

From Clarke & Grumberg

Book, "Model checking"

MIT Press

describes the interleaved execution of the concurrent processes. The formula for the transition relation of process  $P_i$  is conjuncted with  $same(V \setminus V_i) \wedge same(PC \setminus \{pc_i\})$ . This guarantees that a transition in process  $P_i$  can only change variables in  $V_i$ . It also ensures that only one process can make a transition at any time.

**Shared Variables**

Recall that  $V_i$  is the set of variables that may be changed by process  $P_i$ . Concurrent programs for which the sets  $V_i$  overlap are called *shared variable* programs. We show how to extend the translation procedure  $\mathcal{C}$  to some commonly used *process synchronization* statements. Such statements are frequently needed to provide processes with exclusive access to shared variables. These statements are atomic and treated by the labeling transformation accordingly. Assume that the statement belongs to the text of process  $P_i$ .

- **Wait:** Because our primary interest is in finite state programs, we only describe how to implement this statement using *busy waiting*. In particular, we do not consider implementations that require complex data structures like process queues. The statement  $\mathbf{wait}(b)$  repeatedly tests the value of the boolean variable  $b$  until it determines that  $b$  is true. When  $b$  becomes true, a transition is made to the next program point

$\mathcal{C}(l, \mathbf{wait}(b), l')$  is a disjunction of the following two formulas:

- $(pc_i = l \wedge pc'_i = l \wedge \neg b \wedge same(V_i))$
- $(pc_i = l \wedge pc'_i = l' \wedge b \wedge same(V_i))$

- **Lock:** The statement  $\mathbf{lock}(v)$  is similar to the statement  $\mathbf{wait}(v = 0)$ , except that when  $v = 0$  is true the transition changes the value of  $v$  to 1. This statement is often used to guarantee *mutual exclusion* by preventing more than one process from entering its critical region.

$\mathcal{C}(l, \mathbf{lock}(v), l')$  is a disjunction of the following two formulas:

- $(pc_i = l \wedge pc'_i = l \wedge v = 1 \wedge same(V_i))$
- $(pc_i = l \wedge pc'_i = l' \wedge v = 0 \wedge v' = 1 \wedge same(V_i \setminus \{v\}))$

- **Unlock:** The statement  $\mathbf{unlock}(v)$  assigns the value 0 to the variable  $v$ . Typically, this statement enables some other process to enter its critical region.

$\mathcal{C}(l, \mathbf{unlock}(v), l') \equiv pc_i = l \wedge pc'_i = l' \wedge v' = 0 \wedge same(V_i \setminus \{v\})$

**2.3 Example of Program Translation**

Consider a simple *mutual exclusion* program

$P = m : \mathbf{cobegin} P_0 \parallel P_1 \mathbf{coend} m'$

with two

$P_0 :: l_0$

$l'_0$

$P_1 :: l_1$

$l'_1$

The program point of  $P$  active. Each  $CR_i$ , and  $PC =$  the process regions at its noncritical into the cr

The initial

$S_0(V, PC$

Note that or 1. Applied of  $P, \mathcal{R}(V$

▪  $pc = m$

▪  $pc_0 = l'_0$

▪  $\mathcal{C}(l_0, P_0,$

$l'_0,$

▪  $\mathcal{C}(l_1, P_1,$

$l'_1$

with two processes  $P_0$  and  $P_1$ , where

```

 $P_0 :: l_0 : \text{ while } True \text{ do}$ 
            $NC_0 : \text{ wait}(turn = 0);$ 
            $CR_0 : turn := 1;$ 
            $\text{ end while;}$ 

```

$l'_0$

```

 $P_1 :: l_1 : \text{ while } True \text{ do}$ 
            $NC_1 : \text{ wait}(turn = 1);$ 
            $CR_1 : turn := 0;$ 
            $\text{ end while;}$ 

