# CHAPTER 2

# LANGUAGES AND AUTOMATA

## 1 Languages as a way to formally describe events

When considering the state evolution of a Discrete Event System (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 vent sequences of the form $e_1\ e_2 \ldots e_n$. 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 the previous chapter. The behavior of the system is the modeled by a language and 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.

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 and see how we can build a simple DES to perform a "status check" task.

## 1.1 Language Notations 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 $\varepsilon$. (The symbol $\varepsilon$ 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 $\varepsilon$ is taken to be zero.

**Definition 1.1.** A *language* defined over an event set $E$ is a set of finite-length strings formed from events in $E$.

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

$$L_1 = \{\varepsilon,\ b,\ ac,\ cbc\}$$

consisting of four strings only; or the language

$$L_2 = \{\text{all the 4-length strings starting with } a \text{ and ending with } c\}$$

or

$$L_3 = \{\text{all the finite length strings starting with } c \text{ and ending with } c\}$$

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 $cbc$ in $L_1$ above is the concatenation of the string $cb$ with the event (or string of length one) $c$; $cb$ is itself the concatenation of $c$ 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 $\varepsilon$ 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 $\varepsilon$; 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 $E = \{a,\ c\}$, then

$$E^* = \{\varepsilon,\ a,\ c,\ aa,\ ac,\ cc,\ ca,\ aaa,\ aac, \dots\} \tag{2.1}$$

A language over an event set $E$ is therefore a subset of $E^*$. In particular, $\varnothing$, $E$, 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 a *prefix* of $s$;
- $u$ is a *substring* of $s$;
- $v$ is 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 $\varepsilon$ and $s$ are prefixes (substrings, suffixes) of $s$.

## 1.2 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:

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

$$L_aL_b := \{s \in E^* \ : \ (s = s_as_b),\ (s_a \in L_a),\ (s_b \in L_b)\} \tag{2.2}$$

  In words, a string is in $L_aL_b$ 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

$$\overline{L} := \{s \in E^* : (\exists t \in E^*),\ (st \in L)\} \tag{2.3}$$

  In words, the prefix closure of $L$ is the language denoted by $L$ and consisting of all the prefixes of all the strings in $L$. In general, $\overline{L} \subseteq L$. $L$ is said to be *prefix-closed* if $\overline{L} = L$. 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

$$L^* := \{\varepsilon\} \cup L \cup LL \cup LLL \cup \dots \tag{2.4}$$

  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 $\varepsilon$. Note that the $*$ operation is *idempotent*: $(L^*)^* = L^*$.

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

$$L/s := \{t \in E^* : st \in L\} \tag{2.5}$$

  By definition, $L/s = \varnothing$ if $s \notin L$.

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 $L^*$). We make the following observations for technical accuracy:

1. $\varepsilon \notin \varnothing$;
2. $\{\varepsilon\}$ is a nonempty language containing only the empty string;
3. If $L = \varnothing$ then $\overline{L} = \varnothing$ and if $L \neq \varnothing$ the necessarily $\varepsilon \in L$;
4. $\varnothing^* = \{\varepsilon\}$ and $\{\varepsilon\}^* = \{\varepsilon\}$;
5. $\varnothing L = L\varnothing = \varnothing$

## 1.3 Projection Operators

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:

$$P \ : \ E_l^* \to E_s^* \tag{2.6}$$

with

1. $P(\varepsilon) = \varepsilon$
2. $P(e) = \begin{cases} e & \text{if } e \in E_s \\ \varepsilon & \text{if } e \in E_l/E_s \end{cases}$
3. $P(se) = P(s)P(e)$, $s \in E_l^*$, $e \in E_l$

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:

$$P^{-1} \ : \ E_s^* \to 2^{E_l^*} \tag{2.7}$$

defined as follows

$$P^{-1}(t) = \{s \in E_l^* \ : \ P(s) = t, \ t \in E_s^*\} \tag{2.8}$$

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 $L \subseteq E_l^*$

$$P(L) = \{t \in E_s^* \ : \ \exists s \in L, P(s) = t\} \tag{2.9}$$

and for $L_s \subseteq E_s^*$

$$P^{-1}(L_s) = \{s \in E_l^* \ : \ \exists t \in L_s, P(s) = t\} \tag{2.10}$$

## 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 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 2.1.** Let the event set be $E = \{a,\ b,\ g\}$. Consider the state transition diagram in Figure 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 $f\ :\ X \times E \to X$:

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

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

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. △

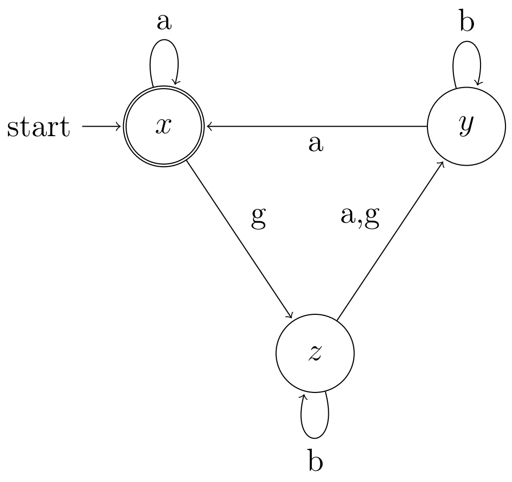


**Figure 2.1:** Automaton

Three observations are worth making regarding the Example. 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. 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.

**Definition 2.1.** A *Deterministic Automaton*, denoted by $G$, is a six-tuple

$$G = (X,\, E,\, f,\, \Gamma,\, x_0,\, X_m) \tag{2.11}$$

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 $\Gamma$ 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 non-deterministic 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. We will return to non-deterministic automata later.

- 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 $\Gamma$ in the definition of $G$ is superfluous in the sense that $\Gamma$ is derived from $f$. For this reason, we will sometimes omit explicitly writing $\Gamma$ 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. In these notes, 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:

$$f(x, \varepsilon) = x$$

$$f(x, se) = f(f(x, s), e), \text{ for } s \in E^* \text{ and } e \in E$$

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.

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 the notes work for infinite-state automata. Of course, explicit representations of infinite-state automata would require infinite memory. Finite-state automata will be discussed when regular expressions and languages notions will be introduced.

## 2.1 Languages generated 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 2.2.** 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

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

The above definitions assume that we are working with the extended transition function $f : X \times E^* \to X$. 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 2.2.** 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 $G = (X,\, E,\, f,\, \Gamma,\, x_0,\, X_m)$ 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 Figure . 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 $\mathcal{L}(G) = E^*$.
$\triangle$

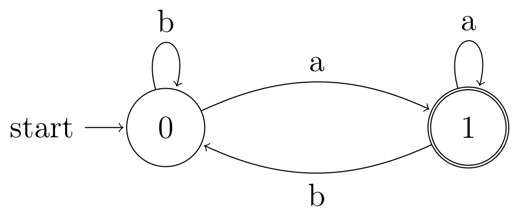


**Figure 2.2:** Automaton of Example 2.2

**Example 2.3.** If we modify the automaton in Example 2.2 by removing the self-loop due to event $b$ at state 0 in Fig. 2.2, that is, by letting $f(0, b)$ undefined, then $\mathcal{L}(G)$ now consists of $\varepsilon$ 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 $a$. $\triangle$

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 $E^*$.

## 2.2 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 2.3.** Automata $G_1$ and $G_2$ are *language equivalent* if

