Chapter 2
Languages and Automata

2.1 INTRODUCTION

We have seen how discrete-event systems (DES) differ from continuous-variable dynamic systems (CVDS) and why DES are not adequately modeled through differential or difference equations. Our first task, therefore, in studying DES is to develop appropriate models, which both adequately describe the behavior of these systems and provide a framework for analytical techniques to meet the goals of design, control, and performance evaluation.

When considering the state evolution of a DES, our first concern is with the sequence of states visited and the associated events causing these state transitions. To begin with, we will not concern ourselves with the issue of when the system enters a particular state or how long the system remains at that state. We will assume that the behavior of the DES is described in terms of event sequences of the form $e_1 e_2 \cdots e_n$. A sequence of that form specifies the order in which various events occur over time, but it does not provide the time instants associated with the occurrence of these events. This is the untimed or logical level of abstraction discussed in Sect. 1.3.3 in Chap. 1, where the behavior of the system is modeled by a *language*. Consequently, our first objective in this chapter is to discuss language models of DES and present operations on languages that will be used extensively in this and the next chapters.

As was mentioned in Sect. 1.3.3, the issue of representing languages using appropriate modeling formalisms is key for performing analysis and control of DES. The second objective of this chapter is to introduce and describe the first of the two untimed modeling formalisms for DES considered in this book to represent languages, *automata*. Automata form the most basic class of DES models. As we shall see in this chapter, they are intuitive, easy to use, amenable to composition operations, and amenable to analysis as well (in the finite-state case). On the other hand, they lack structure and for this reason may lead to very large state spaces when modeling complex systems. Nevertheless, any study of discrete event system

and control theory must start with a study of automata. The second modeling formalism considered in this book, Petri nets, will be presented in Chap. 4. As we shall see in that chapter, Petri nets have more structure than automata, although they do not possess, in general, the same analytical power as automata. Other modeling formalisms have been developed for untimed DES, most notably process algebras and logic-based models. These formalisms are beyond the scope of this book; some relevant references are presented at the end of this chapter.

The third objective of this chapter is to present some of the fundamental logical behavior problems we encounter in our study of DES. We would like to have systematic means for fully testing the logical behavior of a system and guaranteeing that it always does what it is supposed to. Using the automaton formalism, we will present solution techniques for three kinds of verification problems, those of safety (i.e., avoidance of illegal behavior), liveness (i.e., avoidance of deadlock and livelock), and diagnosis (i.e., ability to detect occurrences of unobservable events). These are the most common verification problems that arise in the study of software implementations of control systems for complex automated systems. The following chapter will address the problem of *controlling* the behavior of a DES, in the sense of the feedback control loop presented in Sect. 1.2.8, in order to ensure that the logical behavior of the closed-loop system is satisfactory.

Finally, we emphasize that an important objective of this book is to study timed and stochastic models of DES; establishing untimed models constitutes the first stepping stone towards this goal.

2.2 THE CONCEPTS OF LANGUAGES AND AUTOMATA

2.2.1 Language Models of Discrete Event Systems

One of the formal ways to study the logical behavior of DES is based on the theories of languages and automata. The starting point is the fact that any DES has an underlying event set E associated with it. The set E is thought of as the "alphabet" of a language and event sequences are thought of as "words" in that language. In this framework, we can pose questions such as "Can we build a system that speaks a given language?" or "What language does this system speak?"

To motivate our discussion of languages, let us consider a simple example. Suppose there is a machine we usually turn on once or twice a day (like a car, a photocopier, or a desktop computer), and we would like to design a simple system to perform the following basic task: When the machine is turned on, it should first issue a signal to tell us that it is in fact ON, then give us some simple status report (like, in the case of a car, "everything OK", "check oil", or "I need gas"), and conclude with another signal to inform us that "status report done". Each of these signals defines an event, and all of the possible signals the machine can issue define an alphabet (event set). Thus, our system has the makings of a DES driven by these events. This DES is responsible for recognizing events and giving the proper interpretation to any particular sequence received. For instance, the event sequence: "I'm ON", "everything is OK", "status report done", successfully completes our task. On the other hand, the event sequence: "I'm ON", "status report done", without some sort of actual status report in between, should be interpreted as an abnormal condition requiring special attention. We can therefore think of the combinations of signals issued by the machine as words belonging to the particular language spoken by this machine. In this particular example, the language of interest should be one with three-event words only,

always beginning with "I'm ON" and ending with "status report done". When the DES we build sees such a word, it knows the task is done. When it sees any other word, it knows something is wrong. We will return to this type of system in Example 2.10 and see how we can build a simple DES to perform a "status check" task.

Language Notation and Definitions

We begin by viewing the event set E of a DES as an alphabet. We will assume that E is finite. A sequence of events taken out of this alphabet forms a "word" or "string" (short for "string of events"). We shall use the term "string" in this book; note that the term "trace" is also used in the literature. A string consisting of no events is called the empty string and is denoted by ε. (The symbol ε is not to be confused with the generic symbol e for an element of E.) The length of a string is the number of events contained in it, counting multiple occurrences of the same event. If s is a string, we will denote its length by $|s|$. By convention, the length of the empty string ε is taken to be zero.

Definition. (Language)
A *language* defined over an event set E is a set of finite-length strings formed from events in E. ♦

As an example, let $E = \{a, b, g\}$ be the set of events. We may then define the language

$$L_1 = \{\varepsilon, a, abb\} \tag{2.1}$$

consisting of three strings only; or the language

$$L_2 = \{\text{all possible strings of length 3 starting with event } a\} \tag{2.2}$$

which contains nine strings; or the language

$$L_3 = \{\text{all possible strings of finite length which start with event } a\} \tag{2.3}$$

which contains an infinite number of strings.

The key operation involved in building strings, and thus languages, from a set of events E is *concatenation*. The string abb in L_1 above is the concatenation of the string ab with the event (or string of length one) b; ab is itself the concatenation of a and b. The concatenation uv of two strings u and v is the new string consisting of the events in u immediately followed by the events in v. The empty string ε is the *identity element* of concatenation: $u\varepsilon = \varepsilon u = u$ for any string u.

Let us denote by E^* the set of *all* finite strings of elements of E, including the empty string ε; the * operation is called the *Kleene-closure*. Observe that the set E^* is countably infinite since it contains strings of arbitrarily long length. For example, if $E = \{a, b, c\}$, then

$$E^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \ldots\}$$

A language over an event set E is therefore a *subset* of E^*. In particular, $\emptyset$, E, and E^* are languages.

We conclude this discussion with some terminology about strings. If $tuv = s$ with $t, u, v \in E^*$, then:

- t is called a *prefix* of s,

- u is called a *substring* of s, and
- v is called a *suffix* of s.
  We will sometimes use the notation s/t (read "s after t") to denote the suffix of s after its prefix t. If t is not a prefix of s, then s/t is not defined.

Observe that both ε and s are prefixes (substrings, suffixes) of s.

Operations on Languages

The usual set operations, such as union, intersection, difference, and complement with respect to E^*, are applicable to languages since languages are sets. In addition, we will also use the following operations:[1]

- *Concatenation*: Let $L_a, L_b \subseteq E^*$, then

$$L_a L_b := \{s \in E^* : (s = s_a s_b) \text{ and } (s_a \in L_a) \text{ and } (s_b \in L_b)\}$$

  In words, a string is in $L_a L_b$ if it can be written as the concatenation of a string in L_a with a string in L_b.

- *Prefix-closure*: Let $L \subseteq E^*$, then

$$\overline{L} := \{s \in E^* : (\exists t \in E^*)\ [st \in L]\}$$

  In words, the prefix closure of L is the language denoted by $\overline{L}$ and consisting of all the prefixes of all the strings in L. In general, $L \subseteq \overline{L}$.

  L is said to be *prefix-closed* if $L = \overline{L}$. Thus language L is prefix-closed if any prefix of any string in L is also an element of L.

- *Kleene-closure*: Let $L \subseteq E^*$, then

$$L^* := \{\varepsilon\} \cup L \cup LL \cup LLL \cup \cdots$$

  This is the same operation that we defined above for the set E, except that now it is applied to set L whose elements may be strings of length greater than one. An element of L^* is formed by the concatenation of a finite (but possibly arbitrarily large) number of elements of L; this includes the concatenation of "zero" elements, that is, the empty string ε. Note that the * operation is idempotent: $(L^*)^* = L^*$.

- *Post-language*: Let $L \subseteq E^*$ and $s \in \overline{L}$. Then the post-language of L after s, denoted by L/s, is the language

$$L/s := \{t \in E^* : st \in L\}$$

  By definition, $L/s = \emptyset$ if $s \notin \overline{L}$.

Observe that in expressions involving several operations on languages, prefix-closure and Kleene-closure should be applied first, and concatenation always precedes operations such as union, intersection, and set difference. (This was implicitly assumed in the above definition of L^*.)

[1] ":=" denotes "equal to by definition."

Example 2.1 (Operations on languages)
Let $E = \{a, b, g\}$, and consider the two languages $L_1 = \{\varepsilon, a, abb\}$ and $L_4 = \{g\}$. Neither L_1 nor L_4 are prefix-closed, since $ab \notin L_1$ and $\varepsilon \notin L_4$. Then:

$$\begin{aligned} L_1 L_4 &= \{g, ag, abbg\} \\ \overline{L_1} &= \{\varepsilon, a, ab, abb\} \\ \overline{L_4} &= \{\varepsilon, g\} \\ L_1 \overline{L_4} &= \{\varepsilon, a, abb, g, ag, abbg\} \\ L_4^* &= \{\varepsilon, g, gg, ggg, \ldots\} \\ L_1^* &= \{\varepsilon, a, abb, aa, aabb, abba, abbabb, \ldots\} \end{aligned}$$

We make the following observations for technical accuracy:

(i) $\varepsilon \notin \emptyset$;

(ii) $\{\varepsilon\}$ is a nonempty language containing only the empty string;

(iii) If $L = \emptyset$ then $\overline{L} = \emptyset$, and if $L \neq \emptyset$ then necessarily $\varepsilon \in \overline{L}$;

(iv) $\emptyset^* = \{\varepsilon\}$ and $\{\varepsilon\}^* = \{\varepsilon\}$;

(v) $\emptyset L = L\emptyset = \emptyset$.

Projections of Strings and Languages

Another type of operation frequently performed on strings and languages is the so-called *natural projection*, or simply *projection*, from a set of events, E_l, to a *smaller* set of events, E_s, where $E_s \subset E_l$. Natural projections are denoted by the letter P; a subscript is typically added to specify either E_s or both E_l and E_s for the sake of clarity when dealing with multiple sets. In the present discussion, we assume that the two sets E_l and E_s are fixed and we use the letter P without subscript.

We start by defining the projection P for strings:

$$P : E_l^* \to E_s^*$$

where

$$\begin{aligned} P(\varepsilon) &:= \varepsilon \\ P(e) &:= \begin{cases} e & \text{if } e \in E_s \\ \varepsilon & \text{if } e \in E_l \setminus E_s \end{cases} \\ P(se) &:= P(s)P(e) \text{ for } s \in E_l^*,\ e \in E_l \end{aligned}$$

As can be seen from the definition, the projection operation takes a string formed from the larger event set (E_l) and *erases* events in it that do not belong to the smaller event set (E_s).

We will also be working with the corresponding inverse map

$$P^{-1} : E_s^* \to 2^{E_l^*}$$

defined as follows

$$P^{-1}(t) := \{s \in E_l^* : P(s) = t\}$$

(Given a set A, the notation 2^A means the power set of A, that is, the set of all subsets of A.)
Given a string of events in the smaller event set (E_s), the inverse projection P^{-1} returns the set of all strings from the larger event set (E_l) that project, with P, to the given string.

The projection P and its inverse P^{-1} are extended to languages by simply applying them to all the strings in the language. For $L \subseteq E_l^*$,

$$P(L) := \{t \in E_s^* : (\exists s \in L)\ [P(s) = t]\}$$

and for $L_s \subseteq E_s^*$,

$$P^{-1}(L_s) := \{s \in E_l^* : (\exists t \in L_s)\ [P(s) = t]\}$$

Example 2.2 (Projection)

Let $E_l = \{a, b, c\}$ and consider the two proper subsets $E_1 = \{a, b\}$, $E_2 = \{b, c\}$. Take

$$L = \{c, ccb, abc, cacb, cabcbbca\} \subset E_l^*$$

Consider the two projections $P_i : E_l^* \to E_i^*$, $i = 1, 2$. We have that

$$\begin{aligned} P_1(L) &= \{\varepsilon, b, ab, abbba\} \\ P_2(L) &= \{c, ccb, bc, cbcbbc\} \\ P_1^{-1}(\{\varepsilon\}) &= \{c\}^* \\ P_1^{-1}(\{b\}) &= \{c\}^*\{b\}\{c\}^* \\ P_1^{-1}(\{ab\}) &= \{c\}^*\{a\}\{c\}^*\{b\}\{c\}^* \end{aligned}$$

We can see that

$$P_1^{-1}[P_1(\{abc\})] = P_1^{-1}[\{ab\}] \supset \{abc\}$$

Thus, in general, $P^{-1}[P(A)] \neq A$ for a given language $A \subseteq E_l^*$.

Natural projections play an important role in the study of DES. They will be used extensively in this and the next chapter. We state some useful properties of natural projections. Their proofs follow from the definitions of P and P^{-1} and from set theory.

Proposition. (Properties of natural projections)

1. $P[P^{-1}(L)] = L$
   $L \subseteq P^{-1}[P(L)]$
2. If $A \subseteq B$ then $P(A) \subseteq P(B)$ and $P^{-1}(A) \subseteq P^{-1}(B)$
3. $P(A \cup B) = P(A) \cup P(B)$
   $P(A \cap B) \subseteq P(A) \cap P(B)$
4. $P^{-1}(A \cup B) = P^{-1}(A) \cup P^{-1}(B)$
   $P^{-1}(A \cap B) = P^{-1}(A) \cap P^{-1}(B)$
5. $P(AB) = P(A)P(B)$
   $P^{-1}(AB) = P^{-1}(A)P^{-1}(B)$

Representation of Languages

A language may be thought of as a formal way of describing the behavior of a DES. It specifies all admissible sequences of events that the DES is capable of "processing" or "generating", while bypassing the need for any additional structure. Taking a closer look at the example languages L_1, L_2, and L_3 in equations (2.1)–(2.3) above, we can make the following observations. First, L_1 is easy to define by simple enumeration, since it consists of only three strings. Second, L_2 is defined descriptively, only because it is simpler to do so rather than writing down the nine strings it consists of; but we could also have easily enumerated these strings. Finally, in the case of L_3 we are limited to a descriptive definition, since full enumeration is not possible.

The difficulty here is that "simple" representations of languages are not always easy to specify or work with. In other words, we need a set of compact "structures" which define languages and which can be manipulated through well-defined operations so that we can construct, and subsequently manipulate and analyze, arbitrarily complex languages. In CVDS for instance, we can conveniently describe inputs we are interested in applying to a system by means of functional expressions of time such as $\sin(wt)$ or $(a + bt)^2$; the system itself is described by a differential or difference equation. Basic algebra and calculus provide the framework for manipulating such expressions and solving the problem of interest (for example, does the output trajectory meet certain requirements?). The next section will present the modeling formalism of automata as a framework for representing and manipulating languages and solving problems that pertain to the logical behavior of DES.

2.2.2 Automata

An automaton is a device that is capable of representing a language according to well-defined rules. This section focuses on the formal definition of automaton. The connection between languages and automata will be made in the next section. The simplest way to present the notion of automaton is to consider its directed graph representation, or *state transition diagram*. We use the following example for this purpose.

Example 2.3 (A simple automaton)
Let the event set be $E = \{a, b, g\}$. Consider the state transition diagram in Fig. 2.1, where nodes represent *states* and labeled arcs represent *transitions* between these states. This directed graph provides a description of the dynamics of an automaton. The set of nodes is the state set of the automation, $X = \{x, y, z\}$. The labels of the transitions are elements of the event set (alphabet) E of the automaton. The arcs in the graph provide a graphical representation of the *transition function* of the automaton, which we denote as $f : X \times E \to X$:

$$f(x, a) = x \qquad f(x, g) = z$$
$$f(y, a) = x \qquad f(y, b) = y$$
$$f(z, b) = z \qquad f(z, a) = f(z, g) = y$$

The notation $f(y, a) = x$ means that if the automaton is in state y, then upon the "occurrence" of event a, the automaton will make an instantaneous transition to state x. The cause of the occurrence of event a is irrelevant; the event could be an external input to the system modeled by the automaton, or it could be an event spontaneously "generated" by the system modeled by the automaton.

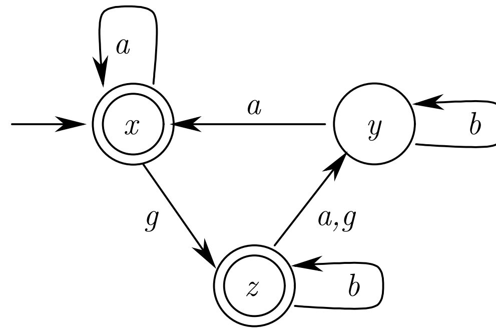

Figure 2.1: State transition diagram for Example 2.3.
The event set is $E = \{a, b, g\}$, and the state set is $X = \{x, y, z\}$. The initial state is x (marked by an arrow), and the set of marked states is $\{x, z\}$ (double circles).

Three observations are worth making regarding Example 2.3. First, an event may occur without changing the state, as in $f(x, a) = x$. Second, two distinct events may occur at a given state causing the exact same transition, as in $f(z, a) = f(z, g) = y$. What is interesting about the latter fact is that we may not be able to distinguish between events a and g by simply observing a transition from state z to state y. Third, the function f is a *partial function* on its domain $X \times E$, that is, there need not be a transition defined for each event in E at each state of X; for instance, $f(x, b)$ and $f(y, g)$ are not defined.

Two more ingredients are necessary to completely define an automaton: An initial state, denoted by x_0, and a subset X_m of X that represents the states of X that are *marked*. The role of the set X_m will become apparent in the remainder of this chapter as well as in Chap. 3. States are marked when it is desired to attach a special meaning to them. Marked states are also referred to as "accepting" states or "final" states. In the figures in this book, the initial state will be identified by an arrow pointing into it and states belonging to X_m will be identified by double circles.

