_i)$ and $O(p_i)$.

When drawing Petri net graphs, we need to differentiate between the two types of nodes, places and transitions. The convention is to use circles to represent places and bars to represent transitions. The arcs connecting places and transitions represent elements of the arc set A. Thus, an arc directed from place p_i to transition t_j means that $p_i \in I(t_j)$. Moreover, if $w(p_i, t_j) = k$, then there are k arcs from p_i to t_j or, equivalently, a single arc accompanied by its weight k. Similarly, if there are k arcs directed from transition t_j to place p_i, this means that $p_i \in O(t_j)$ and $w(t_j, p_i) = k$. We will generally represent weights through multiple arcs on a graph. However, when large weights are involved in a Petri net, writing the weight on the arc is a much more efficient representation. If no weight is shown on an arc of a Petri net graph, we will assume it to be 1. Finally, we mention that it is convenient to extend the domain and co-domain of the weight function w and write

$$w(p_i, t_j) = 0 \ \text{ when } \ p_i \notin I(t_j) \qquad \text{and} \qquad w(t_j, p_i) = 0 \ \text{ when } \ p_i \notin O(t_j)$$

Example 4.1 (Simple Petri net graph)
Consider the Petri net graph defined by

$$P = \{p_1, p_2\} \qquad T = \{t_1\} \qquad A = \{(p_1, t_1), (t_1, p_2)\}$$

$$w(p_1, t_1) = 2 \qquad w(t_1, p_2) = 1$$

In this case, $I(t_1) = \{p_1\}$ and $O(t_1) = \{p_2\}$. The corresponding Petri net graph is shown in Fig. 4.1. The fact that $w(p_1, t_1) = 2$ is indicated by the presence of two input arcs from place p_1 to transition t_1.

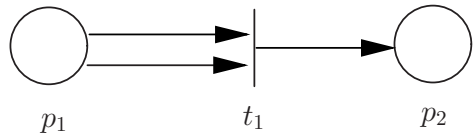

Figure 4.1: Petri net graph for Example 4.1.

Example 4.2
Consider the Petri net graph shown in Fig. 4.2. The Petri net it represents is specified by

$$P = \{p_1, p_2, p_3, p_4\} \qquad T = \{t_1, t_2, t_3, t_4, t_5\}$$

$$A = \{(p_1, t_1), (p_1, t_2), (p_2, t_2), (p_2, t_3), (p_2, t_5), (p_4, t_5), (t_1, p_1), (t_1, p_2), (t_2, p_3), (t_3, p_3), (t_3, p_4), (t_4, p_3), (t_5, p_1)\}$$

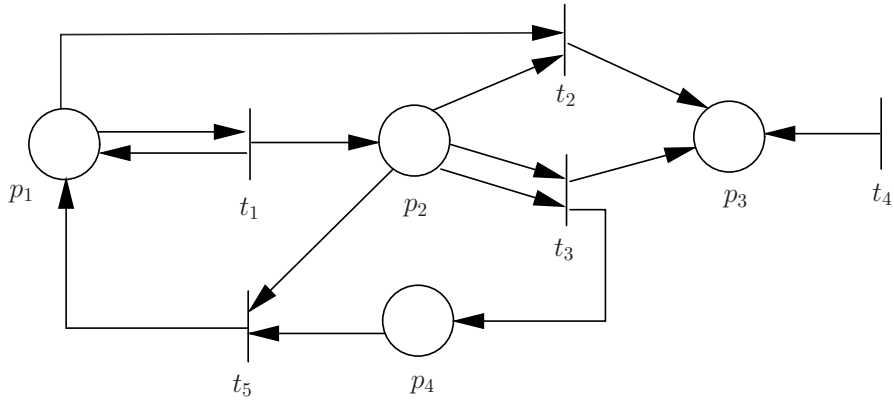

Figure 4.2: Petri net graph for Example 4.2.

$$
\begin{array}{llll}
w(p_1,t_1)=1 & w(p_1,t_2)=1 & w(p_2,t_2)=1 & w(p_2,t_3)=2 \\
w(p_2,t_5)=1 & w(p_4,t_5)=1 & w(t_1,p_1)=1 & w(t_1,p_2)=1 \\
w(t_2,p_3)=1 & w(t_3,p_3)=1 & w(t_3,p_4)=1 & w(t_4,p_3)=1 \\
w(t_5,p_1)=1 & & &
\end{array}
$$

It is worth commenting on the fact that transition t_4 has no input places. If we think of transitions as events and places as conditions related to event occurrences, then the event corresponding to t_4 takes place unconditionally. In contrast, the event corresponding to transition t_2, for example, depends on certain conditions related to places p_1 and p_2.

4.2.2 Petri Net Markings and State Spaces

Let us return to the idea that transitions in a Petri net graph represent the events driving a DES, and that places describe the conditions under which these events can occur. In this framework, we need a mechanism indicating whether these conditions are in fact met or not. This mechanism is provided by assigning *tokens* to places. A token is something we "put in a place" essentially to indicate the fact that the condition described by that place is satisfied. The way in which tokens are assigned to a Petri net graph defines a *marking*. Formally, a *marking* x of a Petri net graph (P, T, A, w) is a function $x : P \to \mathbb{N} = \{0, 1, 2, \ldots\}$. Thus, marking x defines row vector $\mathbf{x} = [x(p_1), x(p_2), \ldots, x(p_n)]$, where n is the number of places in the Petri net. The ith entry of this vector indicates the (non-negative integer) number of tokens in place p_i, $x(p_i) \in \mathbb{N}$. In Petri net graphs, a token is indicated by a dark dot positioned in the appropriate place.

Definition. (Marked Petri net)
A *marked Petri net* is a five-tuple (P, T, A, w, x) where (P, T, A, w) is a Petri net graph and x is a marking of the set of places P; $\mathbf{x} = [x(p_1), x(p_2), \ldots, x(p_n)] \in \mathbb{N}^n$ is the row vector associated with x. ♦

Example 4.3
Consider the Petri net of Fig. 4.1. In Fig. 4.3, we show two possible markings, namely the row vectors $\mathbf{x}_1 = [1, 0]$ and $\mathbf{x}_2 = [2, 1]$.

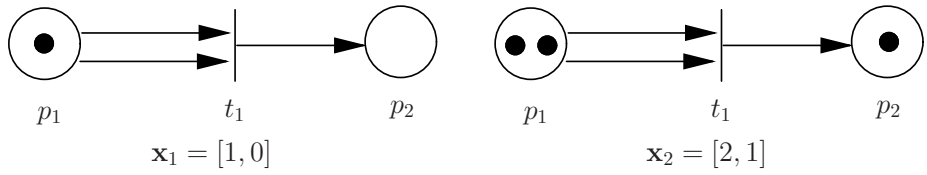

Figure 4.3: Two markings, $\mathbf{x}_1$ and $\mathbf{x}_2$, for the Petri net graph in Fig. 4.1 (Example 4.3).

For simplicity, we shall henceforth refer to a marked Petri net as just a Petri net. Now, since our system modeling efforts have always relied on the concept of state, in the case of a Petri net we identify place marking with the *state* of the Petri net. That is, we define the *state* of a Petri net to be its marking row vector $\mathbf{x} = [x(p_1), x(p_2), \ldots, x(p_n)]$. Note that the number of tokens assigned to a place is an arbitrary non-negative integer, not necessarily bounded. It follows that the number of states we can have is, in general, infinite. Thus, the state space X of a Petri net with n places is defined by all n-dimensional vectors whose entries are non-negative integers, that is, $X = \mathbb{N}^n$. While the term "marking" is more common than "state" in the Petri net literature, the term state is consistent with the role of state in system dynamics, as we shall now see; moreover, the term state avoids the potential confusion between marking in Petri net graphs and marking in the sense of *marked states* in automata (and Petri nets too, cf. Sect. 4.2.4).

The above definitions do not explicitly describe the state transition mechanism of Petri nets. This is clearly a crucial point since we want to use Petri nets to model *dynamic* DES. It turns out that the state transition mechanism is captured by the structure of the Petri net graph. In order to define the state transition mechanism, we first need to introduce the notion of *enabled transition*. Basically, for a transition $t \in T$ to "happen" or to "be enabled," we require a token to be present in each place (i.e., condition) which is input to the transition. Since, however, we allow weighted arcs from places to transitions, we use a slightly more general definition.

Definition. (Enabled transition)
A transition $t_j \in T$ in a Petri net is said to be *enabled* if

$$x(p_i) \geq w(p_i, t_j) \quad \text{for all } p_i \in I(t_j) \qquad \blacklozenge$$

In words, transition t_j in the Petri net is enabled when the number of tokens in p_i is at least as large as the weight of the arc connecting p_i to t_j, for all places p_i that are input to transition t_j. In Fig. 4.3 with state $\mathbf{x}_1$, $x(p_1) = 1 < w(p_1, t_1) = 2$, and therefore t_1 is not enabled. But with state $\mathbf{x}_2$, we have $x(p_1) = 2 = w(p_1, t_1)$, and t_1 is enabled.

As was mentioned above, since places are associated with conditions regarding the occurrence of a transition, then a transition is enabled when all the conditions required for its occurrence are satisfied; tokens are the mechanism used to determine satisfaction of conditions. The set of enabled transitions in a given state of the Petri net is equivalent to the active event set in a given state of an automaton. We are now ready to describe the dynamical evolution of Petri nets.

4.2.3 Petri Net Dynamics

In automata, the state transition mechanism is directly captured by the arcs connecting the nodes (states) in the state transition diagram, equivalently by the transition function f.

The state transition mechanism in Petri nets is provided by *moving tokens* through the net and hence changing the state of the Petri net. When a transition is enabled, we say that it can *fire* or that it can occur (the term "firing" is standard in the Petri net literature). The state transition function of a Petri net is defined through the change in the state of the Petri net due to the firing of an enabled transition.

Definition. (Petri net dynamics)
The *state transition function*, $f : \mathbb{N}^n \times T \to \mathbb{N}^n$, of Petri net (P, T, A, w, x) is defined for transition $t_j \in T$ if and only if

$$x(p_i) \geq w(p_i, t_j) \quad \text{for all } p_i \in I(t_j) \tag{4.1}$$

If $f(\mathbf{x}, t_j)$ is defined, then we set $\mathbf{x}' = f(\mathbf{x}, t_j)$, where

$$x'(p_i) = x(p_i) - w(p_i, t_j) + w(t_j, p_i), \quad i = 1, \ldots, n \tag{4.2}$$

♦

Condition (4.1) ensures that the state transition function is defined only for transitions that are enabled; an "enabled transition" is therefore equivalent to a "feasible event" in an automaton, as we mentioned above. But whereas in automata the state transition function was quite arbitrary, here the state transition function is based on the structure of the Petri net. Thus, the next state defined by (4.2) explicitly depends on the input and output places of a transition and on the weights of the arcs connecting these places to the transition.

According to (4.2), if p_i is an input place of t_j, it loses as many tokens as the weight of the arc from p_i to t_j; if it is an output place of t_j, it gains as many tokens as the weight of the arc from t_j to p_i. Clearly, it is possible that p_i is both an input and output place of t_j, in which case (4.2) removes $w(p_i, t_j)$ tokens from p_i, and then immediately places $w(t_j, p_i)$ new tokens back in it.

It is important to remark that the number of tokens need not be conserved upon the firing of a transition in a Petri net. This is immediately clear from (4.2), since it is entirely possible that

$$\sum_{p_i \in P} w(t_j, p_i) > \sum_{p_i \in P} w(p_i, t_j) \qquad \text{or} \qquad \sum_{p_i \in P} w(t_j, p_i) < \sum_{p_i \in P} w(p_i, t_j)$$

in which case either $\mathbf{x}' = f(\mathbf{x}, t_j)$ contains more tokens than $\mathbf{x}$ or less tokens than $\mathbf{x}$. In general, it is entirely possible that after several transition firings, the resulting state is $\mathbf{x} = [0, \ldots, 0]$, or that the number of tokens in one or more places grows arbitrarily large after an arbitrarily large number of transition firings. The latter phenomenon is a key difference with automata, where finite-state automata have only a finite number of states, by definition. In contrast, a finite Petri net graph may result in a Petri net with an unbounded number of states. We will comment further on this issue later in this section.

Example 4.4 (Firing of transitions)
To illustrate the process of firing transitions and changing the state of a Petri net, consider the Petri net of Fig. 4.4 (a), where the "initial" state is $\mathbf{x}_0 = [2, 0, 0, 1]$. We can see that the only transition enabled is t_1, since it requires a single token from place p_1 and we have $x_0(p_1) = 2$. In other words, $x_0(p_1) \geq w(p_1, t_1)$, and condition (4.1) is

satisfied for transition t_1. When t_1 fires, one token is removed from p_1, and one token is placed in each of places p_2 and p_3, as can be seen from the Petri net graph. We can also directly apply (4.2) to obtain the new state $\mathbf{x}_1 = [1, 1, 1, 1]$, as shown in Fig. 4.4 (b). In this state, all three transitions t_1, t_2, and t_3 are enabled.

Next, suppose transition t_2 fires. One token is removed from each of the input places, p_2 and p_3. The output places are p_2 and p_4. Therefore, a token is immediately placed back in p_2, since $p_2 \in I(t_2) \cap O(t_2)$. In addition, a token is added to p_4. The new state is $\mathbf{x}_2 = [1, 1, 0, 2]$, as shown in Fig. 4.4 (c). At this state, t_2 and t_3 are no longer enabled, but t_1 still is.

Let us now return to state $\mathbf{x}_1$ of Fig. 4.4 (b), where all three transitions are enabled. Instead of firing t_2, let us fire t_3. We remove a token from each of the input places, p_1, p_3, and p_4. Note that there are no output places. The new state is denoted by $\mathbf{x}_2'$ and is given by $\mathbf{x}_2' = [0, 1, 0, 0]$, as shown in Fig. 4.4 (d). We see that no transition is now enabled. No further state changes are possible, and $[0, 1, 0, 0]$ is a "deadlock" state of this Petri net.

The number of tokens is not conserved in this Petri net since $\mathbf{x}_0$ contains three tokens while $\mathbf{x}_1$ and $\mathbf{x}_2$ each contain four tokens and $\mathbf{x}_2'$ contains one token.

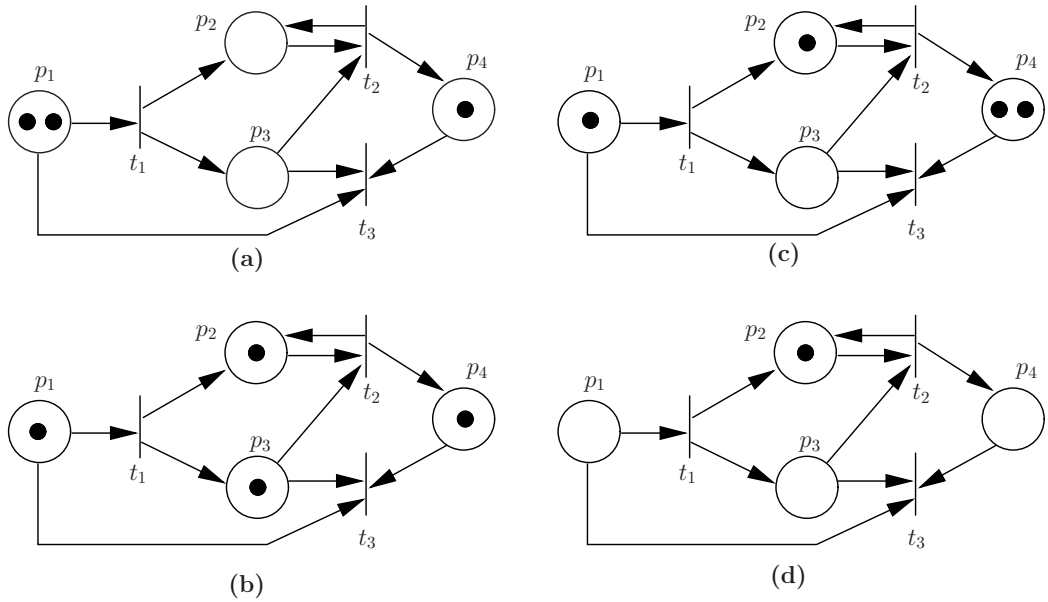

Figure 4.4: Sequences of transition firings in a Petri net (Example 4.4).
In (**a**)*, only transition* t_1 *is enabled in initial state* $\mathbf{x}_0$*. The new state resulting from the firing of* t_1 *in* (**a**) *is denoted by* $\mathbf{x}_1$ *and is shown in* (**b**)*. The new state resulting from the firing of* t_2 *in* (**b**) *is denoted by* $\mathbf{x}_2$ *and is shown in* (**c**)*. The new state resulting from the firing of* t_3 *in state* $\mathbf{x}_1$ *in* **(b)** *is denoted by* $\mathbf{x}_2'$ *and is shown in* (**d**)*.*

The preceding example shows that the sequence in which transitions fire is not prespecified in a Petri net. At state $\mathbf{x}_1$, any one of the three transitions could fire next. This is the same as having three events in the active event set of a state in an automaton model

of a DES. In the study of untimed Petri net models of DES, we must examine each possible sequence of transitions (events), as we did for automata in Chaps. 2 and 3.