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

**Example 2.4.** Let us consider the following two automata

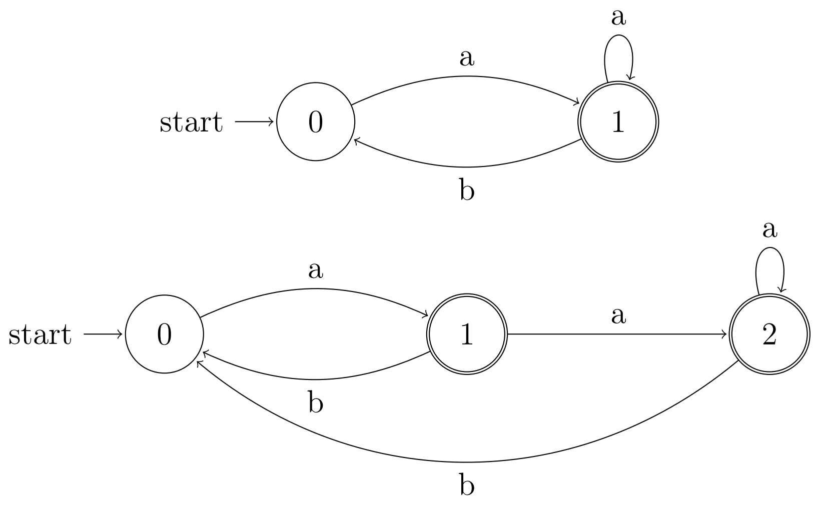


**Figure 2.3:** Language equivalent automata

The two automata shown in Figure 2.3 are language-equivalent, as they all generate the same language and they all mark the same language. △

## 2.3 Blocking

In general we have

$$\mathcal{L}_m(G) \subseteq \overline{\mathcal{L}_m(G)} \subseteq \mathcal{L}(G) \tag{2.12}$$

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 $\Gamma(x) = \varnothing$ 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 $\mathcal{L}_m(G)$ 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 $\mathcal{L}_m(G)$ 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 2.4.** Automaton $G$ is said to be blocking if

$$\overline{\mathcal{L}_m(G)} \subset \mathcal{L}(G) \tag{2.13}$$

where the set inclusion is proper and nonblocking when

$$\overline{\mathcal{L}_m(G)} = \mathcal{L}(G) \tag{2.14}$$

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.

## 2.4 Non-Deterministic 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 $\varepsilon$ 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 "$\varepsilon$-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 $\varepsilon$ 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 $\varepsilon$. Of course, if the transition occurs, the "label" $\varepsilon$ is not seen, since $\varepsilon$ 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 non-deterministic automata.

**Definition 2.5.** A *Non-deterministic Automaton*, denoted by $G_{nd}$ , is a six-tuple

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

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

1. $f_{nd}$ is a function $f_{nd} \;:\; X \times (E \cup \{\varepsilon\}) \to 2^X$, 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 $x_0 \subseteq X$.

**Example 2.5.** Consider the finite-state automaton of Fig. 2.4. 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. △

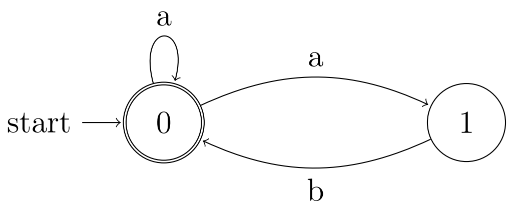


**Figure 2.4:** Non-deterministic automaton of Example 2.5

**Example 2.6.** The automaton in Fig. 2.5 is non-deterministic and includes an $\varepsilon$-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. △

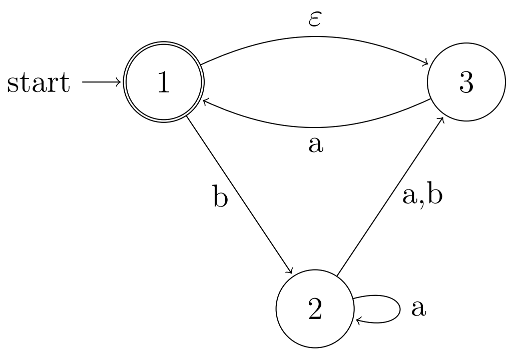


**Figure 2.5:** Non-deterministic automaton of Example 2.6

In order to characterize the strings generated and marked by non-deterministic 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 non-deterministic case, we start by defining the $\varepsilon$-reach of a state $x$ to be the set of all states that can be reached from $x$ by following transitions labeled by $\varepsilon$ 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 $\varepsilon$-transition is not observed. In the case of a set of states $B \subseteq X$,

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

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

$$f_{nd}^{ext}(x, \varepsilon) = \varepsilon R(x) \tag{2.15}$$

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

$$f_{nd}^{ext}(x, ue) = \varepsilon R\left(\left\{z \in X \ : \ z \in f_{nd}(y, e) \text{ for some } y \in f_{nd}^{ext}(x, u)\right\}\right) \tag{2.16}$$

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 $\varepsilon$-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 $\varepsilon$-reach.