We can now state the formal definition of an automaton. We begin with deterministic automata. Nondeterministic automata will be formally defined in Sect. 2.2.4.

Definition. (Deterministic automaton)
A *Deterministic Automaton*, denoted by G, is a six-tuple

$$G = (X, E, f, \Gamma, x_0, X_m)$$

where:

X is the set of *states*

E is the finite set of *events* associated with G

$f : X \times E \to X$ is the *transition function*: $f(x, e) = y$ means that there is a transition labeled by event e from state x to state y; in general, f is a *partial* function on its domain

$\Gamma : X \to 2^E$ is the *active event function* (or feasible event function); $\Gamma(x)$ is the set of all events e for which $f(x, e)$ is defined and it is called the *active event set* (or feasible event set) of G at x

x_0 is the *initial* state

$X_m \subseteq X$ is the set of *marked states*. ♦

We make the following remarks about this definition.

- The words *state machine* and *generator* (which explains the notation G) are also often used to describe the above object.
- If X is a finite set, we call G a *deterministic finite-state automaton*, often abbreviated as DFA in this book.
- The functions f and Γ are completely described by the state transition diagram of the automaton.
- The automaton is said to be *deterministic* because f is a function from $X \times E$ to X, namely, there cannot be two transitions with the same event label out of a state. In contrast, the transition structure of a *nondeterministic* automaton is defined by means of a function from $X \times E$ to 2^X; in this case, there can be multiple transitions with the same event label out of a state. Note that by default, the word automaton will refer to deterministic automaton in this book. We will return to nondeterministic automata in Sect. 2.2.4.
- The fact that we allow the transition function f to be partially defined over its domain $X \times E$ is a variation over the usual definition of automaton in the computer science literature that is quite important in DES theory.
- Formally speaking, the inclusion of Γ in the definition of G is superfluous in the sense that Γ is derived from f. For this reason, we will sometimes omit explicitly writing Γ when specifying an automaton when the active event function is not central to the discussion. One of the reasons why we care about the contents of $\Gamma(x)$ for state x is to help distinguish between events e that are feasible at x but cause no state transition, that is, $f(x, e) = x$, and events e' that are not feasible at x, that is, $f(x, e')$ is not defined.
- Proper selection of which states to mark is a modeling issue that depends on the problem of interest. By designating certain states as marked, we may for instance be recording that the system, upon entering these states, has completed some operation or task.
- The event set E includes all events that appear as transition labels in the state transition diagram of automaton G. In general, the set E might also include additional events, since it is a parameter in the definition of G. Such events do not play a role in defining the dynamics of G since f is not defined for them; however, as we will see later, they may play a role when G is composed with other automata using the parallel composition operation studied in Sect. 2.3.2. In this book, when the event set of an automaton is not explicitly defined, it will be assumed equal to set of events that appear in the state transition diagram of the automaton.

The automaton G operates as follows. It starts in the initial state x_0 and upon the occurrence of an event $e \in \Gamma(x_0) \subseteq E$ it will make a transition to state $f(x_0, e) \in X$. This process then continues based on the transitions for which f is defined.

For the sake of convenience, f is always extended from domain $X \times E$ to domain $X \times E^*$ in the following recursive manner:

$$\begin{aligned} f(x, \varepsilon) &:= x \\ f(x, se) &:= f(f(x, s), e) \quad \text{for } s \in E^* \text{ and } e \in E \end{aligned}$$

Note that the extended form of f subsumes the original f and both are consistent for single events $e \in E$. For this reason, the same notation f can be used for the extended function without any danger of confusion.

Returning to the automaton in Example 2.3, we have for example that

$$\begin{aligned} f(y, \varepsilon) &= y \\ f(x, gba) &= f(f(x, gb), a) = f(f(f(x, g), b), a) = f(f(z, b), a) = f(z, a) = y \\ f(x, aagb) &= z \\ f(z, b^n) &= z \text{ for all } n \geq 0 \end{aligned}$$

where b^n denotes n consecutive occurrences of event b. These results are easily seen by inspection of the state transition diagram in Fig. 2.1.

Remarks.

1. We emphasize that we do not wish to require at this point that the set X be finite. In particular, the concepts and operations introduced in the remainder of Sect. 2.2 and in Sect. 2.3 work for infinite-state automata. Of course, explicit representations of infinite-state automata would require infinite memory. Finite-state automata will be discussed in Sect. 2.4.

2. The automaton model defined in this section is also referred to as a *Generalized Semi-Markov Scheme* (abbreviated as GSMS) in the literature on stochastic processes. The term "semi-Markov" historically comes from the theory of Markov processes. We will cover this theory and use the GSMS terminology in later chapters in this book, in the context of our study of stochastic timed models of DES.

2.2.3 Languages Represented by Automata

The connection between languages and automata is easily made by inspecting the state transition diagram of an automaton. Consider all the directed paths that can be followed in the state transition diagram, starting at the initial state; consider among these all the paths that end in a marked state. This leads to the notions of the languages *generated* and *marked* by an automaton.

Definition. (Languages generated and marked)
The *language generated* by $G = (X, E, f, \Gamma, x_0, X_m)$ is

$$\mathcal{L}(G) := \{s \in E^* : f(x_0, s) \text{ is defined}\}$$

The *language marked* by G is

$$\mathcal{L}_m(G) := \{s \in \mathcal{L}(G) : f(x_0, s) \in X_m\} \qquad \blacklozenge$$

The above definitions assume that we are working with the extended transition function $f : X \times E^* \rightarrow X$. An immediate consequence is that for any G with non-empty X, $\varepsilon \in \mathcal{L}(G)$.

The language $\mathcal{L}(G)$ represents all the directed paths that can be followed along the state transition diagram, starting at the initial state; the string corresponding to a path is the

concatenation of the event labels of the transitions composing the path. Therefore, a string s is in $\mathcal{L}(G)$ if and only if it corresponds to an admissible path in the state transition diagram, equivalently, if and only if f is defined at (x_0, s). $\mathcal{L}(G)$ is prefix-closed by definition, since a path is only possible if all its prefixes are also possible. If f is a total function over its domain, then necessarily $\mathcal{L}(G) = E^*$. We will use the terminology *active event* to denote any event in E that appears in some string in $\mathcal{L}(G)$; recall that not all events in E need be active.

The second language represented by G, $\mathcal{L}_m(G)$, is the subset of $\mathcal{L}(G)$ consisting only of the strings s for which $f(x_0, s) \in X_m$, that is, these strings correspond to paths that end at a marked state in the state transition diagram. Since not all states of X need be marked, the language marked by G, $\mathcal{L}_m(G)$, need not be prefix-closed in general. The language marked is also called the language *recognized* by the automaton, and we often say that the given automaton is a *recognizer* of the given language.

When manipulating the state transition diagram of an automaton, it may happen that all states in X get deleted, resulting in what is termed the *empty automaton*. The empty automaton necessarily generates and marks the empty set.

Example 2.4 (Marked language)
Let $E = \{a, b\}$ be the event set. Consider the language

$$L = \{a, aa, ba, aaa, aba, baa, bba, \ldots\}$$

consisting of all strings of a's and b's always followed by an event a. This language is marked by the finite-state automaton $G = (E, X, f, \Gamma, x_0, X_m)$ where $X = \{0, 1\}$, $x_0 = 0$, $X_m = \{1\}$, and f is defined as follows: $f(0, a) = 1$, $f(0, b) = 0$, $f(1, a) = 1$, $f(1, b) = 0$.

This can be seen as follows. With 0 as the initial state, the only way that the marked state 1 can be reached is if event a occurs at some point. Then, either the state remains forever unaffected or it eventually becomes 0 again if event b takes place. In the latter case, we are back where we started, and the process simply repeats. The state transition diagram of this automaton is shown in Fig. 2.2. We can see from the figure that $\mathcal{L}_m(G) = L$. Note that in this example f is a total function and therefore the language generated by G is $\mathcal{L}(G) = E^*$.

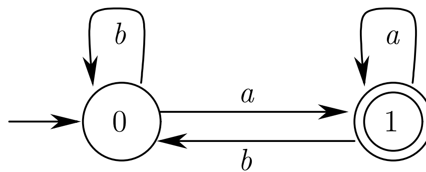

Figure 2.2: Automaton of Example 2.4.
This automaton marks the language $L = \{a, aa, ba, aaa, aba, baa, bba, \ldots\}$ consisting of all strings of a's and b's followed by a, given the event set $E = \{a, b\}$.

Example 2.5 (Generated and marked languages)
If we modify the automaton in Example 2.4 by removing the self-loop due to event b at state 0 in Fig. 2.2, that is, by letting $f(0, b)$ be undefined, then $\mathcal{L}(G)$ now consists of ε together with the strings in E^* that start with event a and where there are no consecutive occurrences of event b. Any b in the string is either the last event of the

string or it is immediately followed by an a. $\mathcal{L}_m(G)$ is the subset of $\mathcal{L}(G)$ consisting of those strings that end with event a.

Thus, an automaton G is a representation of *two* languages: $\mathcal{L}(G)$ and $\mathcal{L}_m(G)$. The state transition diagram of G contains all the information necessary to characterize these two languages. We note again that in the standard definition of automaton in automata theory, the function f is required to be a total function and the notion of language generated is not meaningful since it is always equal to E^*. In DES theory, allowing f to be partial is a consequence of the fact that a system may not be able to produce (or execute) all strings in E^*. Subsequent examples in this and the next chapters will illustrate this point.

Language Equivalence of Automata

It is clear that there are many ways to construct automata that generate, or mark, a given language. Two automata are said to be *language-equivalent* if they generate *and* mark the same languages. Formally:

Definition. (Language-equivalent automata)
Automata G_1 and G_2 are said to be *language-equivalent* if

$$\mathcal{L}(G_1) = \mathcal{L}(G_2) \qquad \text{and} \qquad \mathcal{L}_m(G_1) = \mathcal{L}_m(G_2)$$ ♦

Example 2.6 (Language-equivalent automata)
Let us return to the automaton described in Example 2.5. This automaton is shown in Fig. 2.3. The three automata shown in Fig. 2.3 are language-equivalent, as they all generate the same language and they all mark the same language. Observe that the third automaton in Fig. 2.3 has an infinite state set.

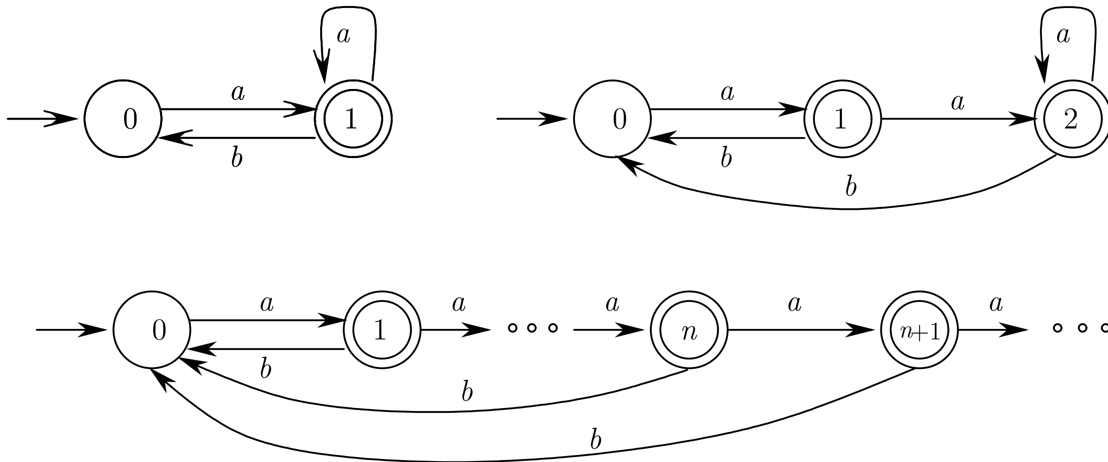

Figure 2.3: Three language-equivalent automata (Example 2.6.).

Blocking

In general, we have from the definitions of G, $\mathcal{L}(G)$, and $\mathcal{L}_m(G)$ that

$$\mathcal{L}_m(G) \subseteq \overline{\mathcal{L}_m(G)} \subseteq \mathcal{L}(G)$$

The first set inclusion is due to the fact that X_m may be a proper subset of X, while the second set inclusion is a consequence of the definition of $\mathcal{L}_m(G)$ and the fact that $\mathcal{L}(G)$ is prefix-closed by definition. It is worth examining this second set inclusion in more detail.

An automaton G could reach a state x where $\Gamma(x) = \emptyset$ but $x \notin X_m$. This is called a *deadlock* because no further event can be executed. Given our interpretation of marking, we say that the system "blocks" because it enters a deadlock state without having terminated the task at hand. If deadlock happens, then necessarily $\overline{\mathcal{L}_m(G)}$ will be a proper subset of $\mathcal{L}(G)$, since any string in $\mathcal{L}(G)$ that ends at state x cannot be a prefix of a string in $\mathcal{L}_m(G)$.

Another issue to consider is when there is a set of unmarked states in G that forms a strongly connected component (i.e., these states are reachable from one another), but with *no transition going out of the set*. If the system enters this set of states, then we get what is called a *livelock*. While the system is "live" in the sense that it can always execute an event, it can never complete the task started since no state in the set is marked and the system cannot leave this set of states. If livelock is possible, then again $\overline{\mathcal{L}_m(G)}$ will be a proper subset of $\mathcal{L}(G)$. Any string in $\mathcal{L}(G)$ that reaches the absorbing set of unmarked states cannot be a prefix of a string in $\mathcal{L}_m(G)$, since we assume that there is no way out of this set. Again, the system is "blocked" in the livelock.

The importance of deadlock and livelock in DES leads us to formulate the following definition.

Definition. (Blocking)
Automaton G is said to be *blocking* if

$$\overline{\mathcal{L}_m(G)} \subset \mathcal{L}(G)$$

where the set inclusion is proper,[2] and *nonblocking* when

$$\overline{\mathcal{L}_m(G)} = \mathcal{L}(G)$$ ♦

Thus, if an automaton is blocking, this means that deadlock and/or livelock can happen.

The notion of marked states and the definitions that we have given for language generated, language marked, and blocking, provide an approach for considering deadlock and livelock that is useful in a wide variety of applications.

Example 2.7 (Blocking)
Let us examine the automaton G depicted in Fig. 2.4. Clearly, state 5 is a deadlock state. Moreover, states 3 and 4, with their associated a, b, and g transitions, form an absorbing strongly connected component; since neither 3 nor 4 is marked, any string that reaches state 3 will lead to a livelock. String $ag \in \mathcal{L}(G)$ but $ag \notin \overline{\mathcal{L}_m(G)}$; the same is true for any string in $\mathcal{L}(G)$ that starts with aa. Thus G is blocking since $\overline{\mathcal{L}_m(G)}$ is a proper subset of $\mathcal{L}(G)$.

We return to examples of DES presented in Chap. 1 in order to further illustrate blocking and livelock.

Example 2.8 (Deadlock in database systems)
Recall our description of the concurrent execution of database transactions in Sect. 1.3.4. Let us assume that we want to build an automaton H_a whose language would

[2] We shall use the notation $\subset$ for "strictly contained in" and $\subseteq$ for "contained in or equal to."

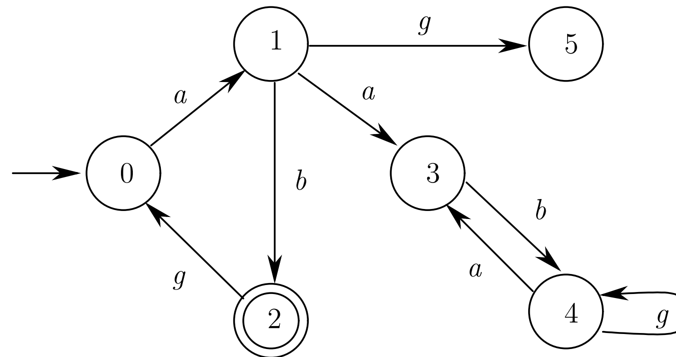

Figure 2.4: Blocking automaton of Example 2.7.
State 5 is a deadlock state, and states 3 and 4 are involved in a livelock.

exactly be the set of *admissible schedules* corresponding to the concurrent execution of transactions:

$$r_1(a)r_1(b) \quad \text{and} \quad r_2(a)w_2(a)r_2(b)w_2(b)$$

This H_a will have a single marked state, reached by all admissible schedules that contain all the events of transactions 1 and 2; that is, this marked state models the successful completion of the execution of both transactions, the desired goal in this problem. We will not completely build H_a here; this is done in Example 3.3 in Chap. 3. However, we have argued in Sect. 1.3.4 that the schedule:

$$S_z = r_1(a)r_2(a)w_2(a)r_2(b)w_2(b)$$

is admissible but its only possible continuation, with event $r_1(b)$, leads to an inadmissible schedule. This means that the state reached in H_a after string S_z has to be a deadlock state, and consequently H_a will be a blocking automaton.

Example 2.9 (Livelock in telephony)
In Sect. 1.3.4, we discussed the modeling of telephone networks for the purpose of studying possible conflicts between call processing features. Let us suppose that we are building an automaton model of the behavior of three users, each subscribing to "call forwarding", and where: (i) user 1 forwards all his calls to user 2, (ii) user 2 forwards all his calls to user 3, and (iii) user 3 forwards all his calls to user 1. The marked state of this automaton could be the initial state, meaning that all call requests can be properly completed and the system eventually returns to the initial state. The automaton model of this system, under the above instructions for forwarding, would be blocking because it would contain a livelock. This livelock would occur after any request for connection by a user, since the resulting behavior would be an endless sequence of forwarding events with no actual connection of the call ever happening. (In practice, the calling party would hang up the phone out of frustration, but we assume here that such an event is not included in the model!)

Other Examples

We conclude this section with two more examples; the first one revisits the machine status check example discussed at the beginning of Sect. 2.2.1 while the second one presents an automaton model of the familiar queueing system.

Example 2.10 (Machine status check)
Let $E = \{a_1, a_2, b\}$ be the set of events. A task is defined as a three-event sequence which begins with event b, followed by event a_1 or a_2, and then event b, followed by any arbitrary event sequence. Thus, we would like to design a device that reads any string formed by this event set, but only recognizes, that is marks, strings of length 3 or more that satisfy the above definition of a task. Each such string must begin with b and include a_1 or a_2 in the second position, followed by b in the third position. What follows after the third position is of no concern in this example.