Another important observation about the dynamic behavior of Petri nets is that not all states in $\mathbb{N}^n$ can necessarily be reached from a Petri net graph with a given initial state. For instance, if we examine the Petri net graph in Fig. 4.3 together with the initial state $\mathbf{x}_2 = [2, 1]$, then we see that the only state that can be reached from $\mathbf{x}_2$ is $[0, 2]$. This leads us to defining the set of reachable states, $R[(P, T, A, w, x)]$, of Petri net (P, T, A, w, x). In this regard, we first need to extend the state transition function f from domain $\mathbb{N}^n \times T$ to domain $\mathbb{N}^n \times T^*$, in the same manner as we extended the transition function of automata in Sect. 2.2.2:

$$\begin{aligned} f(\mathbf{x}, \varepsilon) &:= \mathbf{x} \\ f(\mathbf{x}, st) &:= f(f(\mathbf{x}, s), t) \quad \text{for } s \in T^* \text{ and } t \in T \end{aligned}$$

(Here the symbol ε is to be interpreted as the absence of transition firing.)

Definition. (Reachable states)
The set of *reachable states* of Petri net (P, T, A, w, x) is

$$R[(P, T, A, w, x)] := \{\mathbf{y} \in \mathbb{N}^n : \exists s \in T^*(f(\mathbf{x}, s) = \mathbf{y})\} \qquad \blacklozenge$$

The above definitions of the extended form of the state transition function and of the set of reachable states assume that enabled transitions fire *one at a time*. Consider the Petri net in Fig. 4.4 (b), where all three transitions t_1, t_2, and t_3 are enabled. We can see that transitions t_1 and t_2 could fire simultaneously, since they "consume" tokens from disjoint sets of places: $\{p_1\}$ for t_1 and $\{p_2, p_3\}$ for t_2. Since we are interested in all possible states that can be reached, and since later on we will be labeling transitions with event names and considering the languages represented by Petri nets, we shall henceforth exclude such simultaneous firings of transitions and assume that transitions fire one at a time.

State Equations

Let us return to (4.2) describing how the state value of an individual place changes when a transition fires. It is not difficult to see how we can generate a vector equation from (4.2), in order to specify the next Petri net state $\mathbf{x}' = [x'(p_1), x'(p_2), \ldots, x'(p_n)]$ given the current one $\mathbf{x} = [x(p_1), x(p_2), \ldots, x(p_n)]$ and the fact that a particular transition, t_j, has fired. To do so, let us first define the *firing vector* $\mathbf{u}$, an m-dimensional row vector of the form

$$\mathbf{u} = [0, \ldots, 0, 1, 0, \ldots, 0] \tag{4.3}$$

where the only 1 appears in the jth position, $j \in \{1, \ldots, m\}$, to indicate the fact that the jth transition is currently firing. In addition, we define the *incidence matrix* of a Petri net, $\mathbf{A}$, an $m \times n$ matrix whose (j, i) entry is of the form

$$a_{ji} = w(t_j, p_i) - w(p_i, t_j) \tag{4.4}$$

This matches the weight difference that appears in (4.2) in updating $x(p_i)$. Using the incidence matrix $\mathbf{A}$, we can now write a vector state equation

$$\mathbf{x}' = \mathbf{x} + \mathbf{uA} \tag{4.5}$$

which describes the state transition process as a result of an "input" $\mathbf{u}$, that is, a particular transition firing. The ith equation in (4.5) is precisely equation (4.2). We see, therefore, that $f(\mathbf{x}, t_j) = \mathbf{x} + \mathbf{uA}$, where $f(\mathbf{x}, t_j)$ is the transition function we defined earlier. The argument t_j in this function indicates that it is the jth entry in $\mathbf{u}$ which is nonzero. The state equation provides a convenient algebraic tool and an alternative to purely graphical means for describing the process of firing transitions and changing the state of a Petri net.

Example 4.5 (State equation)
Let us reconsider the Petri net of Fig. 4.4 (a), with the initial state $\mathbf{x}_0 = [2, 0, 0, 1]$. We can first write down the incidence matrix by inspection of the Petri net graph, which in this case is

$$\mathbf{A} = \begin{bmatrix} -1 & 1 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 0 & -1 & -1 \end{bmatrix}$$

The (1, 2) entry, for example, is given by $w(t_1, p_2) - w(p_2, t_1) = 1 - 0$. Using (4.5), the state equation when transition t_1 fires at state $\mathbf{x}_0$ is

$$\begin{aligned} \mathbf{x}_1 &= \begin{bmatrix} 2 & 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 & 1 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 0 & -1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} 2 & 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} -1 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \end{aligned}$$

which is precisely what we obtained in Example 4.4. Similarly, we can determine $\mathbf{x}_2$ as a result of firing t_2 next, as in Fig. 4.4 (c):

$$\begin{aligned} \mathbf{x}_2 &= \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} -1 & 1 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 0 & -1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 0 & 2 \end{bmatrix} \end{aligned}$$

Viewed as a dynamic system, a Petri net gives rise to sample paths similar to those of automata. Specifically, the sample path of a Petri net is a sequence of states $\{\mathbf{x}_0, \mathbf{x}_1, \ldots\}$ resulting from a transition firing sequence input $\{t^1, t^2, \ldots\}$, where t^k is the kth transition fired. Given an initial state $\mathbf{x}_0$, the entire state sequence can be generated through

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, t_k) = \mathbf{x}_k + \mathbf{u}_k \mathbf{A}$$

where $\mathbf{u}_k$ contains the information regarding the kth transition fired. If a state is reached such that no more transitions can fire, we say that the execution of the Petri net deadlocks at that state (as in Fig. 4.4 (d)).

4.2.4 Petri Net Languages

So far, we have focused on the state dynamics of Petri nets, where transitions are enumerated as elements of the set T. We have assumed that transitions correspond to events, but we have not made precise statements about this correspondence. If we wish to look at Petri nets as a modeling formalism for representing *languages*, as was the focus for automata in Chap. 2, then we need to specify precisely what event each transition corresponds to;

this will allow us to specify the language(s) represented (generated and marked) by a Petri net.

Let E be the set of events of the DES under consideration and whose language is to be modeled by a Petri net. Of course, we could require that the Petri net model of the system be such that each transition in T corresponds to a distinct event from the set of events E, and vice-versa. But this would be unnecessarily restrictive; in automata, we allow having two different arcs (originating from two different states) labeled with the same event. This leads us to the definition of a *labeled Petri net*.

Definition. (Labeled Petri net)
A *labeled Petri net* N is an eight-tuple

$$N = (P, T, A, w, E, \ell, \mathbf{x}_0, \mathbf{X}_m)$$

where

(P, T, A, w) is a Petri net graph

E is the event set for transition labeling

$\ell : T \to E$ is the transition labeling function

$\mathbf{x}_0 \in \mathbb{N}^n$ is the initial state of the net (i.e., the initial number of tokens in each place)

$\mathbf{X}_m \subseteq \mathbb{N}^n$ is the set of *marked states* of the net. ♦

In Petri net graphs, the label of a transition is indicated next to the transition. The notion of "marked states" in this definition is completely analogous to the notion of marked states in the definition of automaton in Chap. 2. We introduce marked states in order to define the language *marked* by a labeled Petri net.

Definition. (Languages generated and marked)
The language generated by labeled Petri net $N = (P, T, A, w, E, \ell, \mathbf{x}_0, \mathbf{X}_m)$ is

$$\mathcal{L}(N) := \{\ell(s) \in E^* : s \in T^* \text{ and } f(\mathbf{x}_0, s) \text{ is defined}\}$$

The language marked by N is

$$\mathcal{L}_m(N) := \{\ell(s) \in \mathcal{L}(N) : s \in T^* \text{ and } f(\mathbf{x}_0, s) \in \mathbf{X}_m)\}$$ ♦

(This definition uses the extended form of the transition labeling function $\ell : T^* \to E^*$; this extension is done in the usual manner.) We can see that these definitions are completely consistent with the corresponding definitions for automata. The language $\mathcal{L}(N)$ represents all strings of transition labels that are obtained by all possible (finite) sequences of transition firings in N, starting in the initial state $\mathbf{x}_0$ of N; the marked language $\mathcal{L}_m(N)$ is the subset of these strings that leave the Petri net in a state that is a member of the set of marked states given in the definition of N.

The class of languages that can be represented by labeled Petri nets is

$$\mathcal{PNL} := \{K \subseteq E^* : \exists N = (P, T, A, w, E, \ell, \mathbf{x}_0, \mathbf{X}_m)\ [\mathcal{L}_m(N) = K]\}$$

This is a general definition and the properties of $\mathcal{PNL}$ depend heavily on the specific assumptions that are made about ℓ (e.g., injective or not) and $\mathbf{X}_m$ (e.g., finite or infinite).

We shall restrict ourselves to the following situation: ℓ is not necessarily injective and $\mathbf{X}_m$ need not be finite. Section 4.3 compares Petri nets with automata, and in particular $\mathcal{R}$ with $\mathcal{PNL}$. Finally, we note that we shall often refer to "labeled Petri nets" as "Petri nets"; this will not create confusion with "unlabeled" Petri nets, as it will be clear from the context if we are interested in state properties (in which case event labeling of transitions is irrelevant) or in language properties.

4.2.5 Petri Net Models for Queueing Systems

In Example 2.10 in Sect. 2.2.3, we saw how automata can be used to represent the dynamic behavior of a simple queueing system. We now repeat this process using a Petri net structure. We begin by considering three events (transitions) driving the system:

a: customer arrives

s: service starts

c: service completes and customer departs.

We form the transition set $T = \{a, s, c\}$. (In this example, we shall not need to consider labeled Petri nets; equivalently, we can assume that $E = T$ and that ℓ is a one-to-one mapping between these two sets.)

Transition a is spontaneous and requires no conditions (input places). On the other hand, transition s depends on two conditions: the presence of customers in the queue, and the server being idle. We represent these conditions through two input places for this transition, place Q (queue) and place I (idle server). Finally, transition c requires that the server be busy, so we introduce an input place B (busy server) for it. Thus, our place set is $P = \{Q, I, B\}$.

The complete Petri net graph, along with the simple queueing system it models, are shown in Fig. 4.5 (a) and (b). No tokens are placed in Q, indicating that the queue is empty, and a token is placed in I, indicating that the server is idle. This defines the initial state $\mathbf{x}_0 = [0, 1, 0]$. Since transition a is always enabled, we can generate several possible sample paths. As an example, Fig. 4.5 (c) shows state $[2, 0, 1]$ resulting from the transition firing sequence $\{a, s, a, a, c, s, a\}$. This state corresponds to two customers waiting in queue, while a third is in service (the first arrival in the sequence has already departed after transition c).

Note that a c transition always enables an s transition as long as there is a token in Q. If we assume that the enabled s transition then immediately fires, this is equivalent to the automaton model using only two events (arrivals a and departures d), but specifying a feasible event set $\Gamma(0) = \{a\}$.

A slightly more detailed model of the same queueing system could include the additional transition

d: customer departs

which requires condition F (finished customer). In this case, transition c only means that "service completes." In addition, the external arrival process can also be represented by an input place to transition a, denoted by A, as long as a token is always maintained in A to keep transition a enabled. Thus, in this alternative model we have

$$T = \{a, s, c, d\} \quad \text{and} \quad P = \{A, Q, I, B, F\}$$

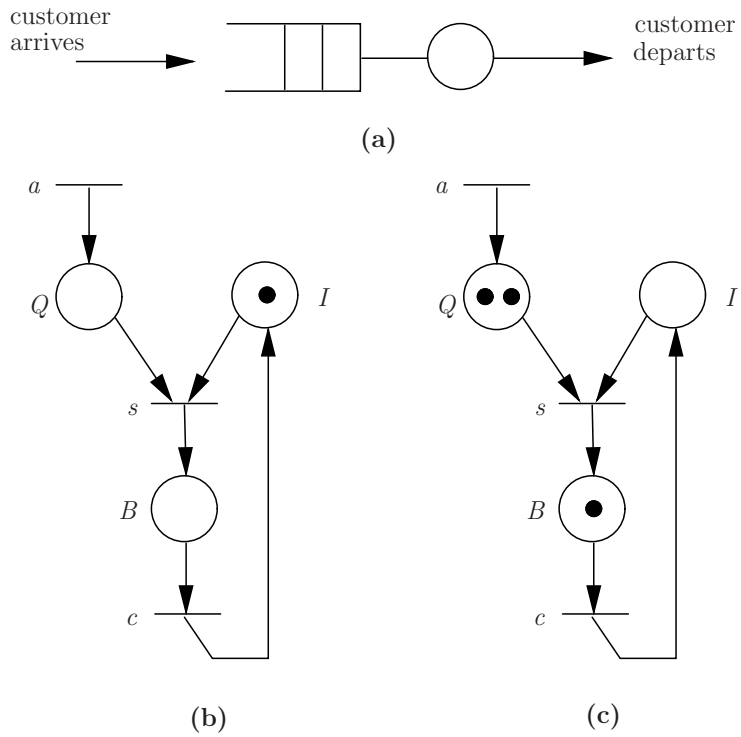

Figure 4.5: (**a**) Simple queueing system. (**b**) Petri net model of simple queueing system in initial state $[0, 1, 0]$. (**c**) Petri net model of simple queueing system with initial state $[0, 1, 0]$ after firing sequence $\{a, s, a, a, c, s, a\}$.

The resulting model is shown at state $[1, 0, 1, 0, 0]$ in Fig. 4.6 (a). Comparing this model with the automaton using only events $\{a, d\}$, we can see that events c and d are combined into one, since transition c is always enabling transition d. This is yet another indication of the flexibility of any modeling process; using the additional transition d may not be needed in general, but there are applications where differentiating between "service completion" of a customer and "customer departure" is useful, if not necessary.