```

$l'_1$

The program counter  $pc$  of the program  $P$  takes only three values:  $m$ , the label of the entry point of  $P$ ;  $m'$ , the label of the exit point of  $P$ ; and  $\perp$  the value of  $pc$  when  $P_0$  and  $P_1$  are active. Each process  $P_i$  has a program counter  $pc_i$  that ranges over the labels  $l_i$ ,  $l'_i$ ,  $NC_i$ ,  $CR_i$ , and  $\perp$ . The two processes share a single variable  $turn$ . Thus,  $V = V_0 = V_1 = \{turn\}$  and  $PC = \{pc, pc_0, pc_1\}$ . When the value of the program counter of a process  $P_i$  is  $CR_i$ , the process is in its *critical region*. Both processes are not allowed to be in their critical regions at the same time. When the value of the program counter is  $NC_i$ , the process is in its *noncritical region*. In this case it waits until  $turn = i$  in order to gain exclusive entry into the critical region.

The initial states of  $P$  are described by the formula

$$\mathcal{S}_0(V, PC) \equiv pc = m \wedge pc_0 = \perp \wedge pc_1 = \perp.$$

Note that no restriction is imposed on the value of  $turn$ . Thus, it may initially be either 0 or 1. Applying the translation procedure  $\mathcal{C}$  we obtain the formula for the transition relation of  $P$ ,  $\mathcal{R}(V, PC, V', PC')$ , which is the disjunction of the following four formulas:

- $pc = m \wedge pc'_0 = l_0 \wedge pc'_1 = l_1 \wedge pc' = \perp$
- $pc_0 = l'_0 \wedge pc_1 = l'_1 \wedge pc' = m' \wedge pc'_0 = \perp \wedge pc'_1 = \perp$
- $\mathcal{C}(l_0, P_0, l'_0) \wedge same(V \setminus V_0) \wedge same(PC \setminus \{pc_0\})$ , which is equivalent to

$$\mathcal{C}(l_0, P_0, l'_0) \wedge same(pc, pc_1)$$

- $\mathcal{C}(l_1, P_1, l'_1) \wedge same(V \setminus V_1) \wedge same(PC \setminus \{pc_1\})$ , which is equivalent to

$$\mathcal{C}(l_1, P_1, l'_1) \wedge same(pc, pc_0)$$

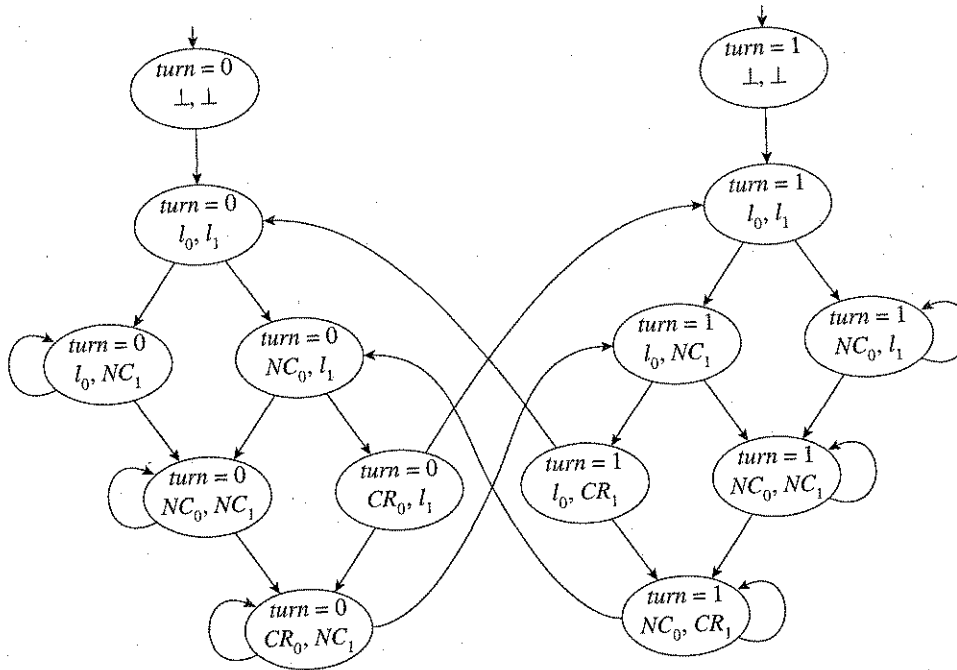


Figure 2.2  
Reachable states of Kripke structure for mutual exclusion example.