A finite-state automaton that marks this language (and can therefore implement the desired task) is shown in Fig. 2.5. The state set $X = \{0, 1, 2, 3, 4, 5\}$ consists of arbitrarily labeled states with $x_0 = 0$ and $X_m = \{4\}$. Note that any string with less than 3 events always ends up in state 0, 1, 2, 3, or 5, and is therefore not marked. The only strings with 3 or more events that are marked are those that start with b, followed by either a_1 or a_2, and then reach state 4 through event b.

There is a simple interpretation for this automaton and the language it marks, which corresponds to the machine status check example introduced at the beginning of Sect. 2.2.1. When a machine is turned on, we consider it to be "Initializing" (state 0), and expect it to issue event b indicating that it is now on. This leads to state 1, whose meaning is simply "I am ON". At this point, the machine is supposed to perform a diagnostic test resulting in one of two events, a_1 or a_2. Event a_1 indicates "My status is OK", while event a_2 indicates "I have a problem". Finally, the machine must inform us that the initial checkup procedure is complete by issuing another event b, leading to state 4 whose meaning is "Status report done". Any sequence that is not of the above form leads to state 5, which should be interpreted as an "Error" state. As for any event occurring after the "Status report done" state is entered, it is simply ignored for the purposes of this task, whose marked (and final in this case) state has already been reached.

We observe that in this example, the focus is on the language marked by the automaton. Since the automaton is designed to accept input events issued by the system, all unexpected inputs, that is, those not in the set $\overline{\mathcal{L}_m(G)}$, send the automaton to state 5. This means that f is a total function; this ensures that the automaton is able to process any input, expected or not. In fact, state 5 is a state where livelock occurs, since it is impossible to reach marked state 4 from state 5. Therefore, our automaton model is a blocking automaton. Finally, we note that states 2 and 3 could be merged without affecting the languages generated and marked by the automaton. The issue of minimizing the number of states without affecting the language properties of an automaton will be discussed in Sect. 2.4.3.

Example 2.11 (Queueing system)
Queueing systems form an important class of DES. A simple queueing system was introduced in the previous chapter and is shown again in Fig. 2.6. Customers arrive and request access to a server. If the server is already busy, they wait in the queue. When a customer completes service, he departs from the system and the next customer in queue (if any) immediately receives service. Thus, the events driving this system are:

a: customer arrival

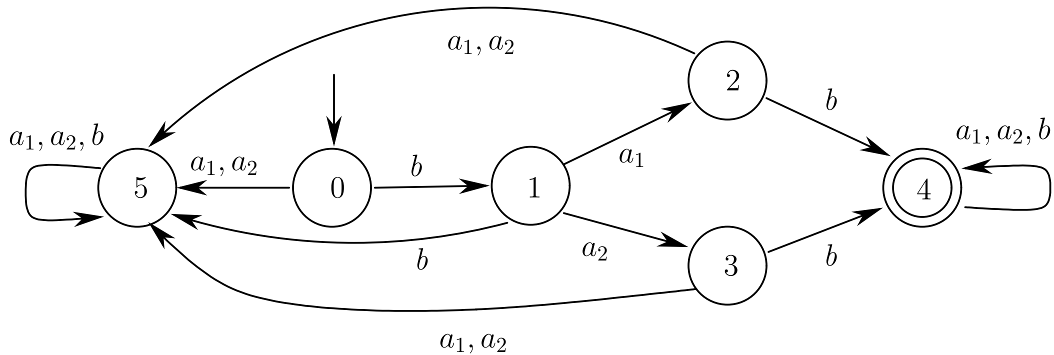

Figure 2.5: Automaton of Example 2.10.
This automaton recognizes event sequences of length 3 or more, which begin with b *and contain* a_1 *or* a_2 *in the second position, followed by* b *in their third position. It can be used as a model of a "status check" system when a machine is turned on, if its states are given the following interpretation: 0 = "Initializing", 1= "I am ON", 2 = "My status is OK", 3 = "I have a problem", 4 = "Status report done", and 5 = "Error".*

d: customer departure.

We can define an infinite-state automaton model G for this system as follows

$$\begin{aligned} E &= \{a, d\} \\ X &= \{0, 1, 2, \ldots\} \\ \Gamma(x) &= \{a, d\} \text{ for all } x > 0, \ \Gamma(0) = \{a\} \\ f(x, a) &= x + 1 \text{ for all } x \geq 0 \qquad (2.4) \\ f(x, d) &= x - 1 \text{ if } x > 0 \qquad (2.5) \end{aligned}$$

Here, the state variable x represents the number of customers in the system (in service and in the queue, if any). The initial state x_0 would be chosen to be the initial number of customers of the system. The feasible event set $\Gamma(0)$ is limited to arrival (a) events, since no departures are possible when the queueing system is empty. Thus, $f(x, d)$ is not defined for $x = 0$. A state transition diagram for this system is shown in Fig. 2.6. Note that the state space in this model is infinite, but countable; also, we have left X_m unspecified.

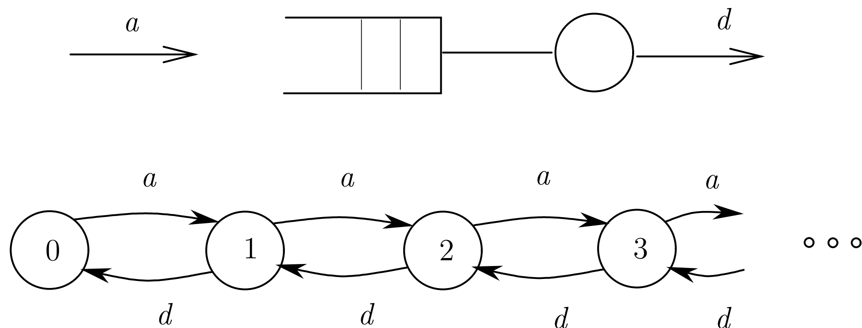

Figure 2.6: Simple queueing system and state transition diagram.
No d *event is included at state 0, since departures are not feasible when* $x = 0$.

Next, let us consider the same simple queueing system, except that now we focus on the state of the server rather than the whole queueing system. The server can be either Idle (denoted by I) or Busy (denoted by B). In addition, we will assume that the server occasionally breaks down. We will refer to this separate state as Down, and denote it by D. When the server breaks down, the customer in service is assumed to be lost; therefore, upon repair, the server is idle.

The events defining the input to this system are assumed to be:

α: service starts

β: service completes

λ: server breaks down

μ: server repaired.

The automaton model of this server is given by

$$
\begin{array}{lll}
E = \{\alpha, \beta, \lambda, \mu\} & X = \{I, B, D\} & \\
\Gamma(I) = \{\alpha\} & f(I, \alpha) = B & \\
\Gamma(B) = \{\beta, \lambda\} & f(B, \beta) = I & f(B, \lambda) = D \\
\Gamma(D) = \{\mu\} & f(D, \mu) = I &
\end{array}
$$

A state transition diagram for this system is shown in Fig. 2.7. An observation worth making in this example is the following. Intuitively, assuming we do not want our server to remain unnecessarily idle, the event "start serving" should always occur immediately upon entering the I state. Of course, this is not possible when the queue is empty, but this model has no knowledge of the queue length. Thus, we limit ourselves to observations of α events by treating them as purely exogenous.

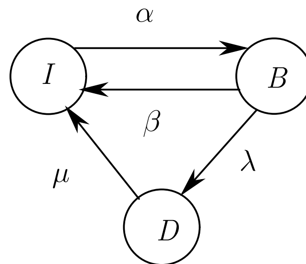

Figure 2.7: State transition diagram for a server with breakdowns.

2.2.4 Nondeterministic Automata

In our definition of deterministic automaton, the initial state is a single state, all transitions have event labels $e \in E$, and the transition function is deterministic in the sense that if event $e \in \Gamma(x)$, then e causes a transition from x to a unique state $y = f(x, e)$. For modeling and analysis purposes, it becomes necessary to relax these three requirements. First, an event e at state x may cause transitions to more than one states. The reason why we may want to allow for this possibility could simply be our own ignorance. Sometimes, we

cannot say with certainty what the effect of an event might be. Or it may be the case that some states of an automaton need to be merged, which could result in multiple transitions with the same label out of the merged state. In this case, $f(x, e)$ should no longer represent a single state, but rather a *set* of states. Second, the label ε may be present in the state transition diagram of an automaton, that is, some transitions between distinct states could have the empty string as label. Our motivation for including so-called "ε-transitions" is again our own ignorance. These transitions may represent events that cause a change in the internal state of a DES but are not "observable" by an outside observer – imagine that there is no sensor that records this state transition. Thus the outside observer cannot attach an event label to such a transition but it recognizes that the transition may occur by using the ε label for it. Or it may also be the case that in the process of analyzing the behavior of the system, some transition labels need to be "erased" and replaced by ε. Of course, if the transition occurs, the "label" ε is not seen, since ε is the identity element in string concatenation. Third, it may be that the initial state of the automaton is not a single state, but is one among a set of states. Motivated by these three observations, we generalize the notion of automaton and define the class of *nondeterministic* automata.

Definition. (Nondeterministic automaton)
A *Nondeterministic Automaton*, denoted by G_{nd}, is a six-tuple

$$G_{nd} = (X, E \cup \{\varepsilon\}, f_{nd}, \Gamma, x_0, X_m)$$

where these objects have the same interpretation as in the definition of deterministic automaton, with the two differences that:

1. f_{nd} is a function $f_{nd} : X \times E \cup \{\varepsilon\} \rightarrow 2^X$, that is, $f_{nd}(x, e) \subseteq X$ whenever it is defined.
2. The *initial* state may itself be a set of states, that is $x_0 \subseteq X$. ♦

We present two examples of nondeterministic automata.

Example 2.12 (A simple nondeterministic automaton)
Consider the finite-state automaton of Fig. 2.8. Note that when event a occurs at state 0, the resulting transition is either to state 1 or back to state 0. The state transition mappings are: $f_{nd}(0, a) = \{0, 1\}$ and $f_{nd}(1, b) = \{0\}$ where the values of f_{nd} are expressed as subsets of the state set X. The transitions $f_{nd}(0, b)$ and $f_{nd}(1, a)$ are not defined. This automaton marks any string of a events, as well as any string containing ab if b is immediately followed by a or ends the string.

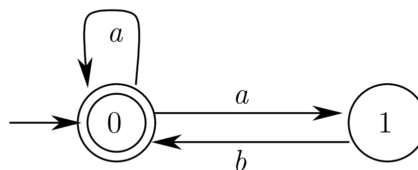

Figure 2.8: Nondeterministic automaton of Example 2.12.

Example 2.13 (A nondeterministic automaton with ε-transitions)
The automaton in Fig. 2.9 is nondeterministic and includes an ε-transition. We have that: $f_{nd}(1,b) = \{2\}$, $f_{nd}(1,\varepsilon) = \{3\}$, $f_{nd}(2,a) = \{2,3\}$, $f_{nd}(2,b) = \{3\}$, and $f_{nd}(3,a) = \{1\}$. Suppose that after turning the system "on" we observe event a. The transition $f_{nd}(1,a)$ is not defined. We conclude that there must have been a transition from state 1 to state 3, followed by event a; thus, immediately after event a, the system is in state 1, although it could move again to state 3 without generating an observable event label. Suppose the string of observed events is baa. Then the system could be in any of its three states, depending of which a's are executed.

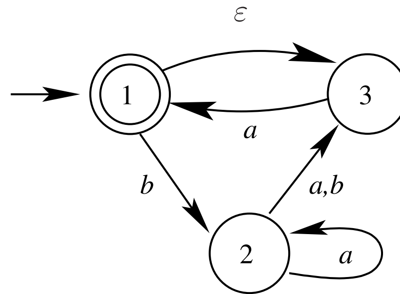

Figure 2.9: Nondeterministic automaton with ε-transition of Example 2.13.

In order to characterize the strings generated and marked by nondeterministic automata, we need to extend f_{nd} to domain $X \times E^*$, just as we did for the transition function f of deterministic automata. For the sake of clarity, let us denote the extended function by f_{nd}^{ext}. Since $\varepsilon \in E^*$, the first step is to define $f_{nd}^{ext}(x,\varepsilon)$. In the deterministic case, we set $f(x,\varepsilon) = x$. In the nondeterministic case, we start by defining the *ε-reach* of a state x to be the set of all states that can be reached from x by following transitions labeled by ε in the state transition diagram. This set is denoted by $\varepsilon R(x)$. By convention, $x \in \varepsilon R(x)$. One may think of $\varepsilon R(x)$ as the *uncertainty set* associated with state x, since the occurrence of an ε-transition is not observed. In the case of a set of states $B \in X$,

$$\varepsilon R(B) = \cup_{x \in B} \varepsilon R(x) \tag{2.6}$$

The construction of f_{nd}^{ext} over its domain $X \times E^*$ proceeds recursively as follows. First, we set

$$f_{nd}^{ext}(x,\varepsilon) := \varepsilon R(x) \tag{2.7}$$

Second, for $u \in E^*$ and $e \in E$, we set

$$f_{nd}^{ext}(x,ue) := \varepsilon R[\{z : z \in f_{nd}(y,e) \text{ for some state } y \in f_{nd}^{ext}(x,u)\}] \tag{2.8}$$

In words, we first identify all states that are reachable from x through string u (recursive part of definition). This is the set of states $f_{nd}^{ext}(x,u)$. We can think of this set as the uncertainty set after the occurrence of string u from state x. Then, we identify among those states all states y at which event e is defined, thereby reaching a state $z \in f_{nd}(y,e)$. Finally, we take the ε-reach of all these z states, which gives us the uncertainty set after the occurrence of string ue. Note the necessity of using a superscript to differentiate f_{nd}^{ext} from f_{nd} since these two functions are not equal over their common domain. In general, for any $\sigma \in E \cup \{\varepsilon\}$, $f_{nd}(x,\sigma) \subseteq f_{nd}^{ext}(x,\sigma)$, as the extended function includes the ε-reach.

As an example, $f_{nd}^{ext}(0, ab) = \{0\}$ in the automaton of Fig. 2.8, since: (i) $f_{nd}(0, a) = \{0, 1\}$ and (ii) $f_{nd}(1, b) = \{0\}$ and $f_{nd}(0, b)$ is not defined. Regarding the automaton of Fig. 2.9, while $f_{nd}(1, \varepsilon) = 3$, $f_{nd}^{ext}(1, \varepsilon) = \{1, 3\}$. In addition, $f_{nd}^{ext}(3, a) = \{1, 3\}$, $f_{nd}^{ext}(1, baa) = \{1, 2, 3\}$, and $f_{nd}^{ext}(1, baabb) = \{3\}$.

Equipped with the extended state transition function, we can characterize the languages generated and marked by nondeterministic automata. These are defined as follows

$$\begin{aligned}
\mathcal{L}(G_{nd}) &= \{s \in E^* : (\exists x \in x_0)\ [f_{nd}^{ext}(x, s) \text{ is defined}]\} \\
\mathcal{L}_m(G_{nd}) &= \{s \in \mathcal{L}(G_{nd}) : (\exists x \in x_0)\ [f_{nd}^{ext}(x, s) \cap X_m \neq \emptyset]\}
\end{aligned}$$

These definitions mean that a string is in the language generated by the nondeterministic automaton if there exists a path in the state transition diagram that is labeled by that string. If it is possible to follow a path that is labeled consistently with a given string and ends in a marked state, then that string is in the language marked by the automaton. For instance, the string aa is in the language marked by the automaton in Fig. 2.8 since we can do two self-loops and stay at state 0, which is marked; it does not matter that the same string can also take us to an unmarked state (1 in this case).

In our study of untimed DES in this and the next chapter, the primary source of nondeterminism will be the limitations of the sensors attached to the system, which will result in *unobservable* events in the state transition diagram of the automaton model. The occurrence of an unobservable event is thus equivalent to the occurrence of an ε-transition from the point of view of an outside observer. In our study of stochastic DES in Chap. 6 and beyond, nondeterminism will arise as a consequence of the stochastic nature of the model. We will compare the language modeling power of nondeterministic and deterministic automata in Sect. 2.3.4.

2.2.5 Automata with Inputs and Outputs

There are two variants to the definition of automaton given in Sect. 2.2.2 that are useful in system modeling: Moore automaton and Mealy automaton. (These kinds of automata are named after E. F. Moore and G. H. Mealy who defined them in 1956 and 1955, respectively.) The differences with the definition in Sect. 2.2.2 are quite simple and are depicted in Fig. 2.10.

- Moore automata are *automata with (state) outputs*. There is an output function that assigns an output to each state. The output associated with a state is shown in bold above the state in Fig. 2.10. This output is "emitted" by the automaton when it enters the state. We can think of "standard" automata as having two outputs: "unmarked" and "marked". Thus, we can view state outputs in Moore automata as a generalization of the notion of marking.

- Mealy automata are *input/output automata*. Transitions are labeled by "events" of the form *input event/output event*, as shown in Fig. 2.10. The set of output events, say E_{output}, need not be the same as the set of input events, E. The interpretation of a transition e_i/e_o from state x to state y is as follows: When the system is in state x, if the automaton "receives" input event e_i, it will make a transition to state y and in that process will "emit" the output event e_o.