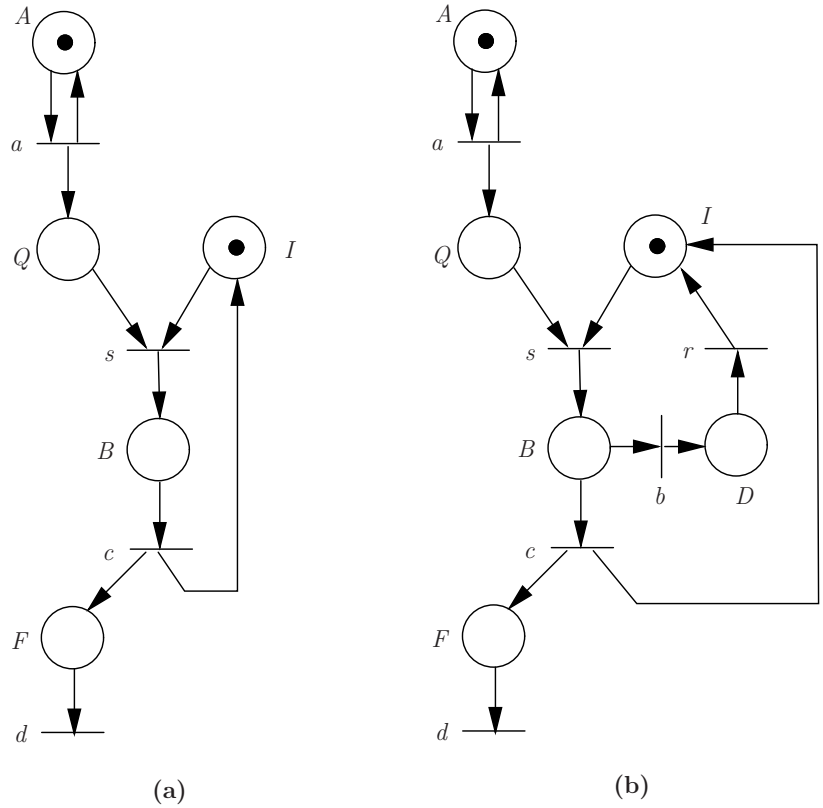

Figure 4.6: (**a**) An alternative Petri net model of the simple queueing system. (**b**) Petri net model of the simple queueing system with server breakdowns.

The model of Fig. 4.6 (a) can be further modified if we allow the server to break down (as in the automaton model of Fig. 2.7). In this case, we introduce two new transitions

b: server breaks down

r: server repaired

and place D (server down) which is input to transition r. Thus,

$$T = \{a, s, c, d, b, r\} \quad \text{and} \quad P = \{A, Q, I, B, F, D\}$$

The Petri net model is shown at state $[1, 0, 1, 0, 0, 0]$ in Fig. 4.6 (b).

4.3 COMPARISON OF PETRI NETS AND AUTOMATA

Automata and Petri nets can both be used to represent the behavior of a DES. We have seen that both formalisms explicitly represent the transition structure of the DES. In automata, this is done by explicitly enumerating all possible states and then "connecting" these states with the possible transitions between them, resulting in the transition function of the automaton. This is not particularly elegant, yet, automata are easily combined by operations such as product and parallel composition and therefore models of complex systems can be built from models of individual components in a systematic manner. Petri nets on the other hand have more structure in their representation of the transition function. States are not enumerated; rather, state information is "distributed" among a set of places that capture key conditions that govern the operation of the system. Of course, some work needs to be done at modeling time to identify all relevant "conditions" that should be captured by places, and then to properly connect these places to the transitions.

It is reasonable to ask the question: Which is a better model of a given DES, an automaton or a Petri net? There is no obvious answer to such a question, as modeling is always subject to personal biases and very frequently depends on the particular application considered. However, if we reformulate the above question more precisely in the context of specific criteria for comparison, it is possible to draw some conclusions.

Language Expressiveness

As a first criterion for comparing automata and Petri nets, let us consider the class of languages that can be represented by each formalism, when constrained to models that require *finite memory*, an obvious practical consideration. We claim that the class $\mathcal{PNL}$ is strictly larger than the class $\mathcal{R}$, meaning that Petri nets with finite sets of places and transitions can represent (i.e., mark) more languages in E^* than finite-state automata. In order to prove this result, let us first see how any *finite-state* automaton can always be "transformed" into a Petri net that generates and marks the same languages. Then we will complete the proof by presenting a non-regular language that can be marked by a Petri net.

Suppose we are given a finite-state automaton $G = (X, E, f_G, \Gamma, x_0, X_m)$, where X (and necessarily E) is a finite set. To construct a (labeled) Petri net $N = (P, T, A, w, E, \ell, \mathbf{x}_0, \mathbf{X}_m)$ such that

$$\mathcal{L}(N) = \mathcal{L}(G) \quad \text{and} \quad \mathcal{L}_m(N) = \mathcal{L}_m(G)$$

we can proceed as follows.

1. We first view each state in X as defining a unique place in P, that is, $P = X$. This also immediately specifies:
   - the initial state $\mathbf{x}_0$ of N: it is the row vector $[0, \ldots, 0, 1, 0, \ldots, 0]$ where the only non-zero entry is for the place in P that corresponds to $x_0 \in X$;
   - the set of marked states $\mathbf{X}_m$ of N: it is the set of all row vectors $[0, \ldots, 0, 1, 0, \ldots, 0]$ where the non-zero entry is for a place that corresponds to a marked state in X_m.
2. Next, we associate each triple (x, e, x') in G, where $x' = f_G(x, e)$ for some $e \in \Gamma(x)$, with a transition $t_{(x,e,x')}$ in T of N. In other words, T has the same cardinality as the set of arcs in the state transition diagram of G. We then:
   - label transition $t_{(x,e,x')}$ in T by the event $e \in E$;

- define two arcs in A for each (x, e, x') triple in G: arc $(x, t_{(x,e,x')})$ and arc $(t_{(x,e,x')}, x')$. All these arcs have weight equal to 1. This mapping of states and arcs in G to places, transitions, and arcs in N is illustrated in Fig. 4.7.

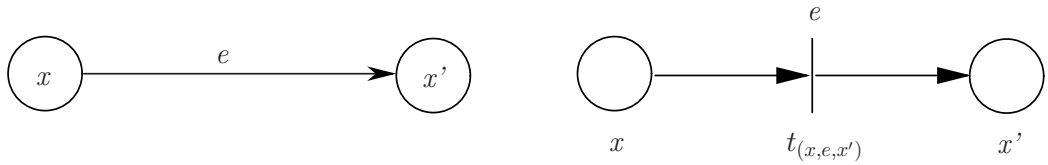

Figure 4.7: Mapping of a transition triple (x, e, x') in G to two places, one labeled transition, and two arcs in the graph of Petri net N.

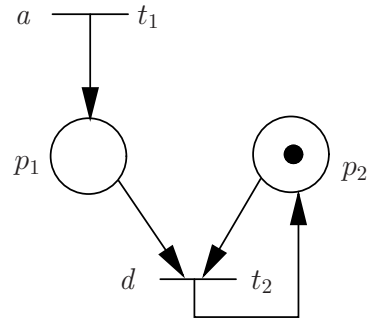

Figure 4.8: Modified Petri net model for queueing system in Fig. 4.5 (a).
This Petri net generates the same language as the automaton in Fig. 2.6, where we take $x_0 = 0$.

We should point out that this transformation technique for building a Petri net out of a given finite-state automaton is merely for the sake of substantiating our above claim relating $\mathcal{PNL}$ and $\mathcal{R}$ and may not result in the most intuitive Petri net model. Indeed, the physical system itself is likely to suggest much better ways for deriving the Petri net model by capturing the structural information in the system.

To prove that $\mathcal{PNL}$ is strictly larger than $\mathcal{R}$, let us return to the queueing system modeled in Fig. 4.5 and slightly modify it by removing place B and transition c. In addition, relabel the transition "service starts" by event d. The resulting Petri net is shown in Fig. 4.8. If we take $\mathbf{x}_0 = [0, 1]$ and $\mathbf{X}_m = \{[x, y] : x \in \mathbb{N} \text{ and } y \in \{0, 1\}\}$, then we can see that this Petri net marks the same *non-regular* language as the infinite-state automaton in Fig. 2.6 in Chap. 2, when $x_0 = 0$ and $X_m = \mathbb{N}$ for that automaton: the set of all strings in $\{a, d\}^*$ where any prefix of any string contains no more d events than a events. Intuitively, the memory required to ensure that any string does not contain more d events than a events and forces the automaton to have an arbitrarily large number of states (since there can be arbitrarily more a events than d events in a string) is taken care of by the tokens in place p_1. Of course, the number of tokens in place p_1 can be arbitrarily large. But this is not a problem in the sense that the Petri net graph requires finite memory (unlike explicit storage of the state transition diagram in Fig. 2.6); a variable can be used to store $x(p_1)$ as it is not necessary to explicitly show all possible values that $x(p_1)$ can take.

While not all Petri nets can be transformed into equivalent finite-state automata, it is clear that any Petri net N whose reachable state set $R(N)$ is finite can be transformed

into an equivalent finite-state automaton. It suffices to create an automaton state for each $\mathbf{x} \in R(N)$ and connect these automaton states by appropriate arcs according to the enabled transitions in $\mathbf{x}$, the states they reach, and their labels.

We conclude that the larger expressive power of Petri nets for languages as compared with finite-state automata occurs when the behavior of any Petri net representation of the given language results in an unbounded number of tokens in one or more places, leading to an infinite set of reachable Petri net states.

Modular Model-Building

Despite the potential complexity of Petri net graphs required to model even relatively simple DES, the Petri net formalism possesses some inherent advantages. One such advantage is its ability to decompose or modularize a potentially complex system. Suppose we have two interacting systems with state spaces X_1 and X_2 modeled as automata. If we combine these two systems into one, its state space, X, can be as large as all of the states in $X_1 \times X_2$, as was seen in Chap. 2; in particular, this upper bound will be achieved if the two systems have no common events. This means that combining multiple systems rapidly increases the complexity of automata. On the other hand, if the systems are modeled through Petri nets, the combined system is often easier to obtain by leaving the original nets as they are and simply adding a few places and/or transitions (or merging some places) representing the coupling effects between the two. Moreover, by looking at such a Petri net graph, one can conveniently see the individual components, discern the level of their interaction, and ultimately decompose a system into logical distinct modules. Example 4.6 below illustrates these claims and shows an instance where the model of the complete system grows "linearly" in the models of the system components. Of course, such a "linear" combination of sub-models into a complete model is a *modeling issue* and not a systematic procedure like combination by product and parallel composition in the case of automata. In fact, we can define product and parallel composition of Petri nets (although we shall not do so in this book) in order to systematically compose sub-models into a complete model; however, these composition operations will not in general result in linear growth.

Along the same lines, Petri nets are oriented towards capturing the concurrent nature of separate processes forming a DES. The combination of two such asynchronous processes in automaton models tends to become complex and hide some of the intuitive structure involved in this combination. Petri nets form a much more natural framework for these situations, and make it easier to visualize this structure.

Example 4.6 (Dining philosophers)
Let us consider the example of two "dining philosophers" that we described in Example 2.18 in Chap 2. Figures 2.18 and 2.19 show how this system is modeled using automata; the complete model is obtained by doing the parallel composition of four component models, one for each philosopher and one for each fork. Figure 4.9 shows a Petri net model of the same system, with the same set of events. In part (a) of Fig. 4.9, we have the model of one philosopher. There is a place corresponding to the condition "philosopher eating" and a place corresponding to the condition "philosopher thinking." This model also accounts for the availability or unavailability of each fork since it includes two "fork" places. In this sense, this model is more elaborate than automaton P_1 in Fig. 2.18. The reason for using the two fork places is to allow easy interconnection of the two philosophers. If we create a second copy of Fig. 4.9 (a) for the second philosopher, then a little thought shows that the complete system model

is obtained by combining the Petri net models of the two philosophers, where the fork places are simply *overlapped* while all other places, all transitions, and all arcs are preserved. This complete model is shown in Fig. 4.9 (b). This merger of the fork places works because the two forks are shared resources, and thus the conditions "fork available" and "fork unavailable" are common to the two component models. The initial state of the Petri net is obtained by placing four tokens, namely one in each of the places corresponding to: philosopher 1 thinking, philosopher 2 thinking, fork 1 available, and fork 2 available.

This example illustrates that while we could have constructed the complete Petri net model by transforming the finite-state automaton of the complete system model (two philosophers and two forks) shown in Fig. 2.19 into a Petri net using the method described earlier in this section, a model derived component by component from the original system description and associating places with conditions governing the operation of the system (as opposed to associating places with system states) is much more intuitive and modular.

Decidability

Another issue in comparing automata and Petri nets is that of decidability. We touched upon the notion of decidability at the end of Chap. 3 (see Sect. 3.8.6) and noted there that most decision problems involving finite-state automata can be solved algorithmically in finite time, i.e., they are decidable. Unfortunately, many problems that are decidable for finite-state automata are no longer decidable for Petri nets, reflecting a natural tradeoff between decidability and model-richness. One such example is the problem of language equivalence between two Petri nets.

Overall, it is probably most helpful to think of Petri nets and automata as complementary modeling approaches, rather than competing ones. As already pointed out, it is often the specific application that suggests which approach might be better suited.

4.4 ANALYSIS OF PETRI NETS

We begin by categorizing some of the most common problems related to the analysis of (untimed) Petri nets in Sect. 4.4.1. These problems are essentially related to the issues of safety and blocking that we considered in the case of automata in Sect. 2.5.1. However, the structural information contained in Petri nets is often used to pose more specific versions of these problems, such as boundedness, conservation, coverability, and persistence presented below. In Sects. 4.4.2 and 4.4.3 we will present the technique of the *coverability tree* to answer some the problems posed in Sect. 4.4.1.

4.4.1 Problem Classification

The following are some of the key issues related to the logical behavior of Petri nets. These issues relate primarily to desirable properties that often have their direct analogs in CVDS. Many of these properties are motivated by the fact that Petri nets are often used in resource sharing environments where we would like to ensure efficient and fair usage of the resources.

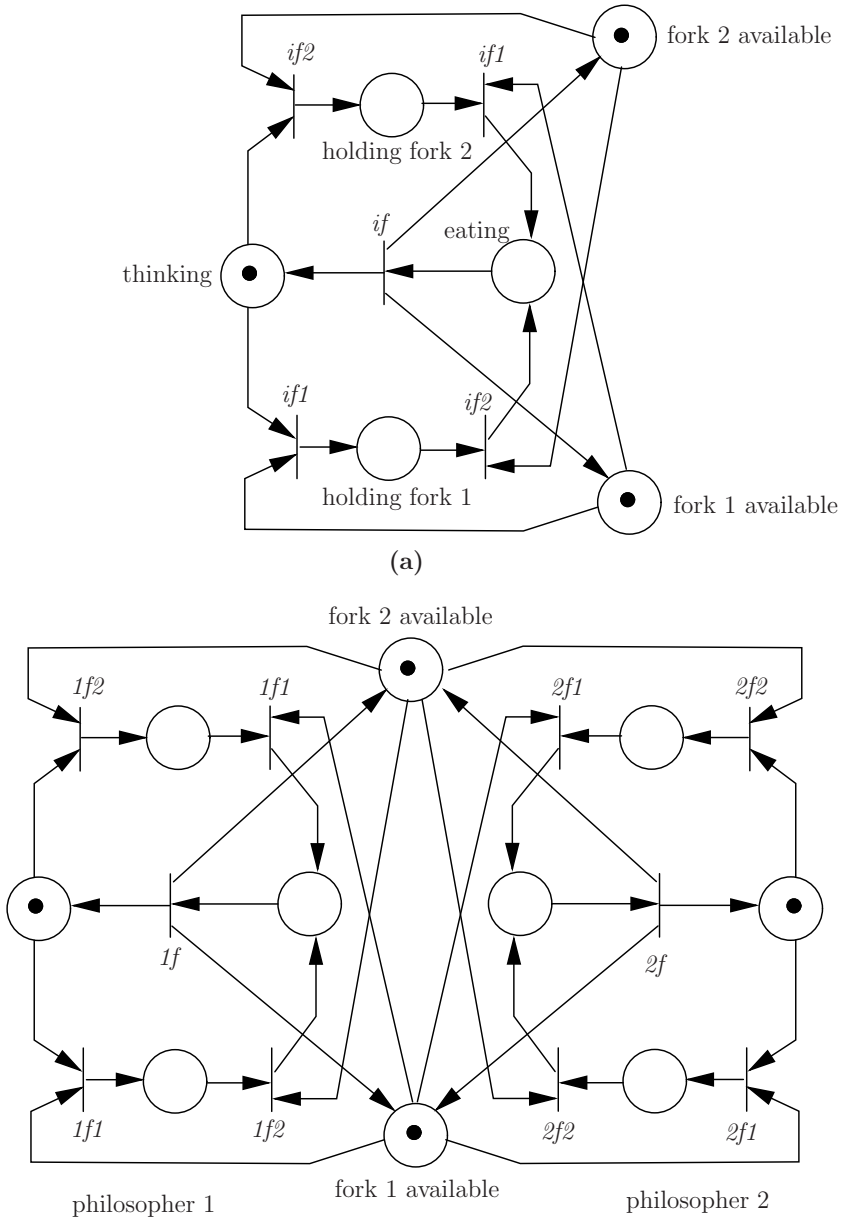

Figure 4.9: Petri net model of the dining philosophers example, with two philosophers and two forks. (Example 4.6).
Part (**a**) *shows the model of one philosopher interacting with two forks. Part* (**b**) *shows the complete model for two philosophers and two forks.*

Boundedness

In many instances, tokens represent customers in a resource sharing system. For example, the tokens in place Q of the Petri net in Fig. 4.5 represent customers entering a queue. Clearly, allowing queues to grow to infinity is undesirable, since it means that customers wait forever to access a server. In classical system theory, a state variable that is allowed to grow to infinity is generally an indicator of instability in the system. Similarly here, unbounded growth in state components (markings) leads to some form of instability.