For each process  $P_i$ ,  $C(l_i, P_i, l'_i)$  is the disjunction of:

- $pc_i = l_i \wedge pc'_i = NC_i \wedge True \wedge same(turn)$
- $pc_i = NC_i \wedge pc'_i = CR_i \wedge turn = i \wedge same(turn)$
- $pc_i = CR_i \wedge pc'_i = l_i \wedge turn' = (i + 1) \bmod 2$
- $pc_i = NC_i \wedge pc'_i = NC_i \wedge turn \neq i \wedge same(turn)$
- $pc_i = l_i \wedge pc'_i = l'_i \wedge False \wedge same(turn)$

The Kripke structure in Figure 2.2 is derived from the formulas  $S_0$  and  $\mathcal{R}$  as described in Section 2.1.1. By examining the state space of the Kripke structure, it is easy to see that the processes will never be in their critical regions at the same time. Thus, the program guarantees the required mutual exclusion property. However, this program fails to guarantee *absence of starvation*, since one of the processes may continuously try to enter its critical region without ever being able to do so, while the other process stays in its critical region forever. Later, we will see how to formulate and model check such properties.

In this chapter we discuss temporal logic semantics for systems or Kripke structures and boolean connected expressions in describing the interaction with and control methodologies, such as output semantics are not reflected in the start of execution. In systems, the computation is designed not to terminate. *Temporal logic* reactive system. It is not; instead, a formula for that an error state is a special *temporal* formula or nested assertions semantics of those

### 3.1 The Compu

Conceptually, CTL\* is defined by designating a Kripke structure into an interpretation. The computation is a path through the structure.

In CTL\* formulae, temporal quantifiers are used to express two such quantifiers. These quantifiers are used to express paths starting at the root of a path through the structure.

- **X** ("next time")
- **F** ("eventually")
- **G** ("always" or "globally")

## 9 Model Checking and Automata Theory

In this chapter we present some basic facts from automata theory and demonstrate how model checking can be performed in this framework. In particular, we show how to translate an LTL formula into an automaton. This gives an alternative model checking algorithm for LTL, which can be performed on the fly. In this approach the checked property guides the construction of the state graph for the modeled system. Consequently, it may be possible to avoid constructing large parts of the state graph.

### 9.1 Automata on Finite and Infinite Words

A finite automaton is a mathematical model of a device that has a constant amount of memory, independent of the size of its input. We will consider finite automata over finite words and finite automata over infinite words (also called  $\omega$ -automata).

Formally, a finite automaton (over finite words)  $\mathcal{A}$  is a five tuple  $\langle \Sigma, Q, \Delta, Q^0, F \rangle$  such that

- $\Sigma$  is the finite *alphabet*.
- $Q$  is the finite set of *states*.
- $\Delta \subseteq Q \times \Sigma \times Q$  is the *transition relation*.
- $Q^0 \subseteq Q$  is the set of *initial states*.
- $F \subseteq Q$  is the set of *final states*.

An automaton can be represented as a graph with labeled transitions, in which the set of nodes is  $Q$  and the edges are given by  $\Delta$ . An example of an automaton is shown in Figure 9.1. There,  $\Sigma = \{a, b\}$ ,  $Q = \{q_1, q_2\}$ ,  $Q^0 = \{q_1\}$  (initial states are marked with an incoming arrow), and  $F = \{q_1\}$  (accepting states are marked with a double circle).

Let  $v$  be a word (string, sequence) of  $\Sigma^*$  of length  $|v|$ . A *run* of  $\mathcal{A}$  over  $v$  is a mapping  $\rho : \{0, 1, \dots, |v|\} \mapsto Q$  such that:

- The first state is an initial state, that is,  $\rho(0) \in Q^0$ .
- Moving from the  $i$ th state  $\rho(i)$  to the  $i + 1$ st state  $\rho(i + 1)$  upon reading the  $i$ th input letter  $v(i)$  is consistent with the transition relation. That is, for  $0 \leq i < |v|$  ( $\rho(i), v(i), \rho(i + 1) \in \Delta$ ).