One can see how the notions of state output and output event could be useful in building models of DES. Let us refer to systems composed of interacting electromechanical components (e.g., assembly and manufacturing systems, engines, process control systems, heating

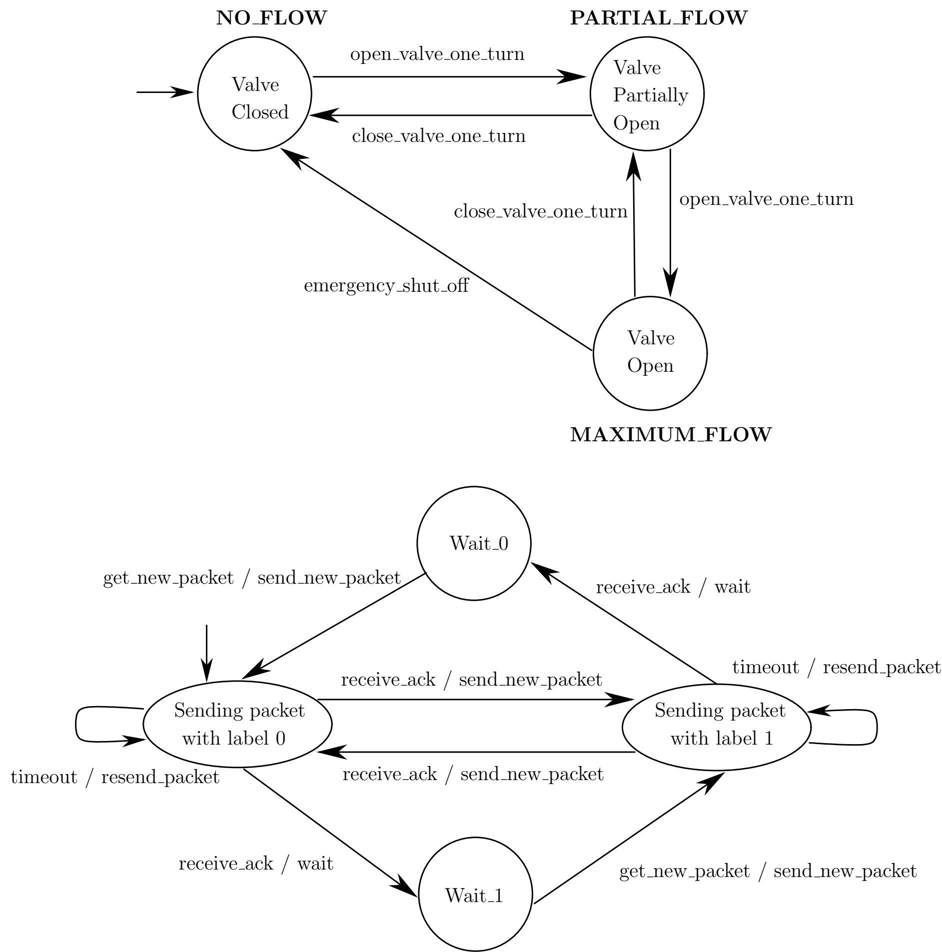

Figure 2.10: Automaton with output (Moore automaton) and input/output automaton (Mealy automaton).
The Moore automaton models a valve together with its flow sensor; the output of a state – indicated in bold next to the state – is the flow sensor reading for that state. The Mealy automaton models the sender in the context of the "Alternating Bit Protocol" for transmission of packets between two nodes in a communication network (half-duplex link in this case). Packets are labeled by a single bit in order to differentiate between subsequent transmissions. (See, e.g., Chapter 6 of Communication Networks. A First Course *by J. Walrand, McGraw-Hill, 1998, for further details on such protocols).*

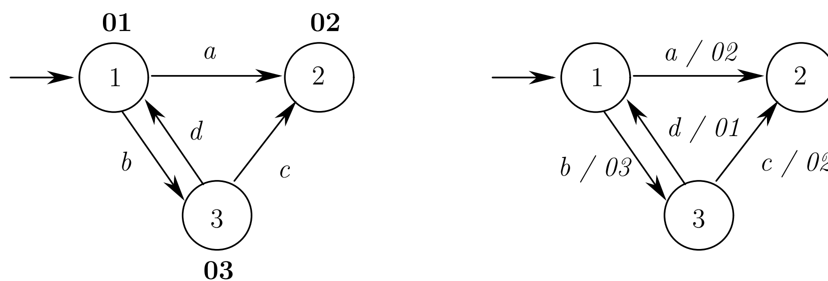

Figure 2.11: Conversion of automaton with output.
The output of each transition in the equivalent Mealy automaton on the right is the output of the state entered by the same transition in the Moore automaton on the left.

and air conditioning units, and so forth) as *physical systems*. These systems are usually equipped with a set of sensors that record the *physical state* of the system. For instance, an air handling system would be equipped with valve flow sensors, pump pressure sensors, thermostats, etc. Thus, Moore automata are a good class of models for such systems, where the output of a state is the set of readings of all sensors when the system is in that (physical) state. Mealy automata are also a convenient class of models since the notion of input-output mapping is central in system and control theory (cf. Sect. 1.2 in Chap. 1). In the modeling of communication protocols for instance, input/output events could model that upon reception of a certain message, the input event, a protocol entity issues a new message, the output event. The same viewpoint applies to software systems in general.

We claim that for the purposes of this book, we can always interpret the behavior of Mealy and Moore automata according to the dynamics of standard automata (i.e., as defined in Sect. 2.2.2). For Mealy automata, this claim is based on the following interpretation. We can view a Mealy automaton as a standard one where events are of the form *input/output*. That is, we view the set E of events as the set of all input/output labels of the Mealy automaton. In this context, the language generated by the automaton will be the set of all *input/output strings* that can be generated by the Mealy automaton. Thus, the material presented in this chapter applies to Mealy automata, when those are interpreted in the above manner. For Moore automata, we can view the state output as the output event associated to all events that enter that state; refer to Fig. 2.11. This effectively transforms the Moore automaton into a Mealy automaton, which can then be interpreted as a standard automaton as we just described. We will revisit the issue of conversion from Moore to standard automata in Sect. 2.5.2 at the end of this chapter.

The above discussion is not meant to be rigorous but rather its purpose is to make the reader aware that most of the material presented in this book for standard automata is meaningful to Moore or Mealy automata.

2.3 OPERATIONS ON AUTOMATA

In order to analyze DES modeled by automata, we need to have a set of operations on a single automaton in order to modify appropriately its state transition diagram according, for instance, to some language operation that we wish to perform. We also need to define operations that allow us to combine, or compose, two or more automata, so that models of complete systems could be built from models of individual system components. This is the first focus of this section; unary operations are covered in Sect. 2.3.1, composition operations are covered in Sect. 2.3.2, and refinement of the state transition function of an automaton is discussed in Sect. 2.3.3. The second focus of this section is on nondeterministic automata. In Sect. 2.3.4 we will define a special kind of automaton, called *observer automaton*, that is a deterministic version of a given nondeterministic automaton preserving language equivalence. The third focus is to revisit the notion of equivalence of automata in Sect. 2.3.5 and consider more general notions than language equivalence. As in the preceding section, our discussion does not require the state set of an automaton to be finite.

2.3.1 Unary Operations

In this section, we consider operations that alter the state transition diagram of an automaton. The event set E remains unchanged.

Accessible Part

From the definitions of $\mathcal{L}(G)$ and $\mathcal{L}_m(G)$, we see that we can delete from G all the states that are not *accessible* or *reachable* from x_0 by some string in $\mathcal{L}(G)$, without affecting the languages generated and marked by G. When we "delete" a state, we also delete all the transitions that are *attached* to that state. We will denote this operation by $Ac(G)$, where Ac stands for taking the "accessible" part. Formally,

$$\begin{array}{rcl} Ac(G) & := & (X_{ac}, E, f_{ac}, x_0, X_{ac,m}) \quad \text{where} \\ X_{ac} & = & \{x \in X : (\exists s \in E^*)\ [f(x_0, s) = x]\} \\ X_{ac,m} & = & X_m \cap X_{ac} \\ f_{ac} & = & f|_{X_{ac} \times E \to X_{ac}} \end{array}$$

The notation $f|_{X_{ac} \times E \to X_{ac}}$ means that we are restricting f to the smaller domain of the accessible states X_{ac}.

Clearly, the Ac operation has no effect on $\mathcal{L}(G)$ and $\mathcal{L}_m(G)$. Thus, from now on, we will always assume, without loss of generality, that an automaton is *accessible*, that is, $G = Ac(G)$.

Coaccessible Part

A state x of G is said to be *coaccessible to* X_m, or simply coaccessible, if there is a path in the state transition diagram of G from state x to a marked state. We denote the operation of deleting all the states of G that are *not* coaccessible by $CoAc(G)$, where $CoAc$ stands for taking the "coaccessible" part.

Taking the coaccessible part of an automaton means building

$$\begin{array}{rcl} CoAc(G) & := & (X_{coac}, E, f_{coac}, x_{0,coac}, X_m) \quad \text{where} \\ X_{coac} & = & \{x \in X : (\exists s \in E^*)\ [f(x, s) \in X_m]\} \\ x_{0,coac} & = & \begin{cases} x_0 & \text{if } x_0 \in X_{coac} \\ \text{undefined} & \text{otherwise} \end{cases} \\ f_{coac} & = & f|_{X_{coac} \times E \to X_{coac}} \end{array}$$

The $CoAc$ operation may shrink $\mathcal{L}(G)$, since we may be deleting states that are accessible from x_0; however, the $CoAc$ operation does not affect $\mathcal{L}_m(G)$, since a deleted state cannot be on any path from x_0 to X_m. If $G = CoAc(G)$, then G is said to be *coaccessible*; in this case, $\mathcal{L}(G) = \overline{\mathcal{L}_m(G)}$.

Coaccessibility is closely related to the concept of *blocking*; recall that an automaton is said to be blocking if $\mathcal{L}(G) \neq \overline{\mathcal{L}_m(G)}$. Therefore, blocking necessarily means that $\overline{\mathcal{L}_m(G)}$ is a proper subset of $\mathcal{L}(G)$ and consequently there are accessible states that are not coaccessible.

Note that if the $CoAc$ operation results in $X_{coac} = \emptyset$ (this would happen if $X_m = \emptyset$ for instance), then we obtain the empty automaton.

Trim Operation

An automaton that is both accessible and coaccessible is said to be *trim*. We define the $Trim$ operation to be

$$Trim(G) := CoAc[Ac(G)] = Ac[CoAc(G)]$$

where the commutativity of Ac and $CoAc$ is easily verified.

Projection and Inverse Projection

Let G have event set E. Consider $E_s \subset E$. The projections of $\mathcal{L}(G)$ and $\mathcal{L}_m(G)$ from E^* to E_s^*, $P_s[\mathcal{L}(G)]$ and $P_s[\mathcal{L}_m(G)]$, can be implemented on G by replacing all transition labels in $E \setminus E_s$ by ε. The result is a nondeterministic automaton that generates and marks the desired languages. An algorithm for transforming it into a language-equivalent deterministic one will be presented in Sect. 2.3.4.

Regarding inverse projection, consider the languages $K_s = \mathcal{L}(G) \subseteq E_s^*$ and $K_{m,s} = \mathcal{L}_m(G)$ and let E_l be a larger event set such that $E_l \supset E_s$. Let P_s be the projection from E_l^* to E_s^*. An automaton that generates $P_s^{-1}(K_s)$ and marks $P_s^{-1}(K_{m,s})$ can be obtained by adding self-loops for all the events in $E_l \setminus E_s$ at all the states of G.

Remark. The Ac, $CoAc$, $Trim$, and projection operations are defined and performed similarly for nondeterministic automata.

Complement

Let us suppose that we have a trim deterministic automaton $G = (X, E, f, \Gamma, x_0, X_m)$ that marks the language $L \subseteq E^*$. Thus G generates the language $\overline{L}$. We wish to build another automaton, denoted by G^{comp}, that will mark the language $E^* \setminus L$. G^{comp} is built by the $Comp$ operation. This operation proceeds in two steps.

The first step of the $Comp$ operation is to "complete" the transition function f of G and make it a total function; let us denote the new transition function by f_{tot}. This is done by adding a new state x_d to X, often called the "dead" or "dump" state. All undefined $f(x, e)$ in G are then assigned to x_d. Formally,

$$f_{tot}(x, e) = \begin{cases} f(x, e) & \text{if } e \in \Gamma(x) \\ x_d & \text{otherwise} \end{cases}$$

Moreover, we set $f_{tot}(x_d, e) = x_d$ for all $e \in E$. Note also that the new state x_d is not marked. The new automaton

$$G_{tot} = (X \cup \{x_d\}, E, f_{tot}, x_0, X_m)$$

is such that $\mathcal{L}(G_{tot}) = E^*$ and $\mathcal{L}_m(G_{tot}) = L$.

The second step of the $Comp$ operation is to change the marking status of all states in G_{tot} by marking all unmarked states (including x_d) and unmarking all marked states. That is, we define

$$Comp(G) := (X \cup \{x_d\}, E, f_{tot}, x_0, (X \cup \{x_d\}) \setminus X_m)$$

Clearly, if $G^{comp} = Comp(G)$, then $\mathcal{L}(G^{comp}) = E^*$ and $\mathcal{L}_m(G^{comp}) = E^* \setminus \mathcal{L}_m(G)$, as desired.

Example 2.14 (Unary operations)
Consider the automaton G depicted in Fig. 2.12. It is a slight variation of the automaton of Fig. 2.4. The new state 6 that has been added is clearly not accessible from state 0. To get $Ac(G)$, it suffices to delete state 6 and the two transitions attached to it; the result is as in Fig. 2.4.

In order to get $CoAc(G)$, we need to identify the states of G that are not coaccessible to the marked state 2. These are states 3, 4, and 5. Deleting these states and the transitions attached to them, we get $CoAc(G)$ depicted in Fig. 2.13 (a). Note that state 6 is not deleted since it can reach state 2. $Trim(G)$ is shown in Fig. 2.13 (b). We can see that the order in which the operations Ac and $CoAc$ are taken does not affect the final result.

Finally, let us take the complement of $Trim(G)$. First, we add a new state, labeled state d, and complete the transition function using this state. Note that $E = \{a, b, g\}$. Next, we reverse the marking of the states. The resulting automaton is shown in Fig. 2.13 (c).

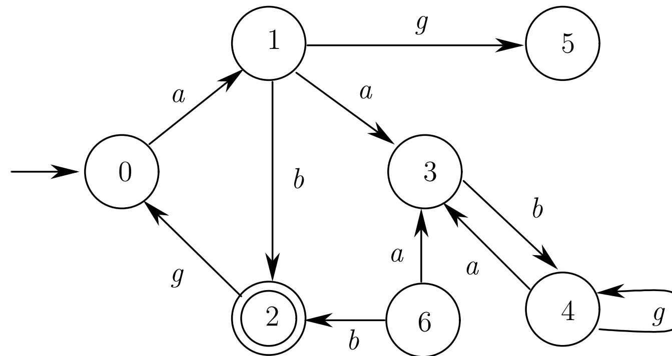

Figure 2.12: Automaton G of Example 2.14.

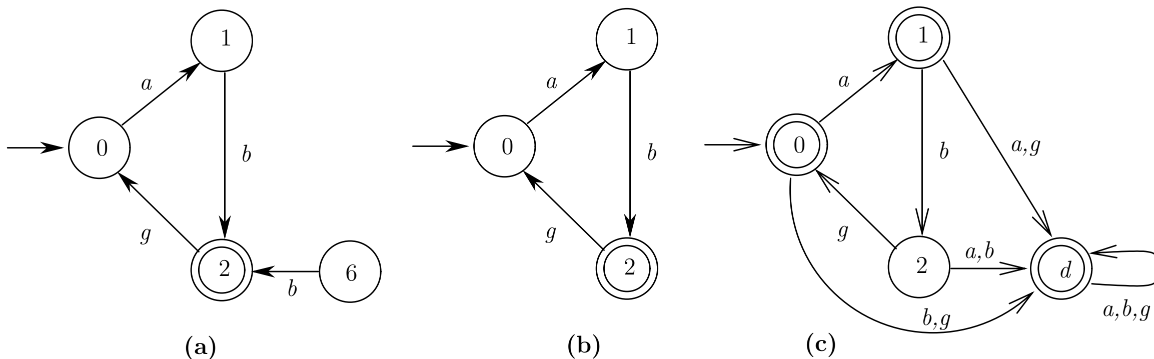

Figure 2.13: (**a**) $CoAc(G)$, (**b**) $Trim(G)$, and (**c**) $Comp[Trim(G)]$, for G of Fig. 2.12 (Example 2.14).

2.3.2 Composition Operations

We define two operations on automata: product, denoted by $\times$, and parallel composition, denoted by $||$. Parallel composition is often called synchronous composition and

product is sometimes called completely synchronous composition. These operations model two forms of joint behavior of a set of automata that operate concurrently. For simplicity, we present these operations for two deterministic automata: (i) They are easily generalized to the composition of a set of automata using the associativity properties discussed below; (ii) Nondeterministic automata can be composed using the same rules for the joint transition function.

We can think of $G_1 \times G_2$ and $G_1||G_2$ as two types of interconnection of system components G_1 and G_2 with event sets E_1 and E_2, respectively, as depicted in Fig. 2.14. As we will see, the key difference between these two operations pertains to how private events, i.e., events that are not in $E_1 \cap E_2$, are handled.

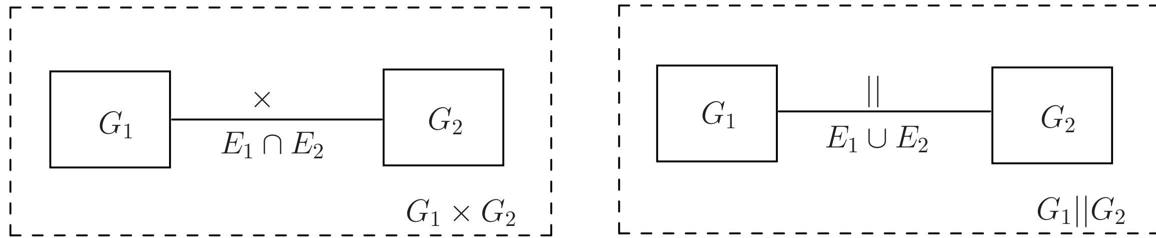

Figure 2.14: This figure illustrates the interconnection of two automata.
The $\times$ operation involves only events in $E_1 \cap E_2$ while the $||$ operation involves all events in $E_1 \cup E_2$.

Consider the two automata

$$G_1 = (X_1, E_1, f_1, \Gamma_1, x_{01}, X_{m1}) \text{ and } G_2 = (X_2, E_2, f_2, \Gamma_2, x_{02}, X_{m2})$$

As mentioned earlier, G_1 and G_2 are assumed to be accessible; however, they need not be coaccessible. No assumptions are made at this point about the two event sets E_1 and E_2.