Boundedness refers to the property of a place to maintain a number of tokens that never exceeds a given positive integer.

Definition. (Boundedness)
Place $p_i \in P$ in Petri net N with initial state $\mathbf{x}_0$ is said to be *k-bounded*, or *k-safe*, if $x(p_i) \leq k$ for all states $\mathbf{x} \in R(N)$, that is, for all reachable states. ♦

If a place is 1-bounded, it is simply called *safe*. If we can find some integer k, not necessarily prespecified, such that a place is k-bounded, then the place is said to be *bounded*. If all places in a Petri net are bounded, then the net is called bounded. The Petri net in Fig. 4.5 is not bounded since the number of tokens in place Q can become arbitrarily large.

Given a DES modeled as a Petri net, a boundedness problem consists of checking if the net is bounded and determining a bound. If boundedness is not satisfied, then our task may be to alter the model so as to ensure boundedness. If the Petri net is bounded, then as we mentioned previously it can be transformed into an equivalent finite-state automaton, allowing the application of the analysis techniques for finite-state automata, if so desired.

Safety and Blocking

These are the same issues that were posed in Sect. 2.5.1 in the context of automata. They can of course be posed for Petri nets as well. These issues can be posed in terms of states or in terms of languages, as in the case of automata. We could be concerned with the reachability of certain states or of certain substrings; we could ask whether there are states where the net deadlocks or whether the net is blocking from a language viewpoint; we could ask whether the behavior of a net is "contained" in the behavior of another net, either from the viewpoint of $R(N)$ or from the viewpoint of $\mathcal{L}(N)$; and so forth. If the Petri net of interest is bounded, then safety and blocking properties can be determined algorithmically, as we shall see in Sect. 4.4.3. (In essence, we build the equivalent finite-state automaton and answer the desired questions on this equivalent model.)

State Coverability

The concept of *state coverability* is a generalization of the concept of state reachability. It is also related to the concept of eventually being able to fire a particular transition. In order to enable a transition, it is often required that a certain number of tokens be present in some places. Consider, for instance, some state $\mathbf{y} = [y(p_1), y(p_2), \ldots, y(p_n)]$ which includes in each place the minimum number of tokens required to enable some transition t_j. Suppose we are currently in state $\mathbf{x}_0$, and we would like to see if t_j could eventually be enabled. It is, therefore, essential to know whether we can reach a state $\mathbf{x}$ from the current state $\mathbf{x}_0$ such that $x(p_i) \geq y(p_i)$ for all $i = 1, \ldots, n$. If this is the case, we say that state $\mathbf{x}$ *covers* state $\mathbf{y}$.

Definition. (State coverability)
Given Petri net N with initial state $\mathbf{x}_0$, state $\mathbf{y}$ is said to be *coverable* if there exists $\mathbf{x} \in R(N)$

such that $x(p_i) \geq y(p_i)$ for all $i = 1, \ldots, n$. ♦

Conservation

Sometimes tokens represent resources, rather than customers, in a resource sharing system. For example, the token in place I of the Petri net in Fig. 4.5 represents the lone server of a simple queueing system. This token is passed on to place B and subsequently returned to I in order to denote the changing state of the server (idle or busy). However, it can never be lost nor can the number of tokens in I or B increase, since there is only one server in our system.

Conservation is a property of Petri nets to maintain a fixed number of tokens for all states reached in a sample path. However, this may be too constraining a property. For instance, consider the Petri net model of the dining philosophers in Fig. 4.9. Initially, there are two "fork" tokens. When a philosopher is eating, these two tokens are replaced by a single token representing an eating philosopher. When the philosopher is done eating, the two fork tokens reappear. Thus, tokens representing resources are not exactly conserved all the time.

We therefore introduce a weighting vector $\boldsymbol{\gamma} = [\gamma_1, \gamma_2, \ldots, \gamma_n]$, with $\gamma_i \geq 0$ for all $i = 1, \ldots, n$, corresponding to places $p_1, p_2, \ldots, p_n$, and define conservation relative to these *weights* (not to be confused with the arc weights of the Petri net). We will also take a weight γ_i to always be an integer number; this is not absolutely necessary, and we do so for simplicity only.

Definition. (Conservation)
Petri net N with initial state $\mathbf{x}_0$ is said to be *conservative with respect to* $\boldsymbol{\gamma} = [\gamma_1, \gamma_2, \ldots, \gamma_n]$ if

$$\sum_{i=1}^{n} \gamma_i x(p_i) = \text{ constant} \tag{4.6}$$

for all states $\mathbf{x} \in R(N)$. ♦

Given a Petri net model of a DES, we are often required to ensure conservation with respect to certain weights representing the fact that resources are not lost or gained. More generally, the loss or gain of tokens must always reflect physical "conservation" properties of the DES we are modeling.

Liveness

A complement to the properties of deadlock and blocking is the notion of *live* transitions. Here, instead of being concerned with being unable to fire any transition or unable to eventually reach a marked state, we are concerned with being able to *eventually* fire a given transition.

Definition. (Liveness)
Petri net N with initial state $\mathbf{x}_0$ is said to be *live* if there always exists some sample path such that any transition can eventually fire from any state reached from $\mathbf{x}_0$. ♦

Clearly, this is a very stringent condition on the behavior of the system. Moreover, checking for liveness as defined above is an extremely tedious process, often practically infeasible for many systems of interest. This has motivated a further classification of liveness into four levels. Thus, given an initial state $\mathbf{x}_0$, a *transition* in a Petri net may be:

- *Dead* or *L0-live*, if the transition can never fire from this state;
- *L1-live*, if there is some firing sequence from $\mathbf{x}_0$ such that the transition can fire at least once;
- *L2-live*, if the transition can fire at least k times for some given positive integer k;
- *L3-live*, if there exists some infinite firing sequence in which the transition appears infinitely often;
- *Live* or *L4-live*, if the transition is L1-live for every possible state reached from $\mathbf{x}_0$.

The concept of coverability is closely related to that of L1-liveness. If $\mathbf{y}$ is the state that includes in each place the minimum number of tokens required to enable some transition t_j, then if $\mathbf{y}$ is not coverable from the current state, transition t_j is dead. Thus, it is possible to identify dead transitions by checking for coverability.

Example 4.7 (Liveness)
We illustrate some of the different liveness levels through the Petri net of Fig. 4.10. Here, transition t_2 is dead. This is because the only way it can fire is if a token is present in both p_1 and p_2, which can never happen (if t_1 fires, p_2 receives a token, but p_1 loses its token). Transition t_1 is L1-live, since it can fire, but only once. In fact, if t_1 fires, then all transitions become dead in the new state. Finally, transition t_3 is L3-live, since it is possible for it to fire infinitely often. It is not, however, L4-live, since it can become dead in the state resulting from t_1 firing.

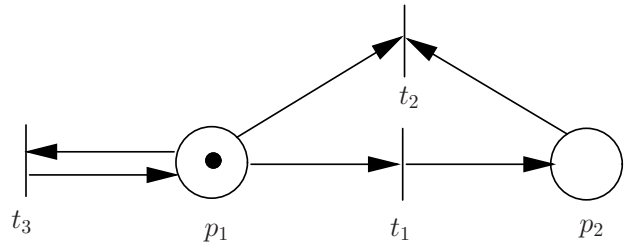

Figure 4.10: Petri net for Example 4.7.
Transition t_2 is dead, because it can never fire. Transition t_1 is L1-live, since it can fire once. Transition t_3 is L3-live, because it can fire an infinite number of times, but it is not L4-live, since it becomes dead if t_1 fires.

Persistence

It is sometimes the case that two different transitions are enabled by the same set of conditions. If one fires, does the other one remain enabled? In general, there is no guarantee that this is so. Actually, the two transitions do not have to depend on the exact same conditions, but only have one condition in common. Persistence is the property of a Petri net not to disable any enabled transition because of the firing of another enabled transition.

Definition. (Persistence)
A Petri net is said to be *persistent* if, for any two enabled transitions, the firing of one cannot disable the other. ♦

An example of a Petri net that is not persistent is the net of Fig. 4.10. While both t_1 and t_3 are enabled in the state shown, if t_1 occurs, then t_3 is no longer enabled. Similarly, the Petri net of Fig. 4.9 is not persistent. On the other hand, the Petri net model of a simple queueing system (Fig. 4.5) can easily be seen to be persistent.

It turns out that persistence is equivalent to a property referred to as *non-interruptedness* in the study of timed DES models. To better understand the choice of the term "non-interruptedness," think of each enabled transition as having to go through some process before it can actually fire, hence incurring some delay. If the firing of a transition disables another enabled transition, it effectively "interrupts" this process. In many ways the absence of the persistence property introduces technical complications which we can roughly compare to nonlinear phenomena in the study of CVDS. We will have the chance to return to this point in Chap. 11 and further elaborate on non-interruptedness.

4.4.2 The Coverability Tree

The *coverability tree* is an analysis technique that may be used to solve some, but not all, of the problems described above. This technique is based on the construction of a tree where nodes are Petri net states and arcs represent transitions. The key idea is quite simple and is best illustrated through some examples.

Example 4.8 (Simple reachability tree)
Consider the Petri net of Fig. 4.11, with initial state $[1, 1, 0]$. Also shown in the same figure is a tree whose root node is $[1, 1, 0]$. We then examine all transitions that can fire from this state, define new nodes in the tree, and repeat until all possible reachable states are identified. In this simple case, the only transition that can initially fire is t_1, and the next state is $[0, 0, 1]$. Now t_2 can fire, and the next state is $[1, 1, 0]$. Since this state already exists (it is the root node), we stop there. In essence, we have constructed an equivalent automaton to the Petri net, although we duplicated state $[1, 1, 0]$ and stopped there; in the equivalent finite-state automaton, the arc labeled t_2 out of state $[0, 0, 1]$ goes back to the root node.

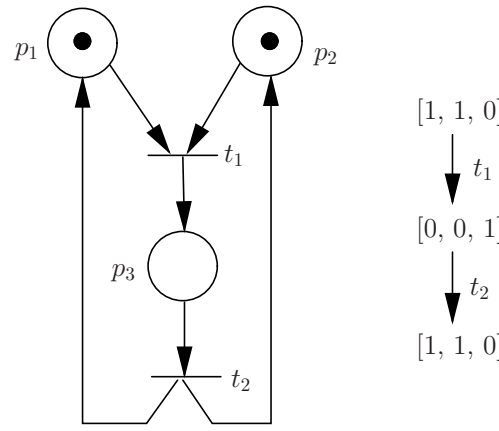

Figure 4.11: Reachability tree for Example 4.8.
The construction of the reachability tree stops at state (node) $[1, 1, 0]$*, since this state is a duplicate of the initial state of the tree.*

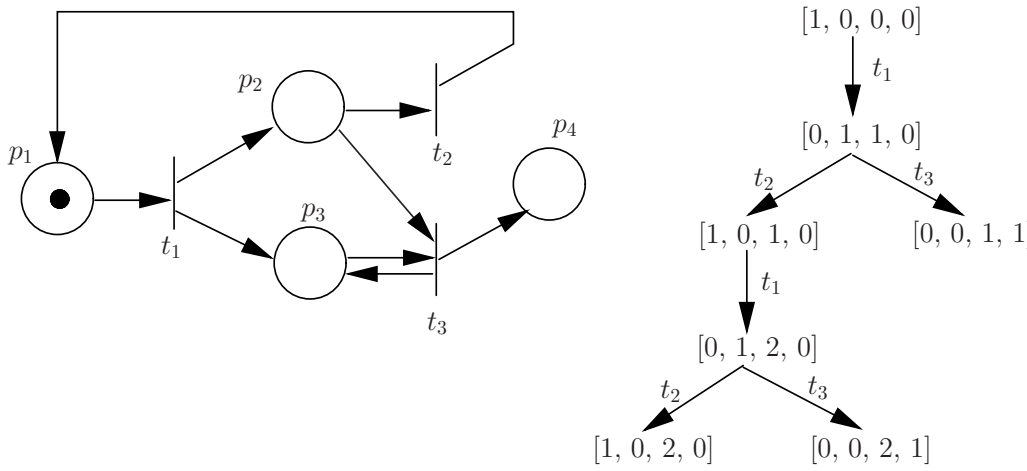

Figure 4.12: Part of the reachability tree for Example 4.9.
The construction of the tree stops at states that are terminal (deadlock) states or that dominate states encountered along the same branch from the root of the tree.

Example 4.9 (Infinite reachability tree)
Consider the Petri net of Fig. 4.12, with initial state $[1, 0, 0, 0]$. The reachability tree starts out with this state, from which only transition t_1 can fire. As shown in Fig. 4.12, the next state is $[0, 1, 1, 0]$. Now either t_2 or t_3 can fire, so we create two branches, with corresponding next states $[1, 0, 1, 0]$ and $[0, 0, 1, 1]$, respectively. Note that no transition can fire at state $[0, 0, 1, 1]$, that is, this state is a deadlock or *terminal* state. (We will use the term "terminal" instead of "deadlock" since the former is more frequently used in the literature when discussing the coverability tree.) The left branch allows transition t_1 to fire, resulting in new state $[0, 1, 2, 0]$. Once again, either t_2 or t_3 can fire, and the corresponding next states are $[1, 0, 2, 0]$ and $[0, 0, 2, 1]$. This is sufficient to detect the pattern of this tree. Every right branch eventually leads to a terminal state, but with one additional token in place p_3. Every left branch repeats itself, but with one extra token in place p_3. In other words, we see that place p_3 is unbounded, meaning that the reachability tree is infinite, since the set of reachable states is infinite. However, we do not build this infinite tree. Rather, upon observing that state $[1, 0, 2, 0]$ is larger, component-wise, than state $[1, 0, 1, 0]$ that precedes it in the path from the root to $[1, 0, 2, 0]$, we stop at state $[1, 0, 2, 0]$. We say that state $[1, 0, 2, 0]$ *dominates* state $[1, 0, 1, 0]$.