A run  $\rho$  of  $\mathcal{A}$  on  $v$  corresponds to a path in the automaton graph from an initial state  $\rho(0)$  to a state  $\rho(|v|)$ , where the edges on this path are labeled according to the letters in  $v$ . We say that  $v$  is an *input* to the automaton  $\mathcal{A}$  or that  $\mathcal{A}$  *reads*  $v$ . A run  $\rho$  over  $v$  is *accepting* if it ends in an accepting state, that is,  $\rho(|v|) \in F$ . An automaton  $\mathcal{A}$  *accepts* a

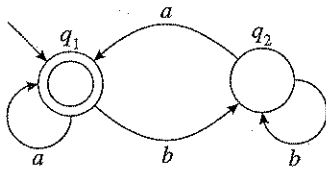


Figure 9.1  
A finite automaton.

word  $v$  if and only if there exists an accepting run of  $\mathcal{A}$  on  $v$ . For example, the automaton in Figure 9.1 accepts the word  $aabba$  because there is a run that passes through the states  $q_1q_1q_1q_2q_2q_1$ .

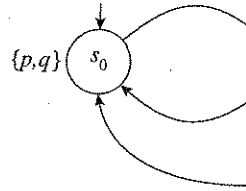
The language of  $\mathcal{A}$ ,  $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$  consists of all the words accepted by  $\mathcal{A}$ . The automaton in Figure 9.1 accepts the language described by the regular expression  $\epsilon + (a + b)^*a$ , that is, either the empty word  $\epsilon$ , or words that consist of any number of  $a$ 's or  $b$ 's and end with an  $a$ . The operator  $+$  indicates a choice, and the  $*$  operator indicates any finite number of repetitions.

Because most concurrent systems are designed not to halt during normal execution, we model computations as infinite sequences of states. Thus, this chapter will focus on finite automata over infinite words. These automata have the same structure as finite automata over finite words. However, they recognize words from  $\Sigma^\omega$ , where the superscript  $\omega$  indicates an infinite number of repetitions.

The simplest automata over infinite words are Büchi [39] automata. A Büchi automaton has the same components as an automaton over finite words. However,  $F$  is called the set of *accepting states*, rather than final states. A run of a Büchi automaton  $\mathcal{A}$  over an infinite word  $v \in \Sigma^\omega$  is defined in almost the same way as a run of a finite automaton over a finite word, except that now  $|v| = \omega$ . Thus, the domain of a run is the set of all natural numbers. Again, a run corresponds to a path in the graph of the automaton, but the path is now an infinite one.

Let  $\text{inf}(\rho)$  be the set of states that appear infinitely often in the run  $\rho$  (when treating the run as an infinite path). A run  $\rho$  of a Büchi automaton  $\mathcal{A}$  over an infinite word is *accepting* if and only if  $\text{inf}(\rho) \cap F \neq \emptyset$ , that is, when some accepting state appears in  $\rho$  infinitely often.

The structure shown in Figure 9.1 can be interpreted as a Büchi automaton. In this case one of the words it accepts is  $(ab)^\omega$ , that is, an infinite sequence of alternating  $a$ 's and  $b$ 's, starting with an  $a$ . The language it accepts is the set of words with *infinitely* many  $a$ 's, which can be written as the  $\omega$ -regular expression  $(b^*a)^\omega$ .



Kripke structure

Figure 9.2  
Transforming a Kripke structure

## 9.2. Model Checking

Finite automata can be used to model the behavior of a system  $M$  over the alphabet  $\Sigma$  or the advantages of using an automaton for the specification are repeated. The specification of a system  $M$  is the language  $L(M)$  of a Kripke structure  $\langle S, R, \Delta, \{t_i\} \rangle$  and only if  $(s, s') \in R$  and  $\alpha = L(s)$ . Figure 9.2 shows the transformation of a Kripke structure into a Büchi automaton.