Product

The *product* of G_1 and G_2 is the automaton

$$G_1 \times G_2 := Ac(X_1 \times X_2, E_1 \cup E_2, f, \Gamma_{1\times 2}, (x_{01}, x_{02}), X_{m1} \times X_{m2})$$

where

$$f((x_1, x_2), e) := \begin{cases} (f_1(x_1, e), f_2(x_2, e)) & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

and thus $\Gamma_{1\times 2}(x_1, x_2) = \Gamma_1(x_1) \cap \Gamma_2(x_2)$. Observe that we take the accessible part on the right-hand side of the definition of $G_1 \times G_2$ since, as was mentioned earlier, we only care about the accessible part of an automaton.

In the product, the transitions of the two automata must always be synchronized on a common event, that is, an event in $E_1 \cap E_2$. $G_1 \times G_2$ thus represents the "lock-step" interconnection of G_1 and G_2, where an event occurs if and only if it occurs in both automata. The states of $G_1 \times G_2$ are denoted by pairs, where the first component is the (current) state of G_1 and the second component is the (current) state of G_2. It is easily verified that

$$\begin{aligned} \mathcal{L}(G_1 \times G_2) &= \mathcal{L}(G_1) \cap \mathcal{L}(G_2) \\ \mathcal{L}_m(G_1 \times G_2) &= \mathcal{L}_m(G_1) \cap \mathcal{L}_m(G_2) \end{aligned}$$

This shows that the intersection of two languages can be "implemented" by doing the product of their automaton representations – an important result. If $E_1 \cap E_2 = \emptyset$, then $\mathcal{L}(G_1 \times G_2) = \{\varepsilon\}$; $\mathcal{L}_m(G_1 \times G_2)$ will be either $\emptyset$ or $\{\varepsilon\}$, depending on the marking status of the initial state (x_{01}, x_{02}).

The event set of $G_1 \times G_2$ is defined to be $E_1 \cup E_2$, in order to record the original event sets in case these are needed later on; we comment further on this issue when discussing parallel composition next. However, the active events of $G_1 \times G_2$ will necessarily be in $E_1 \cap E_2$. In fact, not all events in $E_1 \cap E_2$ need be active in $G_1 \times G_2$, as this depends on the joint transition function f.

Properties of product

1. Product is commutative up to a reordering of the state components in composed states.

2. Product is associative and we define

$$G_1 \times G_2 \times G_3 := (G_1 \times G_2) \times G_3 = G_1 \times (G_2 \times G_3)$$

Associativity follows directly from the definition of product. The product of a set of n automata, $\times_1^n G_i = G_1 \times \cdots \times G_n$, is defined similarly. Thus, in $\times_1^n G_i$, all n automata work in lock-step at each transition. At state $(x_1, \ldots, x_n)$ where $x_i \in X_i$, the only feasible events are those in $\Gamma_1(x_1) \cap \cdots \cap \Gamma_n(x_n)$.

Example 2.15 (Product of two automata)

Two state transition diagrams of automata are depicted in Fig. 2.15. The top one is the result of the product of the automata in Figs. 2.1 and 2.2. The entire event set is $\{a, b, g\}$ and the set of common events is $\{a, b\}$. Note that we have denoted states of the product automaton by pairs, with the first component a state of the automaton of Fig. 2.1 and the second component a state of the automaton of Fig. 2.2. At the initial state $(x, 0)$, the only possible common event is a, which takes x to x and 0 to 1; thus the new state is $(x, 1)$. Comparing the active event sets of x and 1 in their respective automata, we see that the only possible common event is a again, which takes x to x and 1 to 1, that is, $(x, 1)$ again. After this, we are done constructing the product automaton. Only a is active in the product automaton since the automaton of Fig. 2.1 never reaches a state where event b is feasible. Observe that state $(x, 1)$ is marked since both x and 1 are marked in their respective automata.

The second automaton in Fig. 2.15 is the result of the product of the automata in Figs. 2.2 and 2.13 (b). Again, the entire event set is $\{a, b, g\}$ and the set of common events is $\{a, b\}$. The only common behavior is string ab, which takes the product automaton to state $(0, 2)$, at which point this automaton deadlocks. Since there are no marked states, the language marked by the product automaton is empty, as expected if we compare the marked languages of the automata in Figs. 2.2 and 2.13 (b).

Parallel Composition

Composition by product is restrictive as it only allows transitions on common events. In general, when modeling systems composed of interacting components, the event set of each component includes *private* events that pertain to its own internal behavior and *common* events that are shared with other automata and capture the coupling among the respective system components. The standard way of building models of entire systems from models of individual system components is by parallel composition.

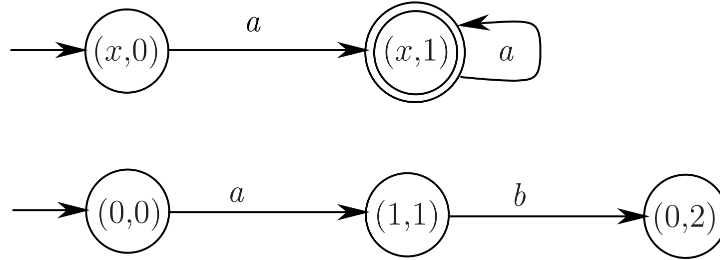

Figure 2.15: Product automata of Example 2.15.
The first automaton is the result of the product of the automata in Figs. 2.1 and 2.2. The second automaton is the result of the product of the automata in Figs. 2.2 and 2.13 (b).

The *parallel composition* of G_1 and G_2 is the automaton

$$G_1 \mid\mid G_2 := Ac(X_1 \times X_2, E_1 \cup E_2, f, \Gamma_{1||2}, (x_{01}, x_{02}), X_{m1} \times X_{m2})$$

where

$$f((x_1, x_2), e) := \begin{cases} (f_1(x_1, e), f_2(x_2, e)) & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f_1(x_1, e), x_2) & \text{if } e \in \Gamma_1(x_1) \setminus E_2 \\ (x_1, f_2(x_2, e)) & \text{if } e \in \Gamma_2(x_2) \setminus E_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and thus $\Gamma_{1||2}(x_1, x_2) = [\Gamma_1(x_1) \cap \Gamma_2(x_2)] \cup [\Gamma_1(x_1) \setminus E_2] \cup [\Gamma_2(x_2) \setminus E_1]$.

In the parallel composition, a common event, that is, an event in $E_1 \cap E_2$, can only be executed if the two automata both execute it simultaneously. Thus, the two automata are "synchronized" on the common events. The private events, that is, those in $(E_2 \setminus E_1) \cup (E_1 \setminus E_2)$, are not subject to such a constraint and can be executed whenever possible. In this kind of interconnection, a component can execute its private events without the participation of the other component; however, a common event can only happen if both components can execute it.

If $E_1 = E_2$, then the parallel composition reduces to the product, since all transitions are forced to be synchronized. If $E_1 \cap E_2 = \emptyset$, then there are no synchronized transitions and $G_1||G_2$ is the *concurrent* behavior of G_1 and G_2. This is often termed the *shuffle* of G_1 and G_2.

In order to precisely characterize the languages generated and marked by $G_1||G_2$ in terms of those of G_1 and G_2, we need to use the operation of language projection introduced earlier, in Sect. 2.2.1. In the present context, let the larger set of events be $E_1 \cup E_2$ and let the smaller set of events be either E_1 or E_2. We consider the two projections

$$P_i : (E_1 \cup E_2)^* \to E_i^* \quad \text{for } i = 1, 2$$

Using these projections, we can characterize the languages resulting from a parallel composition:

1. $\mathcal{L}(G_1||G_2) = P_1^{-1}[\mathcal{L}(G_1)] \cap P_2^{-1}[\mathcal{L}(G_2)]$
2. $\mathcal{L}_m(G_1||G_2) = P_1^{-1}[\mathcal{L}_m(G_1)] \cap P_2^{-1}[\mathcal{L}_m(G_2)]$.

We will not present a formal proof of these results. However, they are fairly intuitive if we relate them to the implementation of inverse projection with self-loops and to the product

of automata. As was mentioned in Sect. 2.3.1, the inverse projections on the right-hand side of the above expressions for $\mathcal{L}(G_1||G_2)$ and $\mathcal{L}_m(G_1||G_2)$ can be implemented by adding self-loops at all states of G_1 and G_2; these self-loops are for events in $E_2 \setminus E_1$ for G_1 and in $E_1 \setminus E_2$ for G_2. Then, we can see that the expression on the right-hand side can be implemented by taking the product of the automata with self-loops. This is correct, as the self-loops guarantee that the private events of each automaton (before the addition of self-loops) will be occurring in the product without any interference from the other automaton, since that other automaton will have a self-loop for this event at every state. Overall, the final result does indeed correspond to the definition of parallel composition. In fact, this discussion shows that parallel composition can be implemented by product, once self-loops have been added in order to lift the event set of each automaton to $E_1 \cup E_2$.

A natural consequence of the above characterization of the behavior of automata under parallel composition using projections is to *define* a parallel composition for *languages*. With $L_i \subseteq E_i^*$ and P_i defined as above, the obvious definition is

$$L_1||L_2 := P_1^{-1}(L_1) \cap P_2^{-1}(L_2) \tag{2.9}$$

Example 2.16 (Parallel composition)
The automaton in Fig. 2.16 is the result of the parallel composition of the automata in Figs. 2.1 and 2.2, referred to as G_1 and G_2, respectively, in this example. (Recall that the product of these automata is shown in Fig. 2.15.) The event set of $G_1||G_2$ is $\{a, b, g\}$. The set of common events is $\{a, b\}$, and G_1 is the only one to have private events, event g in this case. As in the case of the product, the states of $G_1||G_2$ are denoted by pairs. At the initial state $(x, 0)$, the only possible common event is a, which takes $(x, 0)$ to $(x, 1)$, a marked state since x is marked in G_1 and 1 is marked in G_2. In contrast to $G_1 \times G_2$, another transition is possible at $(x, 0)$ in $G_1||G_2$. G_1 can execute event g without the participation of G_2 and take $G_1||G_2$ to the new state $(z, 0)$; G_1 is in state z after event g and G_2 stays in state 0.

We repeat this process in a breadth-first manner and find all possible transitions at $(x, 1)$ and $(z, 0)$, then at all newly generated states, and so forth. We can see that in this example, all six states in $X_1 \times X_2$ are reachable from $(x, 0)$.

Properties of parallel composition:
1. We have the set inclusion

$$P_i[\mathcal{L}(G_1||G_2)] \subseteq \mathcal{L}(G_i), \quad \text{for } i = 1,\ 2$$

The same result holds for the marked language. As is seen in Example 2.16, the coupling of the two automata by common events may prevent some of the strings in their individual generated languages to occur, due to the constraints imposed in the definition of parallel composition regarding these common events.
2. Parallel composition is commutative up to a reordering of the state components in composed states.
3. Parallel composition is associative:

$$(G_1||G_2)||G_3 = G_1||(G_2||G_3)$$

The parallel composition of a set of n automata can therefore be defined using associativity: $G_1||G_2||G_3 := (G_1||G_2)||G_3$. It can also be defined directly by generalizing the preceding

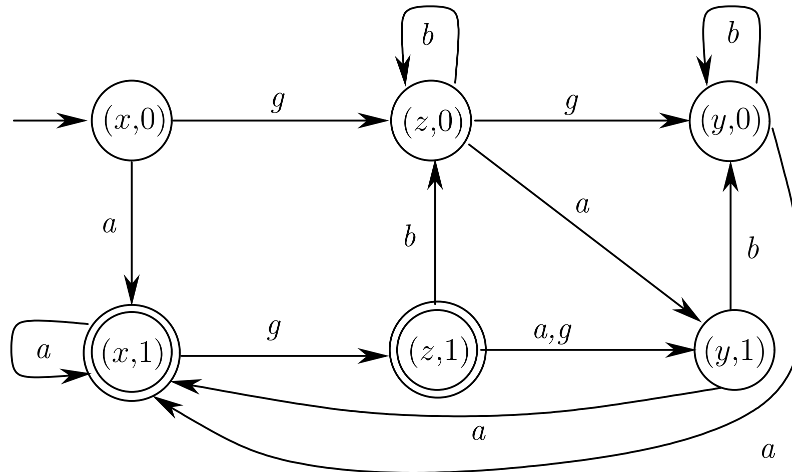

Figure 2.16: Automaton for Example 2.16.
This automaton is the result of the parallel composition of the automata in Figs. 2.1 and 2.2.

definition. In words, we have the following rules: (i) Common events: An event e that is common to *two or more* automata can only occur if all automata G_i for which $e \in E_i$ can execute the event; in this case, these automata perform their respective transitions while the remaining ones (which do not have e in their respective event sets) do not change state. (ii) Private events: An automaton can always execute a private feasible event; in this case the other automata do not change state. We leave it to the reader to write the precise mathematical statement. It is important to observe that care must be taken when exploiting associativity of parallel composition. Consider the following example.

Example 2.17 (Associativity of parallel composition)
Consider the three automata G_1, G_2, and G_3 shown in Fig. 2.17. Their event sets are $E_1 = \{a, b, c\}$, $E_2 = \{a, b, d\}$, and $E_3 = \{a, b, c\}$. The common events are a and b for all pairs G_i and G_j, $i \neq j$, $i, j \in \{1, 2, 3\}$ and c for G_1 and G_3. Let us build $G_{1,2} := G_1||G_2$. It is easily seen that $\mathcal{L}(G_{1,2}) = \overline{\{c\}^*\{a\}\{d\}^*}$ since common event b cannot occur in state $(1, A)$ or in state $(2, B)$, the only two reachable states in $G_{1,2}$. If in the definition of $G_{1,2}$ one "forgets" the original sets E_1 and E_2 and only considers the active events in $G_{1,2}$, which are a, c, and d, then in the parallel composition of $G_{1,2}$ with G_3 one would incorrectly think that the only common events are a and c. In this case, it would appear that event b is a private event of G_3 and thus could be executed at the initial state of $G_{1,2}||G_3$. However, event b is common to all automata and thus it can only occur if all three automata can execute it.

What the above means is that we do need to memorize in the definition of $G_{1,2}$ the original event sets E_1 and E_2. This is why our definition of the event set of $G_1||G_2$ is $E_1 \cup E_2$. In this case, it is recognized that b is a common event of $G_{1,2}$ and G_3, which means that $G_{1,2}||G_3$ will yield an automaton with a single state $(1, A, D)$ and with a self-loop due to event c at that state. This is indeed the correct answer for $G_1||G_2||G_3$: Examining all three automata together, we can see that at the initial state $(1, A, D)$, G_1 can only execute c since the execution of a is prevented by G_3, G_2 cannot execute a for the same reason and it is prevented from executing b by G_1, while G_3 can only execute c jointly with G_1 and is prevented from executing b by G_1.

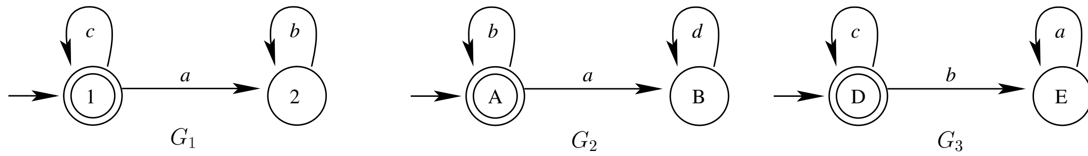

Figure 2.17: Automata of Example 2.17 to illustrate the associativity of parallel composition. *When composing G_1 with G_2, the resulting automaton does not have event b in its set of active events. However, b should still be part of the event set of $G_1||G_2$ in order to record the fact that it is a common event of G_1 (G_2) with G_3.*

The preceding example shows the necessity of having the event set of an automaton defined explicitly rather than implicitly through its set of active events. This issue does not arise in composition by product, since private events do not play a role in this type of composition. However, it cannot be ignored when automata are composed by parallel composition. In fact, the same issue arises when composing languages by parallel composition according to the definition in equation (2.9). It is necessary to keep track of the event set E_i over which each language L_i is defined, and to set the event set of $L_1||L_2$ to be equal to $E_1 \cup E_2$ in order to properly define the projection operations required for further parallel compositions.

Example 2.18 (Two users of two common resources)

A common situation in DES is that of several users sharing a set of common resources. We can present this example in an amusing way by stating it in terms of the traditional story of the *dining philosophers*. For simplicity, we present our model in terms of two philosophers. These two philosophers are seated at a round table where there are two plates of food, one in front of each philosopher, and two forks, one on each side between the plates. The behavior of each philosopher is as follows. The philosopher may be thinking or he may want to eat. In order to go from the "thinking" state to the "eating" state, the philosopher needs to pick up both forks on the table, one at a time, in either order. After the philosopher is done eating, he places both forks back on the table and returns to the "thinking" state. Figure 2.18 shows the automata modeling the two philosophers, denoted by P_1 and P_2, where the events are denoted by *ifj* for "philosopher i picks up fork j" and *jf* for "philosopher j puts both forks down". The event set of each philosopher automaton is equal to its set of active events.

The automata P_1 and P_2 are incomplete to model our system. This is because we need to capture the fact that a fork can only be used by one philosopher at a time. If we build $P_1||P_2$, this will be a shuffle as these two automata have no common events, and thus we will have 16 states reachable from the initial state. In particular, we will have states where a given fork is being used simultaneously by the two philosophers, a situation that is not admissible. Therefore, we add two more automata, one for each fork, to capture the constraint on the resources in this system. These are automata F_1 and F_2 in Fig. 2.18. Their event sets are $\{1f1, 2f1, 1f, 2f\}$ for F_1 and $\{1f2, 2f2, 1f, 2f\}$ for F_2. Fork 1 can be in state "available" or in state "in use". It will go from "available" to "in use" due to either event *1f1* or event *2f1*; it will return to "available" from "in use" upon the occurrence of either *1f* or *2f*.