As Examples 4.8 and 4.9 suggest, the reachability tree is conceptually easy to construct. It may, however, be infinite. Our task, therefore, is to seek a finite representation of this tree. This is possible, but at the expense of losing some information. The finite version of an infinite reachability tree will be called a *coverability tree*.

We will present an algorithm for constructing a finite coverability tree. To do so, we first introduce some notation:

1. *Root node.* This is the first node of the tree, corresponding to the initial state of the given Petri net. For example, $[1, 0, 0, 0]$ is the root node in the tree of Fig. 4.12.

2. *Terminal node.* This is any node from which no transition can fire. For example, in the tree of Fig. 4.12, $[0, 0, 1, 1]$ is a terminal node.

3. *Duplicate node.* This is a node that is identical to a node already in the tree. Often, this definition requires that the identical node be in the path from the root to the node under consideration. For example, in the tree of Fig. 4.11, node $[1, 1, 0]$ resulting from the sequence $t_1 t_2$ is a duplicate node of the root node.

4. *Node dominance.* Let $\mathbf{x} = [x(p_1), \ldots, x(p_n)]$ and $\mathbf{y} = [y(p_1), \ldots, y(p_n)]$ be two states, i.e., nodes in the coverability tree. We will say that "$\mathbf{x}$ *dominates* $\mathbf{y}$," denoted by $\mathbf{x} >_d \mathbf{y}$, if the following two conditions hold:

   (a) $x(p_i) \geq y(p_i)$, for all $i = 1, \ldots, n$

   (b) $x(p_i) > y(p_i)$, for at least some $i = 1, \ldots, n$.

   For example, in the tree of Fig. 4.12 we have $[1, 0, 2, 0] >_d [1, 0, 1, 0]$. But $[1, 0, 2, 0]$ does not dominate $[0, 1, 1, 0]$. Note that condition (a) above is the definition of coverability for states $\mathbf{x}$, $\mathbf{y}$; however, dominance requires the additional condition (b). Thus, dominance corresponds to "strict" covering.

5. *The symbol ω.* This may be thought of as "infinity" in representing the marking (state component) of an unbounded place. We use ω when we identify a node dominance relationship in the coverability tree. In particular, if $\mathbf{x} >_d \mathbf{y}$, then for all i such that $x(p_i) > y(p_i)$ we replace the value of $x(p_i)$ by ω. Note that adding or subtracting tokens to a place which is already marked by ω, does not have any effect, that is, $\omega \pm k = \omega$ for any $k = 0, 1, 2, \ldots$ As an example, in Fig. 4.12 we have $[1, 0, 1, 0] >_d [1, 0, 0, 0]$. We can then replace $[1, 0, 1, 0]$ by $[1, 0, \omega, 0]$.

Coverability Tree Construction Algorithm

Step 1: Initialize $\mathbf{x} = \mathbf{x}_0$ (initial state).

Step 2: For each new node, $\mathbf{x}$, evaluate the transition function $f(\mathbf{x}, t_j)$ for all $t_j \in T$:

Step 2.1: If $f(\mathbf{x}, t_j)$ is undefined for all $t_j \in T$ (i.e., no transition is enabled at state x), then mark $\mathbf{x}$ as a terminal node.

Step 2.2: If $f(\mathbf{x}, t_j)$ is defined for some $t_j \in T$, create a new node $\mathbf{x}' = f(\mathbf{x}, t_j)$. If necessary, adjust the marking of node $\mathbf{x}'$ as follows:

Step 2.2.1: If $x(p_i) = \omega$ for some p_i, set $x'(p_i) = \omega$.

Step 2.2.2: If there exists a node $\mathbf{y}$ in the path from the root node $\mathbf{x}_0$ (included) to $\mathbf{x}$ such that $\mathbf{x}' >_d \mathbf{y}$, set $x'(p_i) = \omega$ for all p_i such that $x'(p_i) > y(p_i)$.

Step 2.2.3: Otherwise, $x'(p_i)$ is as obtained in $f(\mathbf{x}, t_j)$.

If node $\mathbf{x}'$ is identical to a node in the path from $\mathbf{x}_0$ to $\mathbf{x}$, then mark $\mathbf{x}'$ as a duplicate node.

Step 3: If all new nodes have been marked as either terminal or duplicate nodes, then stop.

It can be shown that the coverability tree constructed by this algorithm is indeed finite. The proof (which is beyond the scope of this book) is based on a straightforward contradiction argument making use of some elementary properties of trees and of sequences of non-negative integers.

Example 4.10 (Coverability tree of infinite reachability tree)
Consider once again the Petri net of Fig. 4.12, with initial state $\mathbf{x}_0 = [1, 0, 0, 0]$. We construct its coverability tree by formally following the steps of the algorithm above, as shown in Fig. 4.13.

The first transition fired results in $\mathbf{x}' = f(\mathbf{x}_0, t_1) = [0, 1, 1, 0]$. If transition t_3 fires next, then we obtain the terminal node [0, 0, 1, 1]. If transition t_2 fires, we obtain the new node $[1, 0, 1, 0]$. Going through Step 2.2.2 of the algorithm, we observe that $[1, 0, 1, 0] >_d [1, 0, 0, 0]$, and therefore we replace $[1, 0, 1, 0]$ by $[1, 0, \omega, 0]$. When transition t_1 fires next, by Step 2.2.1, the symbol ω remains unchanged. The new node, $[0, 1, \omega, 0]$, dominates node $[0, 1, 1, 0]$ in the path from the root to $[1, 0, \omega, 0]$. However, the symbol ω has already been used in place p_3. We continue with transition t_3, which leads to the terminal node $[0, 0, \omega, 1]$, and with t_2, which leads to the duplicate node $[1, 0, \omega, 0]$. This completes the construction of the coverability tree.

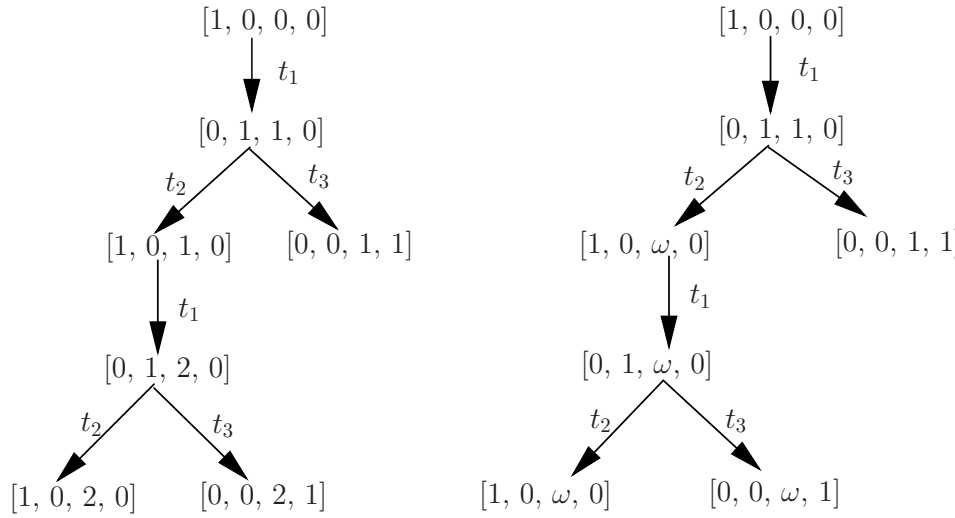

Figure 4.13: Coverability tree construction for Example 4.10.

Example 4.11 (Coverability tree of a simple queueing system)
Let us construct the coverability tree of the Petri net model of the simple queueing system shown in Fig. 4.5 and redrawn in Fig. 4.14. Assume that we start at the initial state $\mathbf{x}_0 = [0, 1, 0]$ (no customer in system), where the elements of $\mathbf{x}_0$ correspond to places Q, I, and B, respectively. Transition a fires and the next state is $[1, 1, 0]$. Observing that $[1, 1, 0] >_d [0, 1, 0]$, we replace $[1, 1, 0]$ by $[\omega, 1, 0]$ (see Fig. 4.14). Two transitions are now enabled. If a fires, the next state is the duplicate node $[\omega, 1, 0]$. If s fires, the next state is $[\omega, 0, 1]$, which is neither a duplicate nor a terminal node. If transition a fires next, the new state is the duplicate node $[\omega, 0, 1]$. If c fires, the new state is the duplicate node $[\omega, 1, 0]$, and the coverability tree construction is complete.

4.4.3 Applications of the Coverability Tree

In Sect. 4.4.1, we presented several problems related to the analysis of Petri net models of untimed DES. Let us now see how some of these problems can be solved by using the

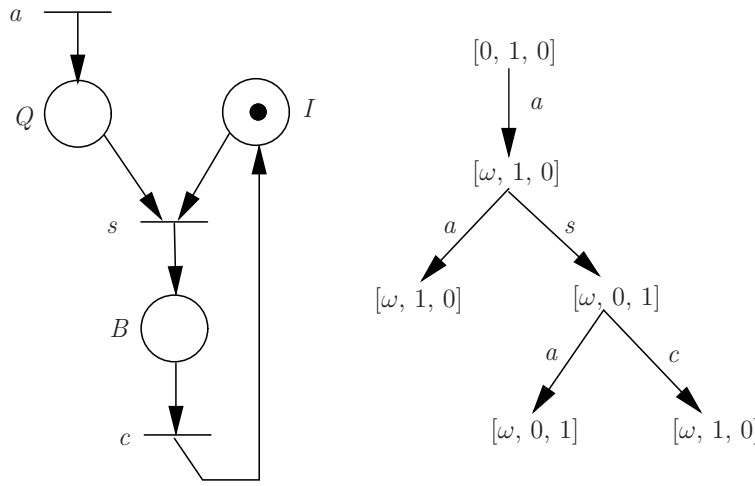

Figure 4.14: Coverability tree construction for a simple queueing system (Example 4.11).

coverability tree. In the process, we will also identify the limitations of this approach.

Boundedness, Safety, and Blocking Problems

The problem of boundedness is easily solved using a coverability tree. A necessary and sufficient condition for a Petri net to be bounded is that the symbol ω never appears in its coverability tree. Since ω represents an infinite number of tokens in some place, if ω appears in place p_i, then p_i is unbounded. For example, in Fig. 4.14, place Q is unbounded; this is to be expected, since there is no limit to the number of customers that may reside in the queue at any time instant.

If ω never appears in the coverability tree, then we are guaranteed that the state space of the DES we are modeling is finite. The coverability tree is then the reachability tree, since it contains all the reachable states. Analysis of such systems becomes much easier to handle, since we can examine all possible states and transitions between them. For instance, to answer safety questions concerning states (including blocking questions), it suffices to examine the finite reachability tree to determine if any of the illegal states is reachable. For safety questions that are posed in language terms (i.e., illegal substrings) one would want to indicate the labels of the transitions in the reachability tree in order to answer these questions; since the finite reachability tree is essentially a finite-state automaton, the discussion in Sect. 2.5.1 applies here as well.

Finally, note that if ω does not appear in place p_i, then the largest value of $x(p_i)$ for any state encountered in the tree specifies a bound for the number of tokens in p_i. For example, $x(I) \leq 1$ in Fig. 4.14. Thus, place I is 1-bounded (or safe). If the coverability (reachability) tree of a Petri net contains states with 0 and 1 as the only place markings, then all places are guaranteed to be safe, and the Petri net is safe.

Coverability Problems

By simple inspection of the coverability tree, it is always possible to determine whether some state $\mathbf{y}$ of a given Petri net with initial state $\mathbf{x}_0$ is coverable or not. We construct the

coverability tree starting with $\mathbf{x}_0$. If we can find a node $\mathbf{x}$ in the tree such that $x(p_i) \geq y(p_i)$ for all $i = 1, \ldots, n$, then $\mathbf{y}$ is coverable. The path from the root $\mathbf{x}_0$ to $\mathbf{x}$ describes the firing sequence that leads to the covering state $\mathbf{x}$. However, if $\mathbf{x}$ contains ω in one or more places, we should realize that this path must include a loop. Depending on the particular values of $y(p_i)$, $i = 1, \ldots, n$, that need to be covered, we can then determine the number of loops involved in this path until $\mathbf{y}$ is covered.

As an example, state $[3, 1, 0]$ in the Petri net of Fig. 4.14 is coverable, since $[\omega, 1, 0]$ is reachable from the root node $[0, 1, 0]$; in fact, $[\omega, 1, 0]$ appears in three separate positions in the tree. Upon closer inspection, we can see that there are several firing sequences that could lead to this state, such as $\{a, a, a\}$ or $\{a, s, a, c, s, a, c, a, a\}$.

Conservation Problems

This is also a problem that can be solved using the coverability tree. Recall that conservation with respect to a weighting vector $\boldsymbol{\gamma} = [\gamma_1, \gamma_2, \ldots, \gamma_n]$, with γ_i a non-negative integer, is defined by condition (4.6), which must be checked for all reachable states. First, we need to observe that if $x(p_i) = \omega$ for some p_i, then we must have $\gamma_i = 0$ if the Petri net is to be conservative. This is because ω does not represent a fixed integer value, but rather an infinite set of values. Suppose we consider two states $\mathbf{x}$ and $\mathbf{y}$ such that $x(p_i) = y(p_i)$ for all p_i, and $x(p_k) = y(p_k) = \omega$ for some p_k. The *actual* values of $x(p_k)$ and $y(p_k)$ may be different in general, since ω simply denotes infinitely many integer values (cf. Example 4.12 hereafter). As a result, if $\omega_k > 0$, we cannot satisfy

$$\sum_{i=1}^{n} \gamma_i x(p_i) = \sum_{i=1}^{n} \gamma_i y(p_i)$$

With this observation in mind, if a vector $\boldsymbol{\gamma}$ is specified such that $\gamma_i = 0$ for all i with $x(p_i) = \omega$ somewhere in the coverability tree, then we can check for conservation by evaluating the weighted sum above for each state in the coverability tree. Conservation requires that this sum be fixed.

Conversely, we can pose the following problem. Given a Petri net, is there a weighting vector $\boldsymbol{\gamma}$ such that the conservation condition (4.6) is satisfied? To solve this problem, we first set $\gamma_i = 0$ for all unbounded places p_i. Next, suppose there are $b \leq n$ bounded places, and let the coverability tree consist of r nodes. Finally, let the value of the constant sum in condition (4.6) be C. We then have a set of r equations of the form

$$\sum_{i=1}^{b} \gamma_i x(p_i) = C \qquad \text{for each state } \mathbf{x} \text{ in the coverability tree}$$

with $(b + 1)$ unknowns, namely, the b strictly positive weights for the bounded places and the constant C. Thus, this conservation problem reduces to a standard algebraic problem for solving a set of linear equations. It is not necessary for a solution to exist, but if one or more solutions do exist, then it is possible to determine them by standard techniques.

As an example, consider the coverability tree of Fig. 4.14, where we set $\gamma_1 = 0$ for unbounded place Q. Then we are left with $r = 6$ equations of the form

$$\gamma_2 x(p_2) + \gamma_3 x(p_3) = C$$

and the three unknowns: γ_2, γ_3, and C. It is easy to see that setting $\gamma_2 = \gamma_3 = 1$ and $C = 1$ yields a conservative Petri net with respect to $\boldsymbol{\gamma} = [0, 1, 1]$. This merely reflects the fact that the token representing the server in this queueing system is always conserved.

Coverability Tree Limitations

Safety properties are in general not solvable using the coverability tree for unbounded Petri nets. The reason is that the symbol ω represents a set of reachable states, but it does not explicitly specify them. In other words, ω is used to aggregate a set of states so that a convenient *finite* representation of the *infinite* set of reachable states can be obtained; however, in the process of obtaining this finite representation, some information is lost. This "loss of information" is illustrated by the following example.