As an example, $f_{nd}^{ext}(0, ab) = \{0\}$ in the automaton of Fig. 2.4, 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. 2.5, 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 non-deterministic automata. These are defined as follows

$$\mathcal{L}(G_{nd}) = \left\{s \in E^* \ : \ \exists x \in x_0, \ f_{nd}^{ext}(x, s) \text{ is defined}\right\}$$

$$\mathcal{L}_m(G_{nd}) = \left\{s \in \mathcal{L}(G_{nd}) \ : \ \exists x \in x_0, \ f_{nd}^{ext}(x, s) \cap X_m \neq \varnothing\right\}$$

These definitions mean that a string is in the language generated by the non-deterministic 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. 2.4 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, the primary source of non-determinism 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 a $\varepsilon$-transition from the point of view of an outside observer. In stochastic DES, non-determinism arises as a consequence of the stochastic nature of the model.

### 2.5 Automata with inputs and outputs

There are two variants to the definition of automaton given in the previous subsections 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.)

- *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 above the state. 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. 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 $e_o$.

We can always interpret the behavior of Mealy and Moore automata according to the dynamics of standard automata. 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 here 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. This effectively transforms the Moore automaton into a Mealy automaton, which can then be interpreted as a standard automaton as we just described.

## 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, compositions operations are covered. The second focus of this section is on non-deterministic automata where we will define a special kind of automaton, called observer automaton, that is a deterministic version of a given non-deterministic automaton preserving language equivalence. As in the preceding subsections, our discussion does not require the state set of an automaton to be finite.

## 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,

$$Ac(G) = (X_{ac},\, E,\, f_{ac},\, \Gamma_{ac},\, x_0,\, X_{ac,\,m})$$

where

$$\begin{aligned} X_{ac} &= \{x \in X \ : \ (\exists s \in E^*),\, x = f(x_0,\, s)\} \\ X_{ac,\,m} &= X_m \cap X_{ac} \\ f_{ac} &= f|_{X_{ac} \times E \rightarrow X_{ac}} \end{aligned}$$

The notation $f|_{X_{ac} \times E \rightarrow X_{ac}}$ 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, $G = Ac(G)$.

**Co-Accessible 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.

$$CoAc(G) = (X_{coac},\, E,\, f_{coac},\, \Gamma_{coac},\, x_{0,\,coac},\, X_m)$$

where

$$\begin{aligned} X_{coac} &= \{x \in X \ : \ (\exists s \in E^*),\, f(x,\, s)\} \\ x_{0,\,coac} &= \begin{cases} x_0 & \text{if } x_0 \in X_{coac} \\ \text{undefined} & \text{otherwise} \end{cases} \\ f_{coac} &= f|_{X_{coac} \times E \rightarrow X_{coac}} \end{aligned}$$

The $CoAc$ operation may shrink $\mathcal{L}(G)$, since we may be deleting states that are accessible from x0; 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 $X_{coac} = \varnothing$ (this would happen if $X_m = \varnothing$ for instance), then we obtain the empty automaton

### Trim

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 $E_s \subseteq E$. 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 $E \backslash E_s$ by $\varepsilon$. The result is a non-deterministic automaton that generates and marks the desired languages. 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 $E_l \supseteq E_s$. 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 $E_l \backslash E_s$ at all the states of $G$.

### 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 $\overline{L}$ (prefix). We wish to build another automaton, denoted by $G_{comp}$, that will mark the language $E^* \backslash L$. $G_{comp}$ 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

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

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

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

Clearly if $G_{comp} = Comp(G)$ then $\mathcal{L}(G_{comp}) = E^*$ and $\mathcal{L}_m(G_{comp}) = E^* \backslash L$.

## 3.2 Composition Operators

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) non-deterministic automata can be composed using the same rules for the joint transition function.