The complete system behavior, which we denote by PF, is the parallel composition of all four automata, P_1, P_2, F_1, and F_2. This automaton is depicted in Fig. 2.19. Due to the common events between the fork automata and the philosopher automata,

only nine states are reachable out of the 64 possible states. The automaton $PF = P_1||P_2||F_1||F_2$ is blocking as it contains two deadlock states. This happens when each philosopher is holding one fork; according to the models P_1 and P_2, each philosopher waits for the other fork to become available, and consequently they both starve to death in this deadlock. To avoid this deadlock, we would have to add another automaton to serve as *deadlock avoidance controller* for our system.[3] The role of this controller would be to prevent a philosopher from picking up an available fork if the other philosopher is holding the other fork. As an exercise, the reader may build an automaton C for this purpose, so that

$$PF||C = CoAc(PF).$$

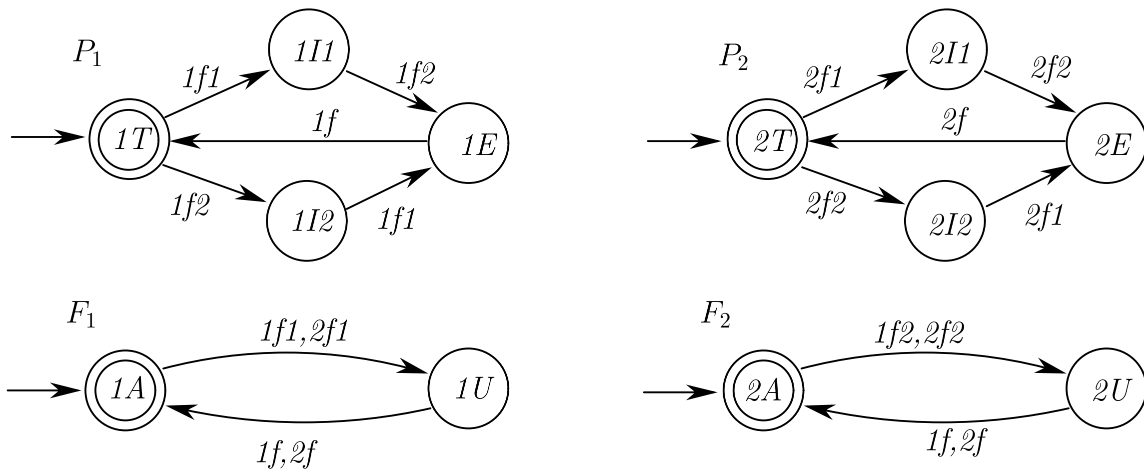

Figure 2.18: The four automata of the dining philosophers example (Example 2.18).

The dining philosophers example shows how parallel composition can be used as a mechanism for *controlling* DES modeled by automata. Each fork automaton can be viewed as a controller for the fork, ensuring that only one philosopher uses the fork at any given time. The controller automaton does not include the dynamics of the system. For instance, automaton F_1 simply enforces the constraint that if event $1f1$ occurs, then event $2f1$ cannot occur until event $1f$ occurs; similarly for event $2f1$; it ignores the other events of G_1 and G_2 that are not relevant to fork 1. This is why parallel composition must be used as the mechanism for coupling the automata; using product would require adding appropriate self-loops at all states of all automata involved. Parallel composition also gives the option of completely preventing a given event from occurring. For instance, if we were to modify the event set of F_1 and add events $1f2, 2f2$ to it without changing the transition function of F_1, then the composition $F_1||(G_1||G_2)$ would effectively disable events $1f2$ and $2f2$ from ever occurring, since they would be common events that F_1 can never execute.

We conclude this section with a discussion on computational issues in DES modeling and analysis. Let us assume that we have a DES composed of 10 components, each modeled by a 5-state automaton. If the event sets of these 10 automata are distinct, then the model of the complete system has 5^{10} states (nearly 10 million), since it corresponds to the shuffle of

[3]The dining philosophers example is revisited in the context of supervisory control in Chap. 3, Example 3.16.

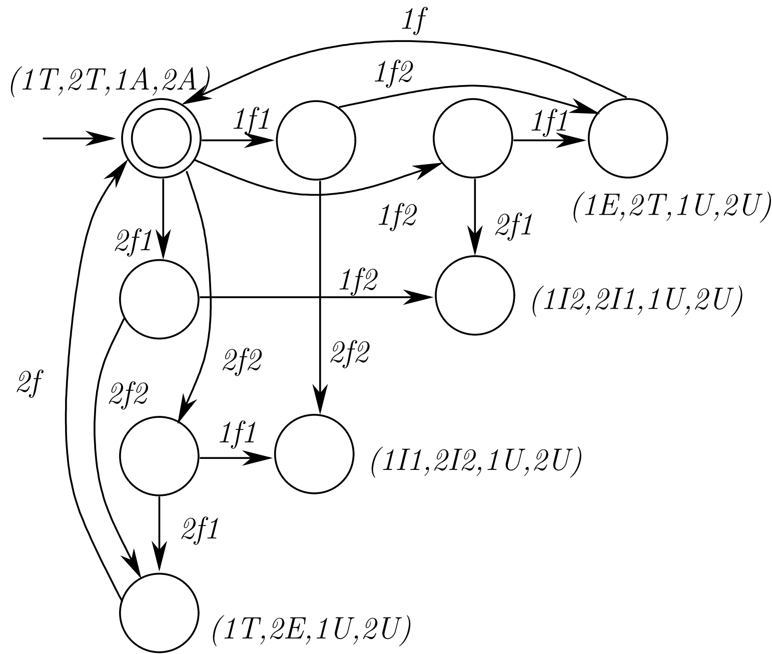

Figure 2.19: The parallel composition of the four automata in Fig. 2.18 (Example 2.18). *For the sake of readability, some state names have been omitted and those that are included are indicated next to the state.*

the 10 automata. This is the origin of the "curse of dimensionality" in DES. This kind of exponential growth need not happen in a parallel composition when there are common events between the components, as we saw in Example 2.18. If we add an 11-th component that models how the above 10 components are interconnected and/or operate (e.g., a controller module), then all the events of that new component will belong to the union of the 10 event sets of the original components. The model of the system of 11 components may then have considerably less than the worst case of $5^{10} \times N_{11}$ states, where N_{11} is the number of states of the 11-th component, due to the synchronization requirements imposed by the common events.

2.3.3 State Space Refinement

Let us suppose that we are interested in comparing two languages $L_1 \subseteq E_1^*$ and $L_2 \subseteq E_2^*$, where $L_1 \subseteq L_2$ and $E_1 \subseteq E_2$, by comparing two automata representations of them, say G_1 and G_2, respectively. For instance, we may want to know what event(s), if any, are possible in L_2 but not in L_1 after string $t \in L_1 \cap L_2$. For simplicity, let us assume that L_1 and L_2 are prefix-closed languages; or, equivalently, we are interested in the languages generated by G_1 and G_2. In order to answer this question, we need to identify what states are reached in G_1 and G_2 after string t and then compare the active event sets of these two states.

In order to make such comparisons more computationally efficient when the above question has to be answered for a large set of strings t's, we would want to be able to "map" the states of G_1 to those of G_2. However, we know that even if $L_1 \subseteq L_2$, there need not be any relationship between the state transition diagram of G_1 and that of G_2. In particular, it could happen that state x_1 of G_1 is reached by strings t_1 and t_2 of $\mathcal{L}(G_1)$ and in G_2, t_1 and t_2 lead to two different states, say y_1 and y_2. This means that state x_1 of G_1 should be

mapped to state y_1 of G_2 if the string of interest is t_1, whereas x_1 of G_1 should be mapped to y_2 of G_2 if the string of interest is t_2. When this happens, it may be convenient to *refine* the state transition diagram of G_1, by adding new states and transitions but without changing the language properties of the automaton, in order to obtain a new automaton whose states could be mapped to those of G_2 by a function that is independent of the string used to reach a state. Intuitively, this means in the above example that we would want to split state x_1 into two states, one reached by t_1, which would then be mapped to y_1 of G_2, and one reached by t_2, which would then be mapped to y_2 of G_2.

This type of refinement, along with the desired function mapping the states of the refined automaton to those of G_2, is easily performed by the product operation as we now describe.

Refinement by Product

To refine the state space of G_1 in the manner described above, it suffices to build

$$G_{1,new} = G_1 \times G_2$$

By our earlier assumption that $L_1 = \mathcal{L}(G_1) \subseteq L_2 = \mathcal{L}(G_2)$, it is clear that $G_{1,new}$ will be language equivalent to G_1. Moreover, since the states of $G_{1,new}$ are pairs (x, y), the second component y of a pair tells us the state that G_2 is in whenever $G_{1,new}$ is in state (x, y). Thus, the desired map from the state space of $G_{1,new}$ to that of G_2 consists of simply reading the second component of a state of $G_{1,new}$.

Example 2.19 (Refinement by product)
Figure 2.20 illustrates how the product operation can refine an automaton with respect to another one. State x_2 of the first automaton can correspond to either y_2 or y_3 of the second automaton, depending on whether it is reached by event a_1 or a_2, respectively. When the first automaton is replaced by the product of the two automata, state x_2 has been split into two states: (x_2, y_2) and (x_2, y_3).

Notion of Subautomaton

An alternative to refinement by product to map states of G_1 to states of G_2 is to require that the state transition diagram of G_1 be a *subgraph* of the state transition diagram of G_2. This idea of subgraph is formalized by the notion of *subautomaton*. We say that G_1 is a subautomaton of G_2, denoted by $G_1 \sqsubseteq G_2$, if

$$f_1(x_{01}, s) = f_2(x_{02}, s) \text{ for all } s \in \mathcal{L}(G_1)$$

Note that this condition implies that $X_1 \subseteq X_2$, $x_{01} = x_{02}$, and $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$. This definition also implies that the state transition diagram of G_1 is a subgraph of that of G_2, as desired. When either G_1 or G_2 has marked states, we have the additional requirement that $X_{m,1} = X_{m,2} \cap X_1$; in other words, marking in G_1 must be consistent with marking in G_2.

This form of correspondence between two automata is stronger than what refinement by product can achieve. In particular, we may have to modify both G_1 and G_2. On the other hand, the subgraph relationship makes it trivial to "match" the states of the two automata. It is not difficult to obtain a general procedure to build G'_1 and G'_2 such that $\mathcal{L}(G'_i) = L_i$, $i = 1, 2$, and $G'_1 \sqsubseteq G'_2$, given any G_i such that $\mathcal{L}(G_i) = L_i$, $i = 1, 2$, and $L_1 \subseteq L_2$:

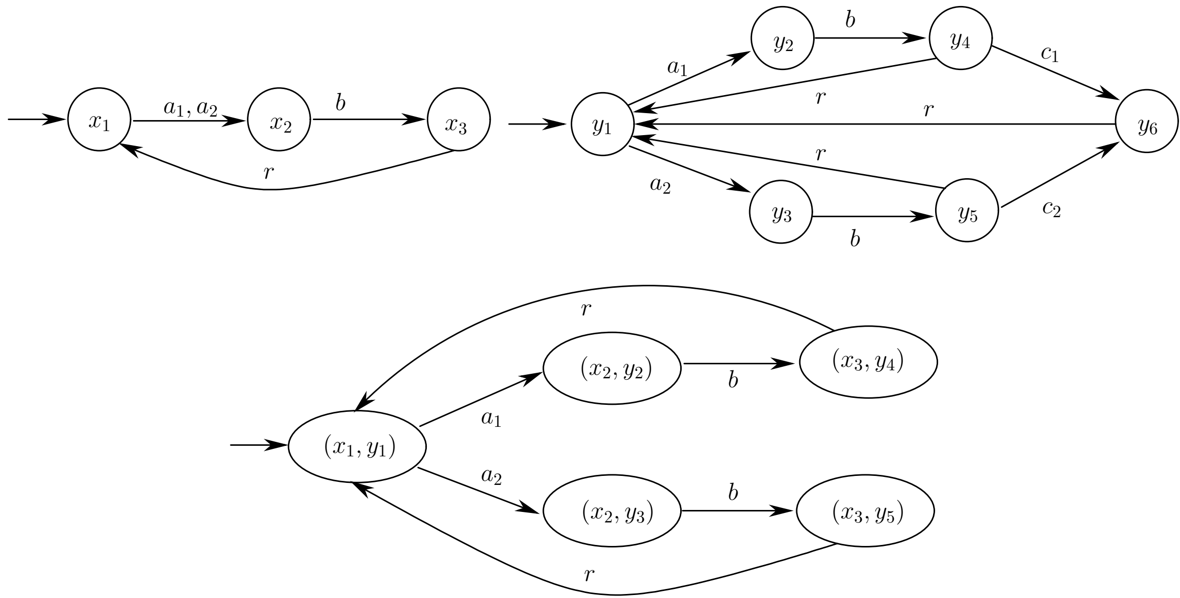

Figure 2.20: Two automata and their product (Example 2.19).

1. Build $G_1' = G_1 \times G_2$;
2. (a) Examine each state x_1 of G_1 and add a self-loop for each event in $E_2 \setminus \Gamma_1(x_1)$ and call the result G_1^{sl};
   (b) Build $G_2' = G_1^{sl} \times G_2$.

The proof of the correctness of this procedure is left to the reader.

2.3.4 Observer Automata

We introduced earlier the class of nondeterministic automata, which differ from deterministic automata by allowing the codomain of f to be 2^X, the power set of the state space of the automaton, and also allowing ε-transitions. The following question then arises: How do deterministic and nondeterministic automata compare in terms of language representation? Let us revisit Example 2.12.

Example 2.20 (An equivalent deterministic automaton)
It is easily verified that the deterministic automaton (let us call it G) depicted in Fig. 2.21 is equivalent to the nondeterministic one in Fig. 2.8 (let us call it G_{nd}). Both automata generate and mark the same languages. In fact, we can think of state A of G as corresponding to state 0 of G_{nd} and of state B of G as corresponding to the set of states $\{0, 1\}$ of G_{nd}. By "corresponds", we mean here that f of G and f_{nd} of G_{nd} match in the sense that:
(i) $f(A, a) = B$ and $f_{nd}(0, a) = \{0, 1\}$;
(ii) $f(A, b)$ and $f_{nd}(0, b)$ are undefined;
(iii) $f(B, a) = B$ and $f_{nd}(0, a) = \{0, 1\}$ with $f_{nd}(1, a)$ undefined; and
(iv) $f(B, b) = A$ and $f_{nd}(1, b) = \{0\}$ with $f_{nd}(0, b)$ undefined.

It turns out that we can always transform a nondeterministic automaton, G_{nd}, into a language-equivalent deterministic one, that is, one that generates and marks the same

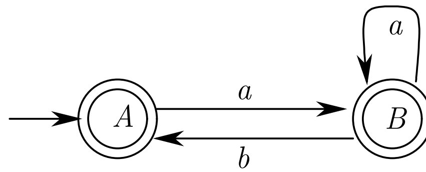

Figure 2.21: Deterministic automaton of Example 2.20.

languages as the original nondeterministic automaton. The state space of the equivalent deterministic automaton will be a subset of the power set of the state space of the nondeterministic one. This means that if the nondeterministic automaton is finite-state, then the equivalent deterministic one will also be finite-state. This latter statement has important implications that will be discussed in Sect. 2.4. The focus of this section is to present an algorithm for this language-preserving transformation from nondeterministic to deterministic automaton. We shall call the resulting equivalent deterministic automaton the *observer* corresponding to the nondeterministic automaton; we will denote the observer of G_{nd} by $Obs(G_{nd})$ and often write G_{obs} when there is no danger of confusion. This terminology is inspired from the concept of observer in system theory; it captures the fact that the equivalent deterministic automaton (the observer) keeps track of the estimate of the state of the nondeterministic automaton upon transitions labeled by events in E. (Recall that the event set of G_{nd} is $E \cup \{\varepsilon\}$.)

Before presenting the details of the algorithm to construct the observer, we illustrate the key points using a simple example.

Example 2.21 (From nondeterministic to deterministic automata)
Consider the nondeterministic automaton G_{nd} in Fig. 2.22, where nondeterminism arises at states 1 and 2, since event b leads to two different states from state 1 and since we have ε-transitions in the active event sets of states 1 and 2. Let us build the automaton G_{obs} from G_{nd}. We start by defining the initial state of G_{obs} and calling it $\{0\}$. Since state 0 is marked in G_{nd}, we mark $\{0\}$ in G_{obs} as well.

First, we analyze state $\{0\}$ of G_{obs}.

- Event a is the only event defined at state 0 in G_{nd}. String a can take G_{nd} to states 1 (via a), 2 (via $a\varepsilon$), and 3 (via $a\varepsilon\varepsilon$) in G_{nd}, so we define a transition from $\{0\}$ to $\{1,2,3\}$, labeled a, in G_{obs}.

Next, we analyze the newly-created state $\{1,2,3\}$ of G_{obs}. We take the union of the active event sets of 1, 2, and 3 in G_{nd} and get events a and b, in addition to ε.

- Event a can only occur from state 2 and it takes G_{nd} to state 0. Thus, we add a transition from $\{1,2,3\}$ to $\{0\}$, labeled a, in G_{obs}.
- Event b can occur in states 1 and 3. From state 1, we can reach states 0 (via b), 1 (via b), 2 (via $b\varepsilon$), and 3 (via $b\varepsilon\varepsilon$). From state 3, we can reach state 0 (via b). Overall, the possible states that can be reached from $\{1,2,3\}$ with string b are 0, 1, 2, and 3. Thus, we add a transition from $\{1,2,3\}$ to $\{0,1,2,3\}$, labeled b, in G_{obs}.

Finally, we analyze the newly-created state $\{0,1,2,3\}$ of G_{obs}. Proceeding similarly as above, we identify the following transitions to be added to G_{obs}:

- A self-loop at $\{0, 1, 2, 3\}$ labeled a;
- A self-loop at $\{0, 1, 2, 3\}$ labeled b.

The first self-loop is due to the fact that from state 0, we can reach states 1, 2, and 3 under a, and from state 2, we can reach state 0 under a. Thus, all of $\{0, 1, 2, 3\}$ is reachable under a from $\{0, 1, 2, 3\}$. Similar reasoning explains the second self-loop. We mark state $\{0, 1, 2, 3\}$ in G_{obs} since state 0 is marked in G_{nd}.

The process of building G_{obs} is completed since all created states have been examined. Automaton G_{obs}, called the observer of G_{nd}, is depicted in Fig. 2.22. It can be seen that G_{nd} and G_{obs} are indeed language equivalent.