The specification can be expressed using Büchi automata. Figure 2.2. In these examples, a subset of the proposition transition corresponds to a Büchi automaton. For example, when  $AP$  is labeled with  $\{X, Z\}$  and  $Y$  is included but may or may not be included.

The set of atomic propositions  $CR_0$  and  $CR_1$  of the mutual

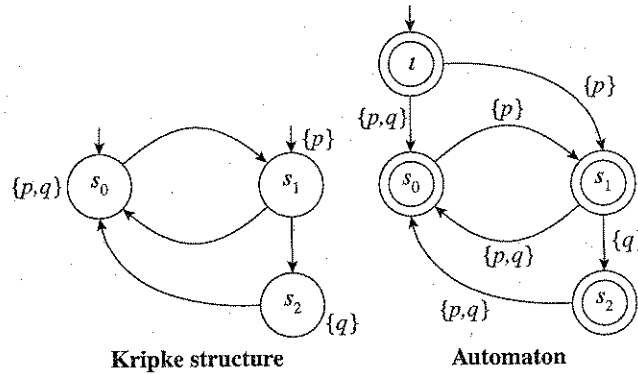


Figure 9.2 Transforming a Kripke structure into an automaton.

### 9.2 Model Checking Using Automata

Finite automata can be used to model concurrent and interactive systems. Either the state  $Q$  or the alphabet  $\Sigma$  can then represent the states of the modeled system. One of the main advantages of using automata for model checking is that both the modeled system and the specification are represented in the same way. A Kripke structure directly corresponds to an  $\omega$ -regular automaton, where all the states are accepting. Then, the set of behaviors of a system  $M$  is the language  $\mathcal{L}(\mathcal{A})$  of the corresponding automaton  $\mathcal{A}$ . Specifically, a Kripke structure  $\langle S, R, S_0, L \rangle$  where  $L : S \rightarrow 2^{AP}$ , can be transformed into an automaton  $\mathcal{A} = \langle \Sigma, S \cup \{t\}, \Delta, \{t\}, S \cup \{t\} \rangle$ , where  $\Sigma = 2^{AP}$ . We have  $(s, \alpha, s') \in \Delta$  for  $s, s' \in S$  if and only if  $(s, s') \in R$  and  $\alpha = L(s')$ . In addition,  $(t, \alpha, s) \in \Delta$  if and only if  $s \in S_0$  and  $\alpha = L(s)$ . Figure 9.2 shows a Kripke structure and its corresponding automaton.

The specification can also be given as an automaton  $\mathcal{S}$ , over the same alphabet. Then,  $\mathcal{L}(\mathcal{S})$  is the set of allowed behaviors. We will present several examples of properties expressed using Büchi automata. The properties refer to the mutual exclusion example in Figure 2.2. In these examples, we annotate edges with boolean expressions rather than a subset of the propositions  $AP$ . Each edge may represent several transitions, where each transition corresponds to a truth assignment for  $AP$  that satisfies the boolean expression. For example, when  $AP = \{X, Y, Z\}$ , an edge labeled  $X \wedge \neg Y$  matches the transitions labeled with  $\{X, Z\}$  and  $\{X\}$  (that is, the sets of propositions that include  $X$  and do not include  $Y$  but may or may not include  $Z$ ).

The set of atomic propositions  $AP$  in the following examples corresponds to the labels  $CR_0$  and  $CR_1$  of the mutual exclusion example. For instance, the proposition  $CR_0$  holds in

ample, the automaton  
passes through the states

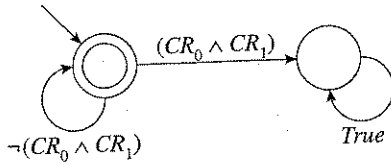
by  $\mathcal{A}$ . The automaton  
on  $\varepsilon + (a + b)^*a$ , that  
's or  $b$ 's and end with  
any finite number of

normal execution, we  
er will focus on finite  
re as finite automata  
re the superscript  $\omega$