Example 4.12 (Loss of information in coverability tree)
This example illustrates the loss of information regarding specific reachable states when a coverability tree is constructed for an unbounded Petri net. In Fig. 4.15 (a) we show a simple Petri net and its coverability tree. Observe that the first time transition t_2 fires the new state is $[1, 0, 1]$. The next time transition t_2 fires the new state becomes $[1, 0, 2]$, and so on. Thus, the reachable markings for place p_3 are $x(p_3) = 1, 2, \ldots$ All those are aggregated under the symbol ω.

In Fig. 4.15 (b), we show a similar – but different – Petri net along with its coverability tree. Note that in the Petri net of Fig. 4.15 (b), the first time transition t_2 fires the new state is $[1, 0, 2]$. The next time transition t_2 fires the new state becomes $[1, 0, 4]$, and so on. The reachable markings for p_3 in this case are $x(p_3) = 2, 4, \ldots$, and not all positive integers as in the Petri net of Fig. 4.15 (a). Despite this difference, the coverability trees in Fig. 4.15 (a) and (b) are identical. The reason is the inherent vagueness of ω.

As Example 4.12 illustrates, state reachability issues cannot be dealt with, in general, using a coverability tree approach. These limitations of the coverability tree also affect the analysis of blocking properties. (Further discussion on blocking and trimming of blocking Petri nets can be found in Sect. 4.5.1.)

Sometimes, the coverability tree does contain sufficient information to solve certain problems. It is often possible, for instance, to determine that some state is not reachable, while being unable to check if some other state is reachable. We can also limit general Petri nets to certain classes which are easier to analyze, yet still rich enough to model several interesting types of DES we encounter in practice. The class of bounded Petri nets is an example. Several other interesting classes have been identified in the literature, based on the structural properties of the Petri net graph. One such class is the class of *marked graphs*, which are Petri nets where all arc weights are 1 and where each place has a single input arc and a single output arc (thus the place has only one input transition and only one output transition). Marked graphs are also called *event graphs* or *decision-free Petri nets*. We will consider this important class in the next chapter, when we are faced with analysis problems related to event timing; this will lead us to the (max,+) algebra as an analytical tool for certain timed DES. Several analytical results are also available for untimed marked graphs. We shall not cover these results in this book; the interested reader may consult the references mentioned at the end of this chapter.

4.4.4 Linear-Algebraic Techniques

We conclude our discussion on the analysis of Petri nets by mentioning that the state equation (4.5) that we derived in Sect. 4.2.3 can be used to address problems such as state reachability and conservation. It provides an algebraic alternative to the graphical

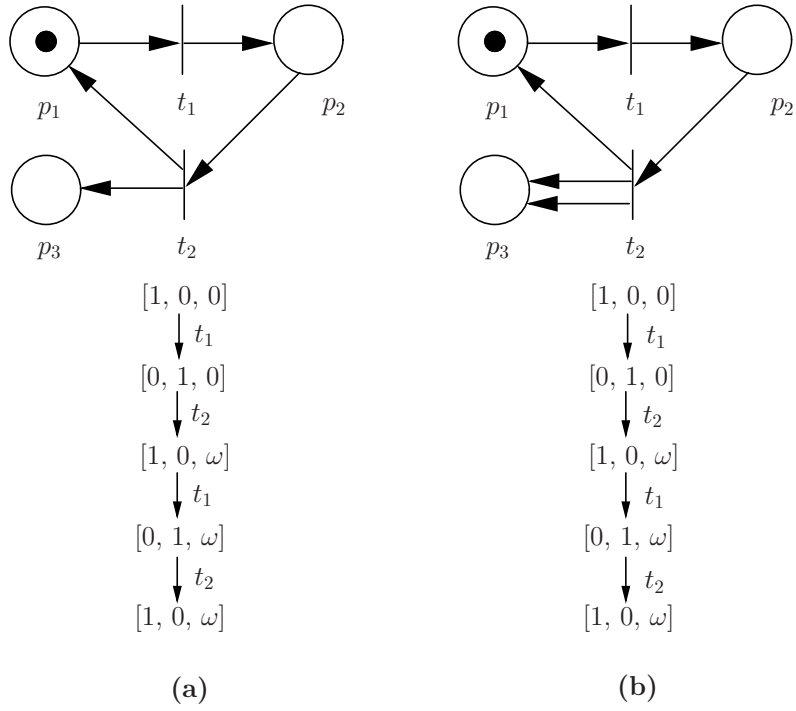

Figure 4.15: Two unbounded Petri nets and their coverability trees (Example 4.12).
In (**a**)*, the symbol* ω *for place* p_3 *represents the set* $\{1, 2, \ldots\}$*, while in* (**b**)*,* ω *for the same place represents the set* $\{2, 4, \ldots\}$*.*

methodology based on the coverability tree, and it can be quite powerful in identifying structural properties that are mostly dependent on the topology of the Petri net graph captured in the incidence matrix $\mathbf{A}$.

With regard to reachability for instance, by looking at (4.5), we can see that a necessary condition for state $\mathbf{x}$ to be reachable from initial state $\mathbf{x}_0$ is for the equation

$$\mathbf{v}\mathbf{A} = \mathbf{x} - \mathbf{x}_0$$

to have a solution $\mathbf{v}$ where all the entries of $\mathbf{v}$ are non-negative integers. Such a vector $\mathbf{v}$ is called a *firing count vector*, since its components specify the number of times that each transition fires in "attempting" to reach state $\mathbf{x}$ from $\mathbf{x}_0$. However, this necessary condition is not in general sufficient, as the existence of a non-negative integer solution $\mathbf{v}$ does not guarantee that the entries in the firing count vector $\mathbf{v}$ can be mapped to an actual feasible ordering of individual transition firings.[1] Consider Example 4.13.

Example 4.13

The Petri net in Fig. 4.16 has incidence matrix

$$\mathbf{A} = \begin{bmatrix} -1 & 1 & -1 & 0 \\ 0 & -1 & 1 & 1 \\ 1 & 0 & 0 & -1 \end{bmatrix}$$

and initial state $\mathbf{x}_0 = [1, 0, 0, 0]$. Consider state $\mathbf{x} = [0, 0, 0, 1]$. We can see that the system of equations

$$\mathbf{v}\mathbf{A} = \mathbf{x} - \mathbf{x}_0 = [-1, 0, 0, 1]$$

has a solution $\mathbf{v} = [1, 1, 0]$. While the entries of $\mathbf{v}$ are indeed non-negative integers, neither t_1 nor t_2 are enabled in state $\mathbf{x}_0$. Thus none of the two possible orderings of transition firings consistent with $\mathbf{v}$, $t_1 t_2$ and $t_2 t_1$, are feasible from $\mathbf{x}_0$.

On the positive side, if we are interested in the reachability of $\mathbf{x}' = [0, 1, 0, 0]$ from $\mathbf{x}_0$, then the fact that the system of equations

$$\mathbf{v}\mathbf{A} = \mathbf{x}' - \mathbf{x}_0 = [-1, 1, 0, 0]$$

does not have a solution allows us to conclude that $\mathbf{x}'$ is not reachable from $\mathbf{x}_0$.

Examination of the state equation (4.5) can also yield some information regarding conservation properties. Let $\mathbf{A}$ be the incidence matrix of a Petri net. A non-zero $n \times 1$ vector[2] $\boldsymbol{\ell} = [\ell_1, \ldots, \ell_n]^T$ such that

$$\mathbf{A}\boldsymbol{\ell} = 0$$

is called a *place-invariant* of the net structure represented by $\mathbf{A}$. If $\boldsymbol{\ell}$ is a place-invariant, then starting from the state equation of the net we can write

$$\begin{aligned} \mathbf{x} &= \mathbf{x}_0 + \mathbf{v}\mathbf{A} \\ \mathbf{x}\boldsymbol{\ell} &= \mathbf{x}_0\boldsymbol{\ell} + \mathbf{v}\mathbf{A}\boldsymbol{\ell} \\ \mathbf{x}\boldsymbol{\ell} &= \mathbf{x}_0\boldsymbol{\ell} \end{aligned}$$

[1] This explains the use of the word "attempting" in the preceding sentence.

[2] T denotes transposed.

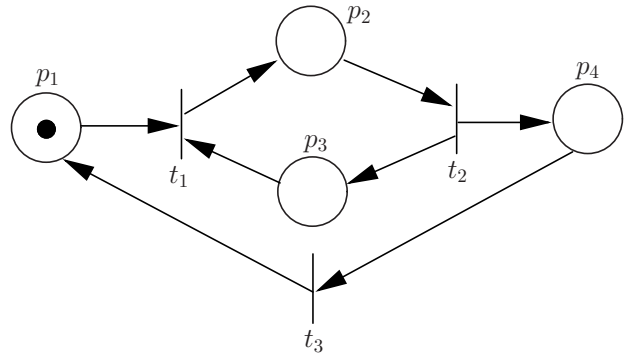

Figure 4.16: Petri net for Example 4.13.

Since this last equality holds for all states $\mathbf{x}$ that are reachable from $\mathbf{x}_0$, this means that if $\ell_i \geq 0$ for $i = 1, \ldots, n$, then the Petri net is conservative with respect to the (row) vector of weights $\boldsymbol{\gamma} := \boldsymbol{\ell}^T$, *for any choice* of $\mathbf{x}_0$.

Many more results are available in the literature regarding structural analysis of Petri nets and subclasses of Petri nets, such as marked graphs, using linear-algebraic techniques. We shall not, however, go into further details on this approach in this book.

4.5 CONTROL OF PETRI NETS

There is large body of literature on the control of Petri nets. We shall limit ourselves to a brief treatment of this topic. In Chap. 3, we presented a comprehensive control theory for DES, known as "supervisory control theory." The first question that one may ask is: Can Petri nets be used instead of automata as the modeling formalism in the context of supervisory control theory? The key system-theoretic properties (among them, controllability, nonconflict, observability, coobservability) studied in Chap. 3 are language properties and therefore they are independent of the particular modeling formalism used to represent languages. The algorithms presented in Chap. 3 for verification of these properties and for computation of supervisors assume that the languages of interest (uncontrolled system language and admissible language) are regular and they are based on finite-state automata representations of these languages. As was seen in that chapter, the class of regular languages possesses nice properties in the context of supervisory control; in particular, this class is closed under the supervisor synthesis operations of interest: $\uparrow C$, $\downarrow C$, $\downarrow O$, $\uparrow N$, and so forth. Moreover, we saw that automata were appropriate for the implementation of these synthesis operations. These closure properties as well as the fact that finitely-convergent algorithms exist for the principal supervisor synthesis operations are results of great practical importance.

Clearly, it is also of interest to consider supervisory control problems where Petri nets are used to model the uncontrolled system, the admissible language, and/or to realize the supervisor. That is, using the notation of Chap. 3, G could be given as a (labeled) Petri net, L_{am} could be represented by a (labeled) Petri net, and/or the supervisor S may be realized by a Petri net N (as opposed to being realized by an automaton R). An important motivation for using Petri nets is to extend the applicability of the results of supervisory control theory beyond the class of regular languages. Unfortunately, it turns out that several

difficulties arise in this endeavor. For this reason, no comprehensive set of results has been developed yet in this regard. We will highlight some of these difficulties in Sect. 4.5.1 below.

Of course, if $\mathcal{L}_m(G)$ and L_{am} are both regular languages, then it is not necessary to use Petri nets, even if these languages are given in terms of their Petri net representations. These Petri nets will necessarily be bounded and thus we can always build equivalent finite-state automata and then apply the algorithms of Chap. 3 to solve the supervisory control problems. However, it may still be useful to work with the Petri net representations for computational complexity reasons, since these Petri nets may be more compact than the corresponding finite-state automata (because of the system structure that they capture).

In the same vein, the structural information captured by Petri nets has motivated the development of several other approaches to the control of Petri nets. Unlike the language-based approach of supervisory control theory, these results are state-based; Sect. 4.5.2 concludes this chapter with a brief discussion one approach to state-based control of Petri nets.

4.5.1 Petri Nets and Supervisory Control Theory

It is instructive to begin this section with some comments on infinite-state systems and/or specifications.[3]

Infinite-State Systems or Specifications

It should be noted that if $\mathcal{L}_m(G)$ is not regular (say, G is an unbounded Petri net) but L_{am} is *regular and controllable* (with respect to $\mathcal{L}(G)$ and E_{uc}), then the non-regularity of $\mathcal{L}_m(G)$ is of no consequence since the desired supervisor S can be realized by a finite-state automaton (as was done in Chap. 3, Sect. 3.4.2). The "catch" here is the verification of the controllability properties of L_{am}, which in general requires manipulating G. In this regard, it should be observed that if L_{am} is controllable with respect to a *superset* of $\mathcal{L}(G)$, then L_{am} will necessarily be controllable with respect to $\mathcal{L}(G)$. Thus if one conjectures that L_{am} is controllable, then one can attempt to formally verify that conjecture by proving the controllability of L_{am} with respect to a *regular* superlanguage of $\mathcal{L}(G)$.

On the other hand, if $\mathcal{L}_m(G)$ is regular but L_{am} is *controllable but not regular*, then S cannot be realized by a finite-state automaton. (The regularity of $\mathcal{L}_m(G)$ is of no help.) But one may be able to realize S by a Petri net. Consider the following example.

Example 4.14 (Petri net realization of supervisor)
Consider the system G where $\mathcal{L}(G) = a^*b^*$ together with the prefix-closed admissible language L_a of the form

$$L_a = \overline{\{a^n b^m : n \geq m \geq 0\}}$$

Note that while $\mathcal{L}(G)$ is regular, L_a is not regular. The set of uncontrollable events is $E_{uc} = \{a\}$.

Since event b is controllable, it is clear that L_a is controllable. Therefore, there exists S such that $\mathcal{L}(S/G) = L_a$, but since L_a is not regular, S cannot be realized by a finite-state automaton. So instead, we wish to realize S with a Petri net. We claim

[3] This section assumes that the reader is familiar with the material presented in Chap. 3, principally Sects. 3.2, 3.4, and 3.5.

that the labeled Petri net N in Fig. 4.17, shown in its initial state $\mathbf{x}_0$, can be used for this purpose.

Let f denote the state transition function of N. Then S can be realized as follows; for $s \in \mathcal{L}(G)$ and $\mathbf{x} = f(\mathbf{x}_0, s)$, define

$$S(s) = \begin{cases} \{a, b\} & \text{if } x(p_3) > 0 \\ \{a\} & \text{if } x(p_3) = 0 \end{cases}$$

Consequently, we do have a realization of S that requires finite memory, namely storing N and the above rule for $S(s)$. It suffices to update the state of N in response to events executed by G (by firing the currently enabled transition with same event label), and read the contents of place p_3 to determine if event b should be enabled or not. In other words, event b is enabled by S whenever a transition labeled by b is enabled in N. Note that $\mathcal{L}(N) = L_a$. This is consistent with the standard realization of supervisors presented in Sect. 3.4.2 in Chap. 3, except that a Petri net is used instead of a finite-state automaton. Note that while two transitions of N share event label b, only one of these two is enabled at any given time; consequently, there is no ambiguity regarding which transition to fire in N if G executes event b.

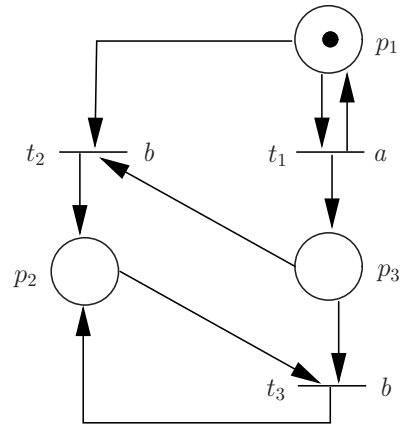

Figure 4.17: Petri net N of Example 4.14.