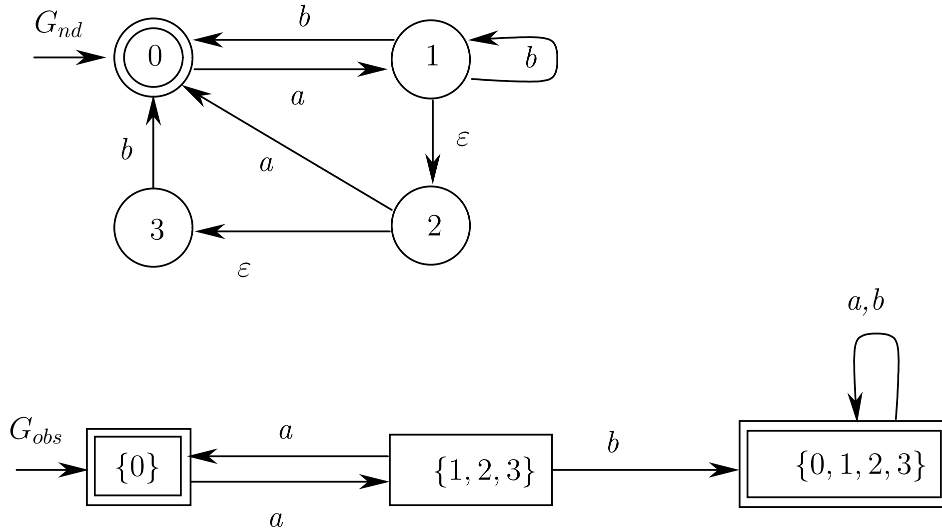

Figure 2.22: Nondeterministic automaton and its (deterministic) observer for Example 2.21.

With the intuition gained from this example, we present the formal steps of the algorithm to construct the observer. Recall from Sect. 2.2.4 the definition of f_{nd}^{ext}, the extension of transition function f_{nd} to strings in E^*; recall also the definition of $\varepsilon R(x)$, the ε-reach of state x.

Procedure for Building Observer $Obs(G_{nd})$ of Nondeterministic Automaton G_{nd}

Let $G_{nd} = (X, E \cup \{\varepsilon\}, f_{nd}, x_0, X_m)$ be a nondeterministic automaton. Then $Obs(G_{nd}) = (X_{obs}, E, f_{obs}, x_{0,obs}, X_{m,obs})$ and it is built as follows.

Step 1: Define $x_{0,obs} := \varepsilon R(x_0)$.

Set $X_{obs} = \{x_{0,obs}\}$.

Step 2: For each $B \in X_{obs}$ and $e \in E$, define

$$f_{obs}(B, e) \quad := \quad \varepsilon R(\{x \in X : (\exists x_e \in B)\ [x \in f_{nd}(x_e, e)]\})$$

whenever $f_{nd}(x_e, e)$ is defined for some $x_e \in B$. In this case, add the state $f_{obs}(B, e)$ to X_{obs}. If $f_{nd}(x_e, e)$ is not defined for any $x_e \in B$, then $f_{obs}(B, e)$ is not defined.

Step 3: Repeat Step 2 until the entire *accessible* part of $Obs(G_{nd})$ has been constructed.

Step 4: $X_{m,obs} := \{B \in X_{obs} : B \cap X_m \neq \emptyset\}$.

We explain the key steps of this algorithm.

- The idea of this procedure is to start with $x_{0,obs}$ as the initial state of the observer. We then identify all the events in E that label all the transitions out of any state in the set $x_{0,obs}$; this results in the active event set of $x_{0,obs}$. For each event e in this active event set, we identify all the states in X that can be reached starting from a state in $x_{0,obs}$. We then extend this set of states to include its ε-reach; this returns the state $f_{obs}(x_{0,obs}, e)$ of $Obs(G_{nd})$. This transition, namely event e taking state $x_{0,obs}$ to state $f_{obs}(x_{0,obs}, e)$, is then added to the state transition diagram of $Obs(G_{nd})$.

  What the above means is that an "outside observer" that knows the system model G_{nd} but only observes the transitions of G_{nd} labeled by events in E will start with $x_{0,obs}$ as its estimate of the state of G_{nd}. Upon observing event $e \in E$, this outside observer will update its state estimate to $f_{obs}(x_{0,obs}, e)$, as this set represents all the states where G_{nd} could be after executing the string e, preceded and/or followed by ε.

- The procedure is repeated for each event in the active event set of $x_{0,obs}$ and then for each state that has been created as an immediate successor of $x_{0,obs}$, and so forth for each successor of $x_{0,obs}$. Clearly, the worst case for all states that could be created is no larger than the set of all non-empty subsets of X.

- Finally, any state of $Obs(G_{nd})$ that contains a marked state of G_{nd} is considered to be marked from the viewpoint of $Obs(G_{nd})$. This is because this state is reachable from $x_{0,obs}$ by a string in $\mathcal{L}_m(G_{nd})$.

The important properties of $Obs(G_{nd})$ are that:

1. $Obs(G_{nd})$ is a deterministic automaton.
2. $\mathcal{L}(Obs(G_{nd})) = \mathcal{L}(G_{nd})$.
3. $\mathcal{L}_m(Obs(G_{nd})) = \mathcal{L}_m(G_{nd})$.

The first result is obvious; the other two results follow directly from the algorithm to construct $Obs(G_{nd})$.

Observer automata are an important tool in the study of partially-observed DES. We will generalize their construction procedure in Sect. 2.5.2 and use them frequently in Chap. 3.

2.3.5 Equivalence of Automata

We established in the preceding section that nondeterministic automata are language equivalent to deterministic automata: Any nondeterministic automaton can be transformed to a language-equivalent deterministic one. However, nondeterminism in an automaton can capture aspects of the system behavior beyond the language generated or marked by the automaton. Consider nondeterministic automaton G in Fig. 2.23. It is language-equivalent to deterministic automaton H in the same figure. However, model G captures the fact that

after executing event a, the system can execute either event b only or event c only, depending on which a transition is chosen out of state 1. This is different from the situation in H, where both b and c are possible after the occurrence of a. Essentially, the modeler chose to use nondeterminism associated with event a in G to capture some aspect of the system behavior; maybe this aspect of the behavior is unknown, or it is due to unmodeled internal behavior, or it is related to the effect of the "environment" on the system, for instance. Thus, in this regard, automata H and G are not "equivalent". This motivates the consideration of more stringent forms of equivalence among automata.

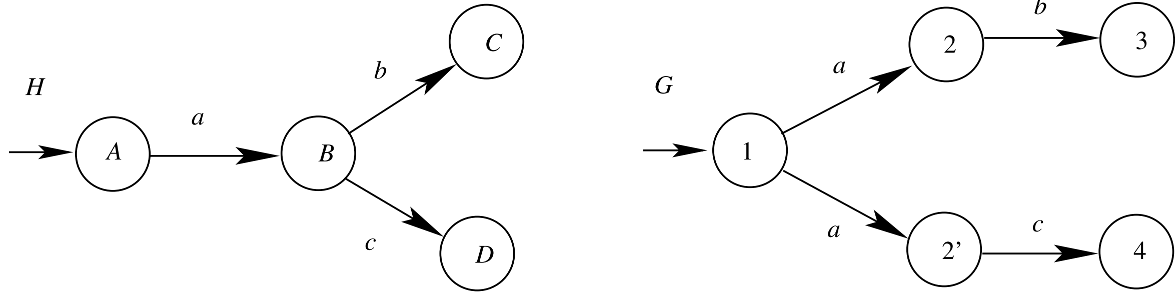

Figure 2.23: Equivalence of automata.
Automata H and G are language equivalent. However, nondeterministic automaton G models the fact that after event a, the system can execute either event b or event c, but not both.

A widely-used notion of equivalence is that of *bisimulation*. It is one of many different semantics that have been proposed and studied in the field of process algebra in computer science. Bisimulation equivalence stipulates that any pair of states reached after a given string of events should have the same future behavior in terms of post-language. In the case of deterministic automata, bisimulation reduces to language equivalence. However, in the case of nondeterministic automata, bisimulation is a stronger form of equivalence that has been deemed desirable in many contexts. This notion of equivalence is formalized by introducing the notion of *bisimulation relation* between two (possibly nondeterministic) automata, $H = (X_H, E, f_H, x_{0,H}, X_{m,H})$ and $G = (X_G, E, f_G, x_{0,G}, X_{m,G})$, with same event set E. Specifically, a bisimulation relation is a set of pairs of states in $X_H \times X_G$ that satisfy certain properties.

For the sake of generality, let us parametrize the definition of bisimulation relation in terms of subsets of X_H, X_G, and E. A *bisimulation relation* between H and G over $S_H \subseteq X_H$, $S_G \subseteq X_G$ and with respect to $E_R \subseteq E$ is a binary relation Φ, $\Phi \subseteq X_H \times X_G$, where the five conditions below are satisfied:

B1. **(a)** For each $x_G \in S_G$ there exists $x_H \in S_H$ such that $(x_H, x_G) \in \Phi$.

(b) For each $x_H \in S_H$ there exists $x_G \in S_G$ such that $(x_H, x_G) \in \Phi$.

B2. **(a)** If $(x_H, x_G) \in \Phi$, $e \in E_R$ and $x'_G \in f_G(x_G, e)$, then there exists x'_H such that $x'_H \in f_H(x_H, e)$ and $(x'_H, x'_G) \in \Phi$.

(b) If $(x_H, x_G) \in \Phi$, $e \in E_R$ and $x'_H \in f_H(x_H, e)$, then there exists x'_G such that $x'_G \in f_G(x_G, e)$ and $(x'_H, x'_G) \in \Phi$.

B3. (Bisimulation with marking) $(x_H, x_G) \in \Phi$ implies $x_H \in X_{m,H}$ iff $x_G \in X_{m,G}$.

Condition B3 is only included if one wishes to consider marked states and marked languages.

Automata H and G are said to be *bisimilar with respect to E_R* if there exists a bisimulation relation between H and G with $S_H = X_H$, $S_G = X_G$, and where $(x_{H,0}, x_{G,0}) \in \Phi$. If this is true when $E_R = E$, then H and G are called bisimilar. It then follows from the definition of Φ that if $(x, y) \in \Phi$, we have that $\mathcal{L}(H(x)) = \mathcal{L}(G(y))$ and $\mathcal{L}_m(H(x)) = \mathcal{L}_m(G(y))$, where $H(x)$ denotes automaton H where the initial state has been set to state x (similarly for $G(y)$). The condition on the marked languages disappears if condition B3 is dropped. In the case of automata H and G in Fig. 2.23, the pairs $(B, 2)$ and $(B, 2')$ and hence the pair $(A, 1)$ are not in Φ if $E_R = E$: States B and 2 do not have same future behavior with respect to event c, while states B and 2' do not have same future behavior with respect to event b. Thus these two automata are not bisimilar. On the other hand, if we restrict attention to event a only, namely, if we pick $E_R = \{a\}$, then H and G are bisimilar with respect to that event set. In this case, the corresponding bisimulation relation would include state pairs $(A, 1), (B, 2), (B, 2'), (C, 3), (D, 4)$.

Note that bisimilar automata need not be isomorphic. For instance, two deterministic automata that mark and generate the same languages will necessarily be bisimilar, but they need not be isomorphic. Bisimilarity is an equivalence relation among automata since it is symmetric, reflexive, and transitive.

We can relax the definition of bisimulation by dropping conditions B1-(b) and B2-(b). In this case, it is only required that every transition defined in G be also defined in H, but not vice-versa. When this holds for all states and all events in G, we say that H *simulates* G since H can always execute what G is executing. Simulation relations are useful when building abstracted models of complex systems. The abstracted model should be able to replicate all system behaviors, but it might include additional behaviors that are not possible in the original system. For instance, in Fig. 2.23, H simulates G, but not vice-versa. Clearly, in general, if H simulates G and G simulates H, then H and G are bisimilar.

2.4 FINITE-STATE AUTOMATA

2.4.1 Definition and Properties of Regular Languages

As we mentioned at the beginning of this chapter, we are normally interested in specifying tasks as event sequences which define some language, and then in obtaining an automaton that can generate or mark (that is, represent) this language. The automaton representation is certainly likely to be more convenient to use than plain enumeration of all the strings in the language. Naturally, a crucial question is: Can we always do that? The answer is "yes", albeit it comes with a "practical" problem. Any language can be marked by an automaton: Simply build the automaton as a (possibly infinite) tree whose root is the initial state and where the nodes at layer n of the tree are entered by the strings of length n or the prefixes of length n of the longer strings. The state space is the set of nodes of the tree and a state is marked if and only if the string that reaches it from the root is an element of the language. Refer to Fig. 2.24. The tree automaton there represents the language

$$L = \{\varepsilon, ab, aabb, aaabbb, \ldots\} = \{a^n b^n : n \geq 0\}$$

over the event set $E = \{a, b\}$. The key point here is that any such tree automaton will have an infinite state space if the cardinality of the language is infinite. Of course, we know that there are infinite languages that can be represented by *finite-state* automata. For example, a single state suffices to represent the language E^*; it suffices to put a self-loop at that state for each event in E. Another example is provided by the automaton in Fig. 2.2.

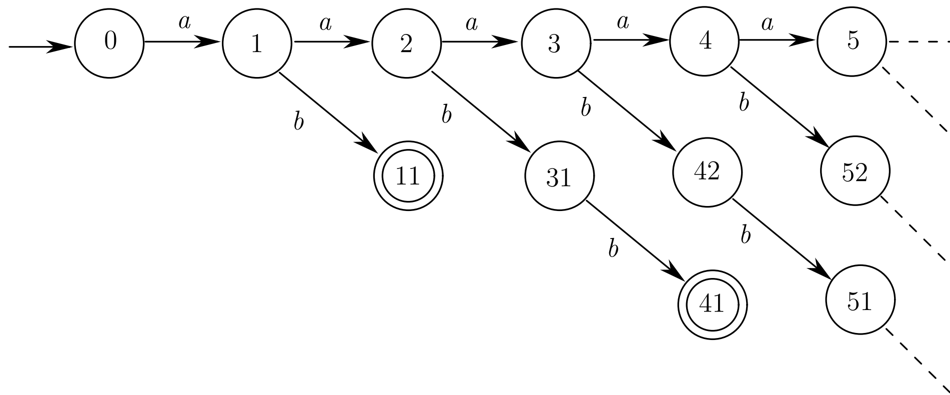

Figure 2.24: Tree automaton marking the language $L = \{a^n b^n : n \geq 0\}$.

The above discussion leads us to ask the question: Are there infinite languages that cannot be represented by finite-state automata? The answer is "yes" and a classic example is the language

$$L = \{a^n b^n : n \geq 0\}$$

mentioned above. To argue that this language cannot be represented by a finite-state automaton, we observe that a marked state must be reached after exactly the same number of b events as that of a events that started the string. Therefore, the automaton must "memorize" how many a events occurred when it starts allowing b events, at which point it must also "memorize" the number of occurrences of b events; this is necessary in order to allow the correct number of b events before entering a marked state. But the number of a events can be arbitrarily large, so any automaton with $2N - 1$ states would not be able to mark the string $a^N b^N$ without marking other strings not in L. Consequently, L cannot be marked by a *finite-state* automaton. This discussion leads us to the following definition.

Definition. (Regular language)
A language is said to be *regular* if it can be marked by a finite-state automaton. The class of regular languages is denoted by $\mathcal{R}$. ♦

We have established that $\mathcal{R}$ is a proper subset of 2^{E^*}. The class $\mathcal{R}$ is very important since it delimits the languages that possess automaton representations that require finite memory when stored in a computer. In other words, automata are a practical means of manipulating *regular* languages in analysis or controller synthesis problems. On the other hand, automata are not a practical means for representing *non-regular* languages, since they would require infinite memory. We will see in Chap. 4 that Petri nets, the other DES modeling formalism considered in this book, can represent some non-regular languages with a finite transition structure.

Nondeterministic Finite-State Automata

We presented in Sect. 2.3.4 a procedure to transform any nondeterministic automaton G_{nd} to an equivalent deterministic one G_{obs}, called the observer of G_{nd}. If we restrict attention to nondeterministic *finite-state* automata, then we can see immediately from the construction procedure for observers that the corresponding observers will be *finite-state* (deterministic) automata. As we noted earlier, the state space of an observer will be a

subset of the power set of the state space of the original nondeterministic automaton; clearly, the power set of a finite set is also a finite set. In other words, for any nondeterministic finite-state automaton, there exists an equivalent deterministic finite-state one.

Theorem. (Regular languages and finite-state automata)
The class of languages representable by nondeterministic finite-state automata is exactly the same as the class of languages representable by deterministic finite-state automata: $\mathcal{R}$.
⧫

This important result does not mean that nondeterministic automata are "useless". We commented earlier how nondeterminism may arise in system modeling. Another aspect is that nondeterministic automata may sometimes require fewer states than deterministic ones to describe certain languages, and this makes them quite useful.

Properties of Regular Languages

The following theorem about the properties of the class of regular languages, along with the proof that we present, illustrates how well-behaved the class $\mathcal{R}$ is and how the duality between regular languages and finite-state automata, be they deterministic or nondeterministic, can be exploited.

Theorem. (Properties of $\mathcal{R}$)
Let L_1 and L_2 be in $\mathcal{R}$. Then the following languages are also in $\mathcal{R}$:

1. $\overline{L_1}$
2. L_1^*
3. $L_1^c := E^* \setminus L_1$
4. $L_1 \cup L_2$
5. $L_1 L_2$
6. $L_1 \cap L_2$.

Proof. Let G_1 and G_2 be two finite-state automata that mark L_1 and L_2, respectively. We prove the first three properties by modifying G_1 in order to obtain finite-state automata that mark $\overline{L_1}$, L_1^*, and L_1^c, respectively. We prove the last three properties by building, from G_1 and G_2, a third finite-state automaton that marks the language obtained after the corresponding operation. Note that in all cases, it does not matter if a finite-state automaton is nondeterministic, as we know we can always build its (deterministic and finite-state) observer to mark the same language.

1. Take the trim of G_1 and then mark all of its states.
2. *Kleene-closure:* Add a new initial state, mark it, and connect it to the old initial state of G_1 by an ε-transition. Then add ε-transitions from every marked state of G_1 to the old initial state. The new finite-state automaton marks L_1^*.
3. *Complement:* This was proved when we considered the complement operation in Sect. 2.3.1; the automaton to build from G_1 has at most one more state than G_1.
4. *Union:* Create a new initial state and connect it, with two ε-transitions, to the initial states of G_1 and G_2. The result is a nondeterministic automaton that marks the union of L_1 and L_2.