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

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

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

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

where

$$f_{1\times 2}((x_1,\, x_2),\, e) = \begin{cases} (f(x_1,\, e),\, f(x_2,\, e)) & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

and, $\Gamma_{1\times 2}(x_1,\, x_2) = \Gamma_1(x_1) \cap \Gamma_2(x_2)$. 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

$$\mathcal{L}\left(G_1 \times G_2\right) = \mathcal{L}\left(G_1\right) \cap \mathcal{L}\left(G_2\right)$$

$$\mathcal{L}_m\left(G_1 \times G_2\right) = \mathcal{L}_m\left(G_1\right) \cap \mathcal{L}_m\left(G_2\right)$$

This shows that the intersection of two languages can be "implemented" by doing the product of their automaton representations – an important result. If $E_1 \cap E_2 = \varnothing$, then $\mathcal{L}\left(G_1 \times G_2\right) = \{\varepsilon\}$; $\mathcal{L}_m\left(G_1 \times G_2\right)$ will be either $\varnothing$ 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 $G_1 G_2$ 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 $f$.

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

2. Product is associative, $G_1 \times G_2 \times G_3 = (G_1 \times G_2) \times G_3 = G_1 \times (G_2 \times G_3)$

**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. The *parallel* of $G_1$ and $G_2$ is the automaton

$$G_1 \parallel G_2 = Ac\left(X_1 \times X_2,\, E_1 \cup E_2,\, f_{1\|2},\, \Gamma_{1\|2},\, (x_{01},\ x_{02})\,,\, X_{m1} \times X_{m2}\right)$$

where

$$f_{1\|2}((x_1,\, x_2),\, e) = \begin{cases} (f(x_1,\, e),\, f(x_2,\, e)) & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f(x_1,\, e),\, x_2) & \text{if } e \in \Gamma_1(x_1) \backslash E_2 \\ (x_1,\, f(x_2,\, e)) & \text{if } e \in \Gamma_2(x_2) \backslash E_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and,

$$\Gamma_{1\|2}(x_1,\, x_2) = (\Gamma_1(x_1) \cap \Gamma_2(x_2)) \cup (\Gamma_1(x_1) \backslash E_2) \cup (\Gamma_2(x_2) \backslash E_1)$$

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

$$(E_2 \backslash E_1) \cup (E_1 \backslash E_2)$$

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 $E_1 \cap E_2 = \varnothing$, then there are no synchronized transitions and $G_1 \parallel G_2$ 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 $G_1 \parallel G_2$ in terms of those of $G_1$ and $G_2$, we need to use the operation of language projection. 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

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

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

$$\mathcal{L}(G_1 \parallel G_2) = P_1^{-1}(\mathcal{L}(G_1)) \cup P_2^{-1}(\mathcal{L}(G_2))$$

$$\mathcal{L}_m(G_1 \parallel G_2) = P_1^{-1}(\mathcal{L}_m(G_1)) \cup P_2^{-1}(\mathcal{L}_m(G_2))$$

### State Spate 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 $x$ 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 $G_{1,new}$.

### 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 $L_1 \subseteq L_2$:

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

## 3.3 Observer Automata

We introduced earlier the class of non-deterministic 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 $\varepsilon$-transitions. The following question then arises: How do deterministic and non-deterministic automata compare in terms of language representation?

**Example 3.1.** Let us reconsider the non-deterministic automaton

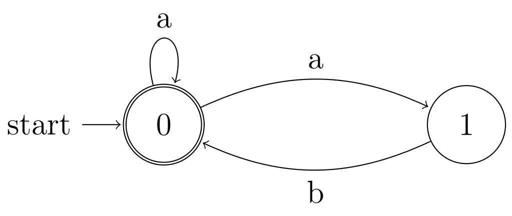


**Figure 2.6:** Non-deterministic automaton of Example 2.5

It is easily verified that the deterministic automaton (let us call it $G$) depicted in Fig. is equivalent to the non-deterministic one in Fig. 2.6 (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:

1. $f(A, a) = B$ and $f_{nd}(0, a) = \{0, 1\}$;
2. $f(A, b)$ and $f_{nd}(0, b)$ are undefined;
3. $f(B, a) = B$ and $f_{nd}(0, a) = \{0, 1\}$ with $f_{nd}(1, a)$ undefined; and

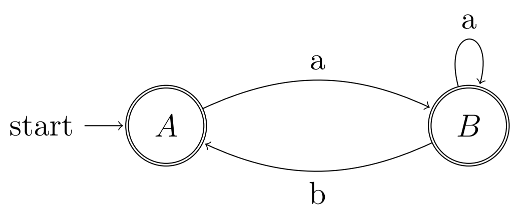

$$\begin{tikzcd}
\text{start} \arrow[r] & A \arrow[rr, bend left, "a"] & & B \arrow[ll, bend left, "b"] \arrow[loop above, "a"]
\end{tikzcd}$$

**Figure 2.7:** Deterministic automaton equivalent to $G_{nd}$

4. $f(B, b) = A$ and $f_{nd}(1, b) = \{0\}$ with $f_{nd}(0, b)$ undefined.