Some Difficulties that Arise

We claim that the extension of the applicability of the supervisor synthesis results of supervisory control theory to non-regular languages by the use of Petri net representations of languages leads to several difficulties that are not easily circumvented; we substantiate this claim by highlighting some of these difficulties.[4]

First, as was seen in Chap. 3, many supervisor synthesis procedures require performing the trim of a (blocking) automaton. The trim operation will most certainly be invoked in any supervisory control problem where blocking is an issue and where it is required that

[4]This section is included for readers interested in the connection between the results of Chap. 3 and Petri nets. Its material is more specialized than that in the rest of this chapter and it may be skipped without loss of continuity.

the controlled system be nonblocking. We saw in Chap. 2 that the trim operation is straightforward to implement on automata. Suppose that we have a Petri net N such that $\mathcal{L}(N) \supset \overline{\mathcal{L}_m(N)}$ and we wish to "transform" N into N' such that $\mathcal{L}_m(N') = \mathcal{L}_m(N)$ and $\mathcal{L}(N') = \overline{\mathcal{L}_m(N')}$. One notices immediately that trimming a blocking Petri net is by no means straightforward as the state of the Petri net is a "distributed object" (number of tokens in places) and thus we cannot simply "delete" blocking states. In fact, the structure of N, namely the set of places and transitions, may have to be modified in order to obtain the desired N'.

It turns out that a fundamental difficulty may arise in trimming blocking Petri nets. Consider the Petri net graph in Fig. 4.18, and take $\mathbf{x}_0 = [1, 0, 0, 0]$ and $\mathbf{X}_m = \{[0, 0, 0, 1]\}$; call the resulting net N. It can be verified that the languages generated and marked by N are:

$$\begin{aligned} \mathcal{L}(N) &= \overline{\{a^m b a^n b : m \geq n \geq 0\}} \\ \mathcal{L}_m(N) &= \{a^m b a^m b : m \geq 0\} \end{aligned}$$

N is blocking since $\mathcal{L}(N) \supset \mathcal{L}_m(N)$; for example, N deadlocks after string *aabab* since the resulting state, $[0, 1, 0, 1]$, is a deadlock state and it is not in $\mathbf{X}_m$.

In order to trim N, we need to construct N' such that

$$\mathcal{L}(N') = \overline{\mathcal{L}_m(N')} = \overline{\mathcal{L}_m(N)} \quad \text{and} \quad \mathcal{L}_m(N') = \mathcal{L}_m(N)$$

The problem that arises here is that $\overline{\mathcal{L}_m(N)} \notin \mathcal{PNL}$, that is, it is not a Petri net language! Thus the desired N' *does not exist*. A formal proof of this result is beyond the scope of this book. Intuitively, in order to trim N, we essentially require that place p_2 be empty before transition t_4 can be allowed to fire. We also note that place p_2 is unbounded. It can be shown that it is not possible to transform N in a manner that enforces the condition "p_2 empty before t_4 fires" without using an infinite number of places, an outcome that defeats the purpose of using a Petri net model![5]

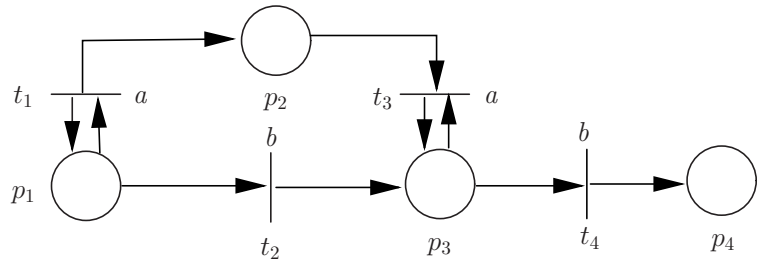

Figure 4.18: Petri net graph that is blocking with $\mathbf{x}_0 = [1, 0, 0, 0]$ and $\mathbf{X}_m = \{[0, 0, 0, 1]\}$.

The second difficulty that we wish to mention concerns the use of Petri nets as realizations of supervisors. Given the semantics of the feedback loop of supervisory control, it is necessary that the Petri net realization of a supervisor be "deterministic" in the sense that there is never any ambiguity about which transition to fire in the Petri net realization in response to the execution of an event by the system. In Chap. 3, we realized supervisors as deterministic automata and consequently this issue did not arise. But the transition

[5] An alternative is to use an *inhibitor arc* – see the references listed at the end of the chapter – but this is undesirable as Petri nets with inhibitor arcs are not tractable analytically.

labeling function of Petri nets need not be injective (cf. Sect. 4.2.4) and therefore this issue needs to be addressed here. In the Petri net in Fig. 4.17 considered in Example 4.14 for instance, two transitions have the event label b. If both of these transitions were enabled and the system executed event b, then which transition should the Petri net supervisor fire? This does not happen in the controlled system considered in Example 4.14, but it may occur in other examples.

The above discussion leads us to consider the subclass of Petri nets that possess the "deterministic property" of never having two distinct transitions with the same event label simultaneously enabled in any of their reachable states. Let us call Petri nets that possess this property *deterministic Petri nets* and let us denote by $\mathcal{PNL}^D$ the class of languages that can be represented by deterministic Petri nets. Clearly, $\mathcal{PNL}^D \subset \mathcal{PNL}$. In view of the difficulty about trimming first mentioned in this section and of the technique of standard realization presented in Chap. 3, an appropriate subclass of $\mathcal{PNL}$ appears to be

$$\mathcal{PNL}^{DP} := \{K \in \mathcal{PNL} : \overline{K} \in \mathcal{PNL}^D\}$$

That is, we are interested in those languages K whose prefix-closure can be generated by a deterministic Petri net; note that we say nothing about K being a deterministic Petri net language or not. If a given language $K \in \mathcal{PNL}^{DP}$ is controllable and $\mathcal{L}_m(G)$-closed, then the results of Chap. 3 and the above definition of the class $\mathcal{PNL}^{DP}$ imply that there exists a nonblocking supervisor S such that $\mathcal{L}_m(S/N) = K$ and moreover S *can be realized by a deterministic Petri net*, which was our original objective.

A key difficulty that arises in using the above result for solving supervisory control problems such as BSCP and BSCP-NB for admissible languages that are not regular is: How do we know if the supremal controllable sublanguage of the admissible language, namely $L_a^{\uparrow C}$ for BSCP and $L_{am}^{\uparrow C}$ for BSCP-NB, will be a member of the class $\mathcal{PNL}^{DP}$? To complicate matters further, it has been shown that the class $\mathcal{PNL}^{DP}$ is not closed under the $\uparrow C$ operation, that is, $K^{\uparrow C}$ need not be in $\mathcal{PNL}^{DP}$ even if $K \in \mathcal{PNL}^{DP}$.

We conclude from the discussion in this section that while Petri nets are a more powerful discrete event modeling formalism than finite-state automata, they do not appear to be a good candidate formalism for a general extension of the *algorithmic* component of supervisory control theory to *non-regular* languages.

4.5.2 State-Based Control of Petri Nets

The negative flavor of the results discussed in the preceding section by no means implies that there is no control theory for DES modeled by Petri nets. In fact, there is a large body of results on the control of Petri nets when the control paradigm is changed from the "language viewpoint" of supervisory control theory to a "state viewpoint" for a given (uncontrolled) system modeled by a Petri net. By this we mean that the specifications on the uncontrolled system are stated in terms of *forbidden states* and the control mechanism is to decide which of the controllable and enabled transitions in the system should be allowed to fire by the controller. In the same spirit as the notion of supremal sublanguages in supervisory control, the controller should guarantee that none of the forbidden states is reached, and it should do so with "minimal intervention" on the behavior of the system. The synthesis of such a controller should exploit the system structure captured by the Petri net model.

Coverage of this important theory is beyond the scope of this book; we refer the reader to the references at the end of this chapter. We shall limit ourselves to a brief discussion of one approach that exploits the structure of the Petri net. We start with a simple example

that shows how undesirable behavior can be avoided in a given Petri net by the judicious addition of places and arcs, resulting in a "controlled Petri net" with satisfactory behavior.

Example 4.15

Consider the Petri net N shown in Fig. 4.19; assume that N models the uncontrolled system behavior. Suppose that the specification on the states is given in terms of the linear inequality

$$x(p_2) + x(p_3) \leq 2 \tag{4.7}$$

where a state is deemed admissible if and only if it satisfies this inequality. Next, change the structure of N to obtain N_{aug} shown in Fig. 4.19. There is one additional place p_c that is connected to transitions t_4, t_1, and t_5 as shown in the figure. The initial state of p_c is set to two tokens. Place p_c can be thought of as a *control place* realizing a given control policy for N.[6] Given the structure of N_{aug}, it is implicitly assumed in this control policy that transitions t_1 and t_5 are controllable. Thus, if $x(p_c) > 0$, t_1 and t_5 are enabled, while they are disabled if $x(p_c) = 0$. The state of this simple controller is updated upon the firing of either t_4, t_1, or t_5.

The initial number of tokens in new place p_c was determined by "augmenting" the linear inequality (4.7) to a linear equality with p_c as the new variable:

$$x(p_2) + x(p_3) + x(p_c) = 2$$

leading to $x_0(p_c) = 2$ for the initial state of N shown in Fig. 4.19. Careful examination of the behavior of N_{aug} shows that the linear inequality (4.7) holds for all reachable states from its initial state. Essentially, place p_c ensures that at most two tokens get placed in the subnet consisting of places p_2, p_3 and transitions t_2, t_3.

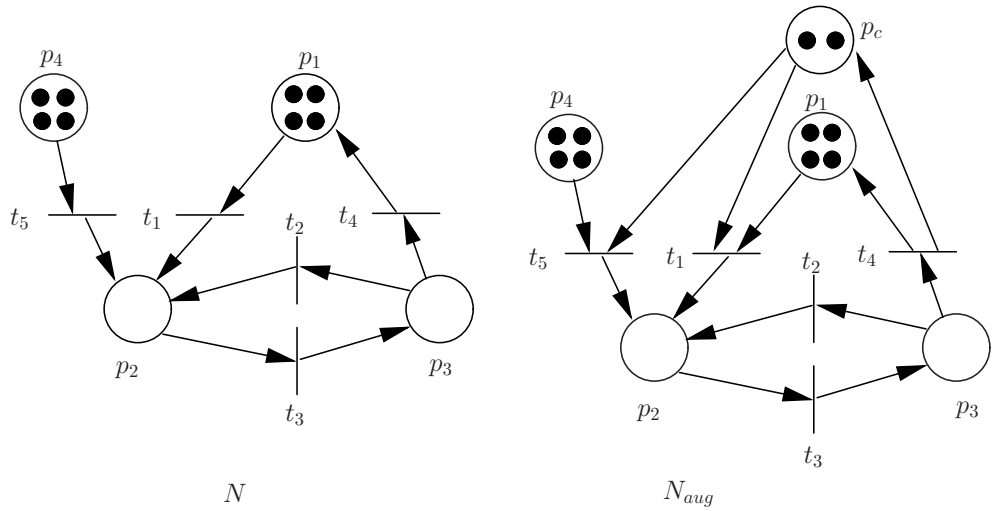

Figure 4.19: Petri net N for uncontrolled system and augmented Petri net N_{aug} where the constraint $x(p_2) + x(p_3) \leq 2$ is enforced (Example 4.15).

The above example raises the question: How should new place p_c be connected to the transitions of N in order to guarantee that the augmented net will satisfy the give linear

[6] The term "monitor" is also used in the literature to describe control places.

inequality for all its reachable states? The underlying concept at the heart of the solution procedure is that of place-invariant, which was introduced in Sect. 4.4.4. Let $\mathbf{A}$ be the incidence matrix of the Petri net graph of N; recall that $\mathbf{A}$ is of dimension $m \times n$, where $m = |T|$ and $n = |P|$. Assume that the specification for the behavior of N under control is captured in terms of a linear inequality on the state $\mathbf{x}$ of the net that must hold for all reachable states:

$$\mathbf{x}\boldsymbol{\ell} \leq b \tag{4.8}$$

for all $\mathbf{x} \in R(N)$. The vector $\boldsymbol{\ell}$ is a (column) vector of dimension n; the constant b and the entries of $\boldsymbol{\ell}$ are integers. Linear inequalities of the form (4.8) are often called *generalized mutual exclusion conditions*.

The key step for answering the question of what are the input and output arcs to p_c is to relate the specification in (4.8) with the property of conservation and place-invariants. Recall from Sect. 4.4.4 that if $\boldsymbol{\ell}$ is a place-invariant, then

$$\begin{aligned} \mathbf{x}\boldsymbol{\ell} &= \mathbf{x}_0\boldsymbol{\ell} + \mathbf{v}\mathbf{A}\boldsymbol{\ell} \\ &= \mathbf{x}_0\boldsymbol{\ell} \end{aligned}$$

for all reachable states $\mathbf{x} \in R(N)$. As was done in Example 4.15, let us rewrite the linear inequality (4.8) as an equality by adding control place p_c:

$$\mathbf{x}\boldsymbol{\ell} + x(p_c) = b \tag{4.9}$$

The introduction of place p_c means that the new incidence matrix for the modified Petri net graph is of the form:

$$\mathbf{A_{new}} = [\mathbf{A} \quad A_c]$$

where A_c is a column of dimension m describing the connectivity of new place p_c to the transitions of N. If we can find a place-invariant for $\mathbf{A_{new}}$, denoted by $\boldsymbol{\ell}_{new}$, of the form $\boldsymbol{\ell}_{new}^T = [\boldsymbol{\ell}^T \quad 1]$, that is,

$$\mathbf{A_{new}}\boldsymbol{\ell}_{new} = [\mathbf{A} \quad A_c] \begin{bmatrix} \boldsymbol{\ell} \\ 1 \end{bmatrix} = 0$$

then we will get

$$\mathbf{x}_{new}\boldsymbol{\ell}_{new} = \text{constant} \tag{4.10}$$

$$\Rightarrow \mathbf{x}\boldsymbol{\ell} + x(p_c) = \text{constant} \tag{4.11}$$

for all $\mathbf{x} \in R(N)$ and for all choices of x_0 in N. The desired A_c is therefore given by

$$A_c = -\mathbf{A}\boldsymbol{\ell} \tag{4.12}$$

We can verify that the connectivity of place p_c in Fig. 4.19 of Example 4.15 is indeed the result of this calculation.

The initial marking in place p_c is chosen based on $\mathbf{x}_0$ to force the "constant" in (4.10) to be equal to b:

$$x_0(p_c) = b - \mathbf{x}_0\boldsymbol{\ell} \tag{4.13}$$

If this results in a negative entry for $x_0(p_c)$, then we conclude that no solution exists to the original problem.

With A_c and $x_0(p_c)$ determined by (4.12) and (4.13), respectively, we conclude that

$$\mathbf{x}\boldsymbol{\ell} + x(p_c) = b$$

for all reachable states $\mathbf{x} \in R(N)$, which satisfies the original specification in (4.8).

Using multiple control places, this method can be extended to a *set* of linear inequalities of the form in (4.8). It can also be adapted to handle uncontrollable transitions and unobservable transitions.[7]

SUMMARY

- Petri nets are another modeling formalism for DES. A Petri net is defined by a set of places P, a set of transitions T, a set of arcs A, and a weight function w applied to arcs. Transitions correspond to events, and the places input to a transition are associated with conditions required for an event to occur.

- The dynamic behavior of a Petri net is described by using tokens in places to enable transitions, which then cause tokens to move around. The state of a Petri net is defined by a vector $[x(p_1), x(p_2), \ldots, x(p_n)]$ where $x(p_i)$ is the number of tokens present in place p_i. When transition t_j fires, the state components (markings) of its input and output places change according to $x'(p_i) = x(p_i) - w(p_i, t_j) + w(t_j, p_i)$.