5. *Concatenation:* Connect the marked states of G_1 to the initial state of G_2 by ε-transitions. Unmark all the states of G_1. The resulting nondeterministic automaton marks the language $L_1 L_2$.

6. *Intersection:* As was seen in Sect. 2.3.2, $G_1 \times G_2$ marks $L_1 \cap L_2$. □

2.4.2 Regular Expressions

Another way to describe regular languages is through compact expressions that are called *regular expressions.* We have already defined the operations of concatenation, Kleene-closure, and union on languages. We now establish some notational conventions. First, we adopt the symbol "+" rather than "∪" to remind us that this operation is equivalent to the logical "OR" function. Next, suppose we are interested in the Kleene-closure of $\{u\}$, where u is a string. This is given by $\{u\}^* = \{\varepsilon, u, uu, uuu, \ldots\}$ as was seen earlier. Let us agree, however, to omit the braces and write u^* in place of $\{u\}^*$. By extension, let us agree to think of u as either a string or as the set $\{u\}$, depending on the context. Then, the concatenation of $\{u\}$ and $\{v\}$, $\{uv\}$, is written as uv, and the union of $\{u\}$ and $\{v\}$, $\{u, v\}$, is written as $(u + v)$. Expressions such as u^* or $(u + v)^*$ are therefore used to denote sets that may otherwise be too cumbersome to write down by enumerating their elements. These are what we refer to as regular expressions, which we can now define recursively as follows:

1. $\emptyset$ is a regular expression denoting the empty set, ε is a regular expression denoting the set $\{\varepsilon\}$, and e is a regular expression denoting the set $\{e\}$, for all $e \in E$.

2. If r and s are regular expressions, then rs, $(r + s)$, r^*, s^* are regular expressions.

3. There are no regular expressions other than those constructed by applying rules 1 and 2 above a finite number of times.

Regular expressions provide a compact *finite* representation for potentially complex languages with an infinite number of strings.

Example 2.22 (Regular expressions)
Let $E = \{a, b, g\}$ be the set of events. The regular expression $(a + b)g^*$ denotes the language

$$L = \{a, b, ag, bg, agg, bgg, aggg, bggg, \ldots\}$$

which consists of all strings that start with either event a or event b and are followed by a repetition of event g. Note that even though L contains infinitely many elements, the corresponding regular expression provides a simple finite representation of L.

The regular expression $(ab)^* + g$ denotes the language

$$L = \{\varepsilon, g, ab, abab, ababab, \ldots\}$$

which consists of the empty string, event g, and repetitions of the string ab any number of times.

The following theorem is an important result in automata theory. We shall state it without proof.

Theorem. (Regular expressions and regular languages)
Any language that can be denoted by a regular expression is a regular language; conversely, any regular language can be denoted by a regular expression. ♦

This theorem establishes the equivalence of regular expressions and finite-state automata in the representation of languages. This equivalence is known as Kleene's Theorem after S.C. Kleene who proved it in the 1950's. There are algorithmic techniques for converting a given regular expression into an automaton that marks the corresponding language, and vice-versa.

The focus of this and the next chapter is on the representation and manipulation of languages using automata. However, we will often use regular expressions to define languages.

2.4.3 State Space Minimization

In order to simplify the discussion in this section, we assume that the *automata under consideration have completely defined transition functions.* In other words, we are only concerned with the language *marked* by an automaton G and we assume that $\mathcal{L}(G) = E^*$ where E is the event set of G.

There is no unique way to build an automaton G that marks a given language $K \subseteq E^*$. For $K \in \mathcal{R}$, define $||K||$ to be the minimum of $|X|$ (the cardinality of X) among all deterministic finite-state automata G that mark K. The automaton that achieves this minimum is called the *canonical recognizer* of K. Under the above condition $\mathcal{L}(G) = E^*$, the canonical recognizer of a language is unique, up to a renaming of the states. For example, $||\emptyset|| = ||E^*|| = 1$: in both cases, we have a single state with a self-loop for all events in E; this state is unmarked in the case of $\emptyset$ and marked in the case of E^*. If $E = \{a, b\}$ and $L = \{a\}^*$, then $||L|| = 2$: The first state is marked and has a self-loop for event a, while event b causes a transition to the second (unmarked) state, which has self-loops for a and b. It should be emphasized that $|| \cdot ||$ is not related to the cardinality of the language. For instance, $||E^*|| = 1$ yet E^* is an infinite set. Thus a larger language may be (but need not be) representable with fewer states than a smaller language.

The usefulness of the canonical recognizer is that it provides a representation for a (regular) language that essentially minimizes the amount of memory required (in the automaton modeling formalism). We write "essentially" in the preceding sentence because the storage required is proportional to the number of transitions in the state transition diagram of the automaton. However, since we are dealing with deterministic automata, the active event set at any state of an automaton never exceeds $|E|$. In practical applications, the number of events is almost always much less than the number of states; recall our discussion about computational complexity at the end of Sect. 2.3.2. For this reason, it is customary to express the computational complexity of various manipulations of automata in terms of the cardinalities of the state spaces of the automata involved.

Obtaining the canonical recognizer for the language marked by a given automaton G means that we should look for states that are redundant, or *equivalent*, in the sense that they can be replaced by a single "aggregate" state. The notion of equivalence that matters here is language equivalence:

$$x \text{ and } y \text{ are equivalent states in } G \text{ if } \mathcal{L}_m(G(x)) = \mathcal{L}_m(G(y))$$

Recall that $G(x)$ denotes the same automaton as G with the only difference that the initial state is state x. In other words, two states are equivalent if they have the same future

behavior in terms of marked language; of course, two equivalent states also have the same future generated language, E^*, by our assumption of complete transition function. This notion of state equivalence is clearly transitive. Thus, two or more equivalent states can be "merged", that is, replaced by a single aggregate state.

There are some simple observations we can immediately make regarding the above definition of equivalence.

1. If we consider two states x, y such that $x \in X_m$ and $y \notin X_m$, then they can never be equivalent. For instance, $f(x, \varepsilon) \in X_m$, whereas $f(y, \varepsilon) \notin X_m$.

2. Suppose two states x, y are either both in X_m or neither one is in X_m. Therefore, they are eligible to be equivalent. If $f(x, e) = f(y, e)$ for any event $e \in E \cup \{\varepsilon\}$, then states x and y are always equivalent, since they both go through the exact same state sequence for any string applied to them.

3. The preceding property still holds if $f(x, e) = y$ and $f(y, e) = x$ for one or more events e, since the role of x and y is simply interchanged after e. Of course, for all remaining events e we must still have $f(x, e) = f(y, e)$.

4. More generally, let R be a set of states such that either $R \subseteq X_m$ or $R \cap X_m = \emptyset$. Then, R consists of equivalent states if $f(x, e) = z \notin R$ implies that $f(y, e) = z$, for any x, $y \in R$. In other words, all possible transitions from states in R to states outside R must be caused by the same event and must lead to the same next state.

Example 2.23 (A digit sequence detector)
In this example, we design an automaton model for a machine that reads digits from the set $\{1, 2, 3\}$ and detects (that is, marks) any string that ends with the substring 123. The event set is $E = \{1, 2, 3\}$, where event n means "the machine just read digit n," with $n = 1, 2, 3$. The automaton should generate the language E^*, since it should accept any input digit at any time, and it should mark the language

$$L = \{st \in E^* : s \in E^* \text{ and } t = 123\}$$

Let us use simple logic to determine a state space X and build the corresponding transition function f. Let us begin by defining state x_0, the initial state, to represent that no digit has been read thus far. Since we are interested in the substring 123, let us also define states x_1 and x_{not1} to represent that the first event is 1, for x_1, or 2 or 3, for x_{not1}. Given the form of L, we can see that we need the automaton to "memorize" that the suffix of the string read thus far is either 1, 12, 123, or none of these. Therefore, the role of state x_1 is extended to represent that the most recent event is 1, corresponding to the suffix 1. In addition to x_1, we also need states to memorize suffixes 12 and 123; we call these states x_{12} and x_{123}, respectively. Finally, we know that any suffix other than 1, 12, and 123 need not be memorized. Thus, events that result in a suffix that is not 1, 12, or 123 will be sent to state x_{not1}, meaning that this state will represent that 1, 12, and 123 are not suffixes of the string read thus far.

Given $X = \{x_0, x_1, x_{not1}, x_{12}, x_{123}\}$, we construct the correct (completely defined) f as follows:

- From x_0: $f(x_0, 1) = x_1$ and $f(x_0, 2) = f(x_0, 3) = x_{not1}$, consistent with the above-described meaning of these three states.

- From x_1: $f(x_1, 1) = x_1$, $f(x_1, 2) = x_2$, and $f(x_1, 3) = x_{not1}$, in order for the automaton to be in the state corresponding to the current suffix, namely 1, 12, or none of 1, 12 and 123.
- From x_{12}: $f(x_{12}, 1) = x_1$, $f(x_{12}, 2) = x_{not1}$, and $f(x_{12}, 3) = x_{123}$, by the same reasoning as above; if suffix 12 is "broken" by a 1 event, then we have to return to x_1, whereas if it is broken by a 2 event, we must go to x_{not1}.
- From x_{123}: $f(x_{123}, 1) = x_1$, $f(x_{123}, 2) = x_{not1}$, and $f(x_{123}, 3) = x_{not1}$.
- From x_{not1}: $f(x_{not1}, 1) = x_1$, $f(x_{not1}, 2) = f(x_{not1}, 3) = x_{not1}$, by definition of these states.

These state transitions are shown in Fig. 2.25; the initial state is x_0 and the only marked state is x_{123}, since it is reached by any string that ends with the suffix 123. Consequently, the automaton in Fig. 2.25 marks L, as desired.

However, a simpler automaton, with four states instead of five, can be built to mark L (and generate E^*). This automaton is also shown in Fig. 2.25. It is obtained by applying the equivalence condition defined at the beginning of this section to the set of states $R = \{x_0, x_{not1}\}$. Specifically, when event 1 occurs we have $f(x_0, 1) = f(x_{not1}, 1) = x_1$. Moreover, $f(x_0, 2) = f(x_{not1}, 2) = x_{not1}$, and $f(x_0, 3) = f(x_{not1}, 3) = x_{not1}$. Thus, states x_0 and x_{not1} have indistinguishable future behaviors, which means that they can be merged into a single state, denoted by $x_{0,new}$ in the state transition diagram of the second automaton in Fig. 2.25. It is easily verified that there are no equivalent states in the four-state automaton and therefore this automaton is the canonical recognizer of L and $||L|| = 4$.

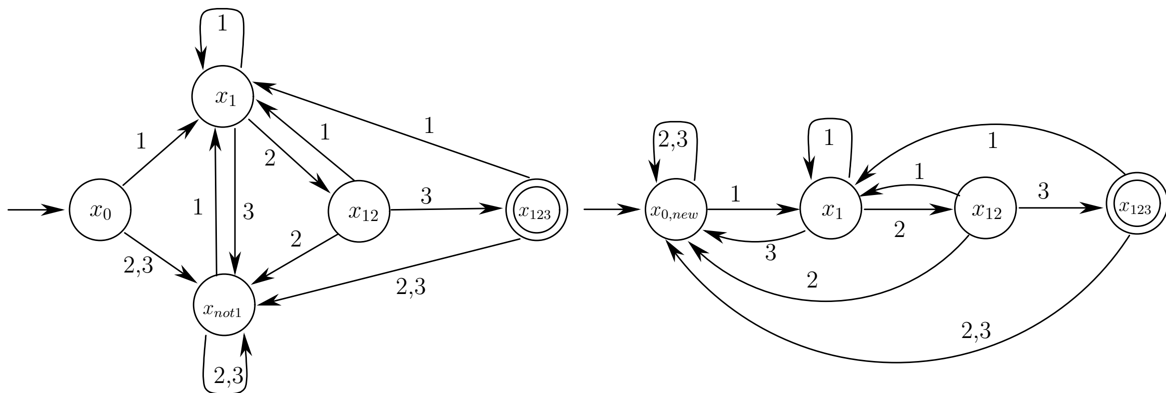

Figure 2.25: State transition diagrams for digit sequence detector in Example 2.23.
Both automata detect the event sequence 123 *upon entering state* x_{123}. *However, in the first model (left), states* x_0 *and* x_{not1} *are equivalent, since* $f(x_0, e) = f(x_{not1}, e)$ *for any* $e \in \{1, 2, 3\}$. *These states are aggregated into a single state* x_0 *to give the second simpler model (right).*

The process of replacing a set of states by a single one is also referred to as state aggregation. This is an important process in many aspects of system theory, because it results in state space reduction, which in turn translates into a decrease of computational complexity for many analytical problems, as we mentioned earlier. Furthermore, *approximate* state aggregation is sometimes a good way to decrease complexity at the expense of some loss of information.

Obviously, it is of interest to develop a general procedure that can always give us the canonical recognizer. Let us assume that we are given automaton $G = (X, E, f, E_G, x_0, X_m)$ where $\mathcal{L}_m(G) = L$ and $\mathcal{L}(G) = E^*$. We want to build from G the canonical recognizer

$$G_{can} = (X_{can}, E, f_{can}, \Gamma_{can}, x_{0,can}, X_{m,can})$$

where $\mathcal{L}_m(G_{can}) = L$, $\mathcal{L}(G_{can}) = E^*$, and $|X_{can}|$ is minimum among all other automata that mark L and generate E^*. The algorithm below allows us to identify all sets of equivalent states. Once these sets are identified, we can form an aggregate state for each one, and thereby obtain the canonical recognizer. The state space X_{can} is the smallest set of states after aggregation, f_{can} is the resulting transition function after aggregation, $x_{0,can}$ is the state in X_{can} that contains x_0, and $X_{m,can}$ is the subset of X_{can} containing marked states; since equivalent states must share the same marking property (either marked or unmarked), there is no ambiguity in defining $X_{m,can}$.

The main idea in the algorithm is to begin by "flagging" all pairs of states (x, y) such that x is a marked state and y is not. Such flagged pairs cannot be equivalent. Here, of course, we are assuming that there is at least one state y that is not marked, that is, $X_m \neq X$. Next, every remaining pair of states (x, y) is considered, and we check whether some event $e \in E$ may lead to a new pair of states $(f(x, e), f(y, e))$ which is already flagged. If this is not the case, and $f(x, e) \neq f(y, e)$, then we create a list of states associated with $(f(x, e), f(y, e))$ and place (x, y) in it. As the process goes on, we will eventually determine if $(f(x, e), f(y, e))$ should be flagged; if it is, then the flagging propagates to (x, y) and all other pairs in its list.

Algorithm for Identifying Equivalent States

Step 1: Flag (x, y) for all $x \in X_m$, $y \notin X_m$.

Step 2: For every pair (x, y) not flagged in Step 1:

Step 2.1: If $(f(x, e), f(y, e))$ is flagged for some $e \in E$, then:

Step 2.1.1: Flag (x, y).

Step 2.1.2: Flag all unflagged pairs (w, z) in the list of (x, y). Then, repeat this step for each (w, z) until no more flagging is possible.

Step 2.2: Otherwise, that is, no $(f(x, e), f(y, e))$ is flagged, then for every $e \in E$:

Step 2.2.1: If $f(x, e) \neq f(y, e)$, then add (x, y) to the list of $(f(x, e), f(y, e))$.

At the end of this procedure, we examine all pairs that have remained unflagged. These pairs correspond to states that are equivalent. If two pairs have an element in common, we group them together into a set, and so forth, since equivalence is transitive. At the end, we get disjoint sets of equivalent states; each set is associated with a single aggregate state in the canonical recognizer.

The algorithm is easier to visualize through a table containing all possible state pairs, where an entry (x, y) is flagged to indicate that x and y are not equivalent. At the end of the algorithm, equivalent state pairs correspond to entries in the table that are not flagged. As an example, the table corresponding to the automaton of Fig. 2.25 before state aggregation is shown in Fig. 2.26. We begin by flagging (indicated by an "F" in the table) all entries

in the first column, corresponding to pairs consisting of the marked state x_{123} and the remaining four states. Then, proceeding by column and applying the algorithm above, we obtain one pair of equivalent states, (x_0, x_{not1}), which are aggregated as in Fig. 2.25. More precisely, the step by step application of the algorithm proceeds as follows:

- Flag all first column entries.
- Flag (x_{12}, x_0), because $f(x_{12}, 3) = x_{123}$, $f(x_0, 3) = x_{not1}$, and (x_{123}, x_{not1}) is already flagged.
- Flag (x_{12}, x_1), because $f(x_{12}, 3) = x_{123}$, $f(x_1, 3) = x_{not1}$, and (x_{123}, x_{not1}) is already flagged.
- Flag (x_{12}, x_{not1}), because $f(x_{12}, 3) = x_{123}$, $f(x_{not1}, 3) = x_{not1}$, and (x_{123}, x_{not1}) is already flagged.
- Do not flag (x_{not1}, x_0), since $f(x_{not1}, e) = f(x_0, e)$ for all $e = 1, 2, 3$.
- Flag (x_{not1}, x_1), because $f(x_{not1}, 2) = x_{not1}$, $f(x_1, 2) = x_{12}$, and (x_{12}, x_{not1}) is already flagged.
- Flag (x_1, x_0), because $f(x_1, 2) = x_{12}$, $f(x_0, 2) = x_{not1}$, and (x_{12}, x_{not1}) is already flagged.

	x_{123}	x_{12}	x_{not1}	x_1
x_0	F	F		F
x_1	F	F	F	
x_{not1}	F	F		
x_{12}	F			

Figure 2.26: Identifying equivalent states in the automaton of Fig. 2.25. Flagged state pairs are identified by the letter F.

2.5 ANALYSIS OF DISCRETE-EVENT SYSTEMS

One of the key reasons for using finite-state automata to model DES is their amenability to analysis for answering various questions about the behavior of the system.

We discuss in the next three subsections the most-often encountered analysis problems for DES. Sect. 2.5.1 considers safety and blocking properties of deterministic automata where all events are observable. Sect. 2.5.2 considers the case of partially-observed DES, where some of the events are "unobservable". An unobservable event is either an ε-transition or some