△

It turns out that we can always transform a non-deterministic automaton, $G_{nd}$, into a language-equivalent deterministic one, that is, one that generates and marks the same languages as the original non-deterministic automaton. The state space of the equivalent deterministic automaton will be a subset of the power set of the state space of the non-deterministic one. This means that if the non-deterministic 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 Regular Languages section. The focus of this part is to present an algorithm for this language-preserving transformation from non-deterministic to deterministic automaton. We shall call the resulting equivalent deterministic automaton the observer corresponding to the non-deterministic 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 non-deterministic automaton upon transitions labeled by events in $E$. (Recall that the event set of $G_{nd}$ is $E \cup \{\varepsilon\}$.)

**Example 3.2.** Let us consider the following non-deterministic automaton

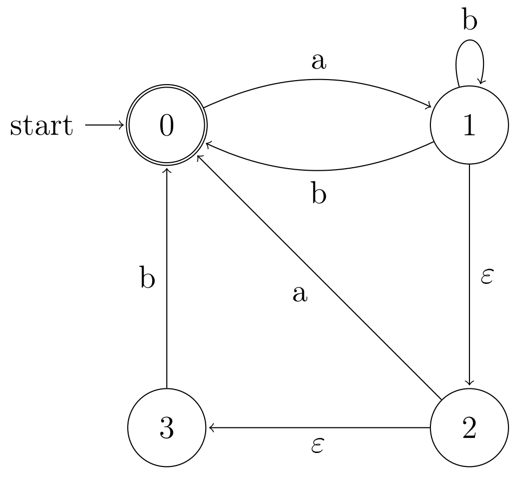


**Figure 2.8:** Non-Deterministic automaton

where nondeterminism arises at states 1 and 2, since event $b$ leads to two different states from state 1 and since we have $\varepsilon$-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 $\varepsilon$.

- 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 $G_{obs}$.

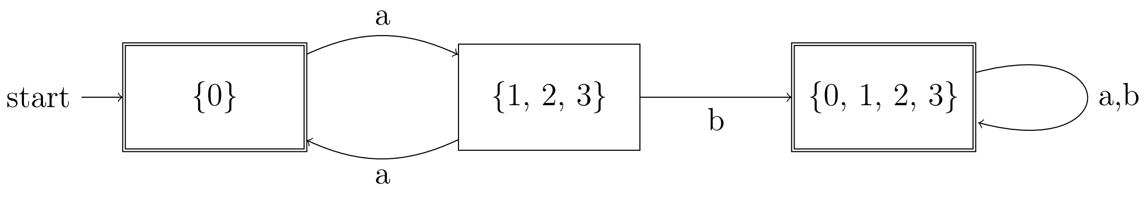
Finally, we analyze the newly-created state $\{0, 1, 2, 3\}$ of $G_{obs}$. Proceeding similarly as above, we identify the following transitions to be added to $G_{obs}$:

- 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. . It can be seen that $G_{nd}$ and $G_{obs}$ are indeed language equivalent. △



With the intuition gained from this example, we present the formal steps of the algorithm to construct the observer. Recall from 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 $\varepsilon$-reach of state $x$:

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

## 3.4 Procedure for Building Observer $Obs(G_{nd})$ of Non-deterministic Automaton $G_{nd}$

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

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

  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: $X_{m,obs} = \{B \subseteq X_{obs} : B \cap X_m = \varnothing\}$.

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 $\varepsilon$-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 $fobs(x_{0,obs},\, e)$, 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 $fobs(x_{0,obs}, e)$, as this set represents all the states where $G_{nd}$ could be after executing the string $e$, preceded and/or followed by $\varepsilon$.

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.