- It is always possible to obtain a Petri net from a finite-state automaton, but the converse need not be true if the set of reachable states of the Petri net is infinite, which occurs when the number of tokens in a place can become arbitrarily large. This means that Petri nets can represent a larger class of DES than finite-state automata, or in other words the class of languages that can be marked by Petri nets is strictly larger than the class $\mathcal{R}$. The drawback is that analyzing Petri nets with unbounded places is much more challenging; in fact, many questions are undecidable.

- The coverability tree can be used to solve some, but not all, of the basic problems related to the behavior of Petri nets, such as boundedness, coverability, and conservation. For Petri nets with finite state spaces, the coverability tree becomes the reachability tree and is essentially a finite-state automaton that is equivalent to the original Petri net.

- Linear-algebraic techniques can also be used to analyze the behavior of Petri nets. The concept of place-invariant is useful in that regard.

- The extension of the scope of supervisory control theory to non-regular languages by the use of Petri net models leads to several technical difficulties. Control paradigms that exploit directly the system structure captured in the Petri net model have proven more effective.

[7] See the books by Moody & Antsaklis and Iordache & Antsaklis for a detailed treatment of this theory.

PROBLEMS

4.1 Consider the Petri net defined by:

$$P = \{p_1, p_2, p_3\} \qquad T = \{t_1, t_2, t_3\}$$
$$A = \{(p_1, t_1), (p_1, t_3), (p_2, t_1), (p_2, t_2), (p_3, t_3), (t_1, p_2), (t_1, p_3), (t_2, p_3), (t_3, p_1), (t_3, p_2)\}$$

with all arc weights equal to 1 except for $w(p_1, t_1) = 2$.

(a) Draw the corresponding Petri net graph.

(b) Let $\mathbf{x}_0 = [1, 0, 1]$ be the initial state. Show that in any subsequent operation of the Petri net, transition t_1 can never be enabled.

(c) Let $\mathbf{x}_0 = [2, 1, 1]$ be another initial state. Show that in any subsequent operation of the Petri net, either a deadlock occurs (no transition can be enabled) or a return to $\mathbf{x}_0$ results.

4.2 Consider the queueing system with state transition diagram in part (b) of Problem 2.15, Fig. 2.39 (b). It corresponds to a closed system with two queues and total capacity of five customers.

(a) Draw the corresponding Petri net graph.

(b) Show one possible Petri net state that corresponds to one of the six states.

(c) Are there any deadlock states?

4.3 For the system described in Problem 2.39, draw a Petri net graph and make sure to describe what each place and transition represents physically, keeping in mind the operating rules specified. For simplicity, assume that machine M_1 never breaks down. (There may be several Petri net models you can come up with, but the simplest one should consist of eight places.) Construct the coverability tree for the Petri net model you obtain.

4.4 Construct a Petri net model of the DES described in Problem 2.14. Show a state where a timeout may occur while task 1 is being processed.

4.5 Consider the Petri net shown in Fig. 4.20, with initial state $\mathbf{x}_0 = [1, 1, 0, 2]$.

(a) After the Petri net fires twice, find a state where all transitions are dead.

(b) Suppose we want to apply the firing sequence $(t_3, t_1, t_3, t_1, ...)$. Show that this is not possible for all future times.

(c) Find the state $\mathbf{x}_s$ resulting from the firing sequence $(t_1, t_2, t_3, t_3, t_3)$.

4.6 Consider the Petri net shown in Fig. 4.21, with initial state $\mathbf{x}_0 = [1, 1, 0, 0]$. Construct its coverability tree and use it to show that the empty state $[0, 0, 0, 0]$ is not reachable. Is this Petri net bounded?

4.7 Petri nets are often very useful in modeling the flow of data as they are processed for computational purposes. Transitions may represent operations (e.g., addition, multiplication), and places are used to represent the storage of data (e.g., placing a

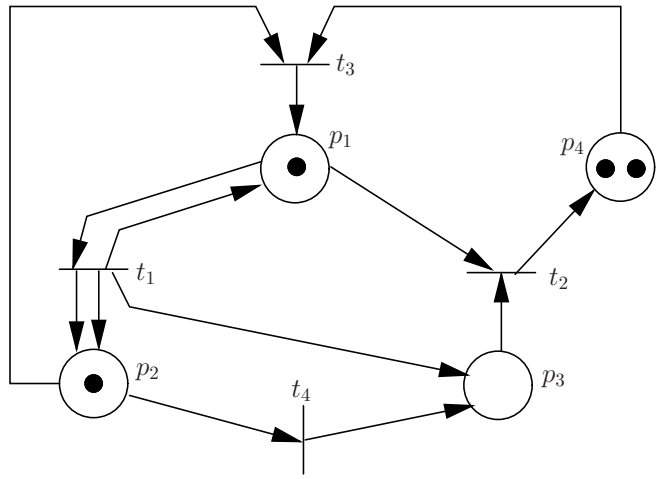

Figure 4.20: Problem 4.5.

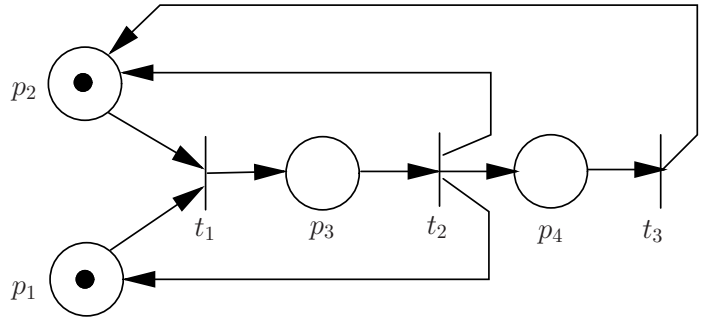

Figure 4.21: Problem 4.6.

token in a place called $X+Y$ represents the result of an operation where two numbers X and Y are added). Construct a Petri net graph to model the following computation:

$$(X+Y)(X-Z)+\frac{XY}{X-Y}$$

the result of which is denoted by R. (Make sure your model includes checks to prevent illegal operations from taking place.)

4.8 A Petri net in which there are no closed loops (directed circuits) formed is called *acyclic*. For this class of Petri nets, a necessary and sufficient condition for state $\mathbf{x}$ to be reachable from initial state $\mathbf{x}_0$ is that there exists a non-negative integer solution $\mathbf{z}$ to the equation:

$$\mathbf{x} = \mathbf{x}_0 + \mathbf{z}\mathbf{A}$$

where $\mathbf{A}$ is the incidence matrix (defined in Sect. 4.2.3).

(a) Use this result to show that $\mathbf{x} = [0, 0, 1, 2]$ is reachable from $\mathbf{x}_0 = [3, 1, 0, 0]$ for the Petri net:

$$P = \{p_1, p_2, p_3, p_4\} \qquad T = \{t_1, t_2\}$$
$$A = \{(p_1, t_1), (p_2, t_2), (p_3, t_2), (t_1, p_2), (t_1, p_3), (t_2, p_4)\}$$

with all arc weights equal to 1 except for $w(p_2, t_2) = 2$.

(b) What is the interpretation of the vector $\mathbf{z}$?

(c) Prove the sufficiency of the result above (i.e., the existence of a non-negative integer solution $\mathbf{z}$ implies reachability).

4.9 Construct a labeled Petri net N such that

$$\mathcal{L}_m(N) = \{a^n b^n c^n, n \geq 0\}$$

4.10 Consider the Petri net graph in Fig. 4.22.

(a) Determine if this Petri net is conservative for any choice of $\mathbf{x}_0$.

(b) Find an initial state such that this Petri net is live.

(c) Use the state equation of the Petri net to verify that state $\mathbf{x} = [0, 1, 1, 1, 1]$ is reachable from initial state $\mathbf{x}_0 = [2, 0, 0, 0, 0]$.

4.11 Prove or disprove:

(a) The class of Petri net languages $\mathcal{PNL}^{DP}$ is strictly larger than the class of regular languages $\mathcal{R}$.

(b) The class of marked graph languages (i.e., languages recognized by marked graphs) is incomparable with $\mathcal{R}$.

4.12 Consider the Petri net N in Fig. 4.23 where arc weight $r > 1$. Take $\mathbf{X}_m = \{[0, 0, 1]\}$. Verify that N is blocking. Find Petri net N' such that $N' = Trim(N)$. (*Hint:* Add $r-1$ transitions between places p_1 and p_2.)

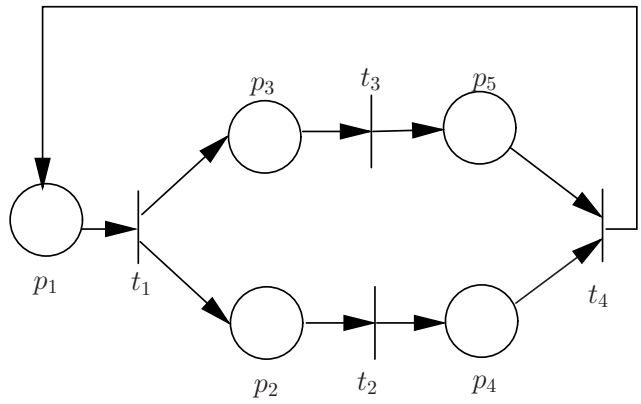

Figure 4.22: Problem 4.10.

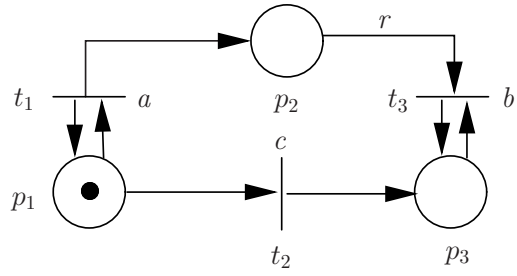

Figure 4.23: Problem 4.12.

4.13 Consider the DES G where $\mathcal{L}(G) = \overline{a^*ba^*}$ and $\mathcal{L}_m(G) = a^*ba^*$. Let $E_{uc} = \{b\}$. Take the admissible language to be

$$L_{am} = \{a^m ba^n : m \geq n \geq 0\}$$

(a) Is $\overline{L_{am}}$ a regular language?

(b) Verify that there exists a nonblocking supervisor S such that

$$\mathcal{L}_m(S/G) = L_{am}$$

(c) Build a Petri net realization of S.

4.14 In many DES modeled by Petri nets, places often have a maximum capacity in terms of the number of tokens that they can hold (e.g., the place may model a finite buffer). Imposing a maximum capacity on a place is a change in the Petri net dynamics described in Sect. 4.2.3, since an enabled transition cannot fire when its firing would lead to a violation of the capacity of one of its output places. Show how a finite-capacity place can be modeled without changing the operational rules of a Petri net (i.e., as described in Sect. 4.2.3). (*Hint:* Add a "complementary place" to the finite-capacity place and connect this new place appropriately with the existing transitions.)

4.15 Consider the Petri net N depicted in Fig. 4.24. Modify this Petri net in order to get a "controlled net" N_{aug} that satisfies the constraint $x(p_2) \leq x(p_1)$ for all reachable states from the initial state.

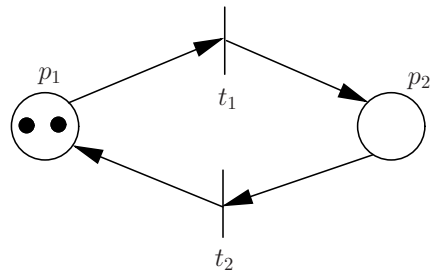

Figure 4.24: Problem 4.15.

4.16 Consider the Petri net in Fig. 4.25. Note that all arcs have weight 1 except the arcs from p_3 to t_2 and from t_4 to p_3, which have weight k. Take $\mathbf{x}_0 = [k\ 0\ k\ 0\ ]^T$.

(a) Is this Petri net bounded?

(b) Is this Petri net live?

(c) Is this Petri net conservative?

4.17 Consider the Petri net in Fig. 4.25, with the initial state $\mathbf{x}_0 = [k\ 0\ k\ 0\ ]^T$. Show how to build a controlled version of this net where the following constraint is satisfied in all reachable states (or explain why this is not possible):

$$x(p_3) \geq 1$$

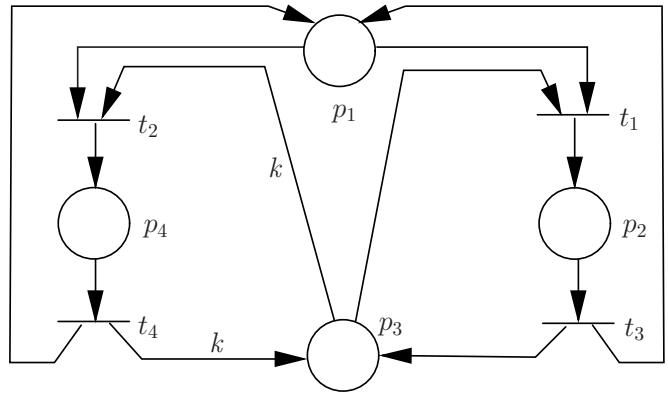

Figure 4.25: Problem 4.16.

SELECTED REFERENCES

- *Introductory books or survey papers on Petri nets*
  - David, R., and H. Alla, *Petri Nets & Grafcet: Tools for Modelling Discrete Event Systems*, Prentice-Hall, New York, 1992.
  - Murata, T., "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, Vol. 77, No. 4, pp. 541–580, 1989.
  - Peterson, J.L., *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, 1981.
- *Control of Petri nets*
  - Holloway, L.E., B.H. Krogh, and A. Giua, "A Survey of Petri Net Methods for Controlled Discrete Event Systems," *Journal of Discrete Event Dynamic Systems: Theory and Applications*, Vol. 7, No. 2, pp. 151–190, April 1997.
  - Iordache, M.V., and P.J. Antsaklis, *Supervisory Control of Concurrent Systems - A Petri Net Structural Approach*, Birkhäuser, Boston, 2006.
  - Moody, J.O., and P.J. Antsaklis, *Supervisory Control of Discrete Event Systems Using Petri Nets*, Kluwer Academic Publishers, Boston, 1998.
  - Stremersch, G., *Supervision of Petri Nets*, Kluwer Academic Publishers, Boston, 2001.
- *Miscellaneous*
  - Desrochers, A.A., and R.Y. Al-Jaar, *Applications of Petri Nets in Automated Manufacturing Systems: Modeling, Control, and Performance Analysis*, IEEE Press, Piscataway, NJ, 1995.
  - Giua, A., and F. DiCesare, "Blocking and Controllability of Petri Nets in Supervisory Control," *IEEE Transactions on Automatic Control*, Vol. 39, No. 4, pp. 818–823, April 1994.

– Sreenivas, R.S., and B.H. Krogh, "On Petri Net Models of Infinite State Supervisors," *IEEE Transactions on Automatic Control*, Vol. 37, No. 2, pp. 274–277, February 1992.

– Zhou, M.C., and F. DiCesare, *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*, Kluwer Academic Publishers, Boston, 1993.

■ *Notes*

1. The above survey paper by Murata contains a wealth of results and an extensive bibliography on Petri nets and their analysis (in particular, the linear-algebraic approach).
2. Readers interested in more details on Petri net languages and decidability issues (cf. discussion in Sect. 4.3) may consult the above book by Peterson.
3. The above survey paper by Holloway, Krogh, and Giua contains an insightful discussion and a detailed bibliography on state-based and language-based control methods for Petri nets.
4. The discussion and examples in Sect. 4.5.1 are primarily based on the work of Giua & DiCesare and Sreenivas & Krogh listed above.
5. The above books by Iordache & Antsaklis and Moody & Antsaklis cover in detail the state-based approach for control of Petri nets presented in Sect. 4.5.2 and have extensive lists of references.
6. The web site at url `http://www.informatik.uni-hamburg.de/TGI/PetriNets/` ("Petri Nets World") is a very useful source of information on Petri nets; in particular, this site contains bibliographical data on Petri nets and information about software tools for analyzing and simulating Petri net models.