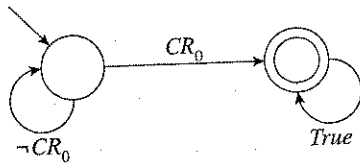
1. A Büchi automaton  
er,  $F$  is called the set  
on  $\mathcal{A}$  over an infinite  
tomaton over a finite  
all natural numbers.  
at the path is now an

$\rho$  (when treating the  
ite word is *accepting*  
appears in  $\rho$  infinitely

omaton. In this case  
ernating  $a$ 's and  $b$ 's,  
infinitely many  $a$ 's,



**Figure 9.3**  
Mutual exclusion property.



**Figure 9.4**  
A liveness property.

the states where the program counter of process  $P_0$  is  $CR_0$ . Figure 9.3 shows an automaton that specifies the property that the two processes cannot enter their critical section at the same time. This specification is given by the LTL path formula  $G \neg(CR_0 \wedge CR_1)$ . The property obviously holds for the mutual exclusion example.

Figure 9.4 shows an automaton that specifies the property that the process  $P_0$  will eventually enter its critical section, and is given by the LTL path formula  $F CR_0$ . This property does not hold in our example system, for it is possible that  $P_0$  never attempts to enter its critical section.

The system  $\mathcal{A}$  satisfies the specification  $S$  when

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(S) \tag{9.1}$$

That is, each behavior of the modeled system is among the behaviors that are allowed by the specification. Let  $\overline{\mathcal{L}(S)}$  be the language  $\Sigma^\omega - \mathcal{L}(S)$ . Then (9.1) can be rewritten as

$$\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(S)} = \emptyset \tag{9.2}$$

This means that there is no behavior of  $\mathcal{A}$  that is disallowed by  $S$ . If the intersection is not empty, any behavior in it corresponds to a counterexample.

Büchi automata are closed under intersection and complementation [39]. This means that there exists an automaton that accepts exactly the intersection of the languages of two automata, and an automaton that recognizes exactly the complement of the language of

a given automaton the intersection of computing the complement of this purpose can be

The formulation checking procedure

1. Complement the language  $\overline{\mathcal{L}(S)}$ .
2. Construct the au

If the intersection is must provide a counter can be represented  $u v^\omega$  where  $u$  and  $v$

In some implementations automaton for the complement this approach, the universality [162] is to use is easy.

Finally, the automata language such as LTL complementing  $S$ , and for the complement translation from LTL

Let  $\mathcal{B}_1 = \langle \Sigma, Q_1 \rangle$  automaton that accepts  $Q_1^0 \times Q_2^0 \times \{0\}$ ,  $Q_1$  following condition:

- $(r_i, a, r_m) \in \Delta_1$  and transitions of  $\mathcal{B}_1$  and  $\mathcal{B}_2$ .
- The third component
- if  $x = 0$  and  $r_m \in \dots$
- if  $x = 1$  and  $q_n \in \dots$
- if  $x = 2$  then  $y = \dots$
- otherwise,  $y = x$ .

a given automaton. We will later show how to construct an automaton that recognizes the intersection of two languages accepted by a pair of Büchi automata. The details of computing the complement of a Büchi automaton are rather involved. Constructions for this purpose can be found in [226, 234].

The formulation of the correctness criterion in (9.2) suggests the following model-checking procedure:

1. Complement the automaton  $\mathcal{S}$ , that is, construct an automaton  $\overline{\mathcal{S}}$  that recognizes the language  $\overline{\mathcal{L}(\mathcal{S})}$ .
2. Construct the automaton that accepts the intersection of the languages  $\mathcal{L}(\mathcal{A})$  and  $\overline{\mathcal{L}(\mathcal{S})}$ .

If the intersection is empty, announce that the specification  $\mathcal{S}$  holds for  $\mathcal{A}$ . Otherwise, we must provide a counterexample. We will show later that an infinite word in the intersection can be represented in a finitary way. Specifically, there is a counterexample of the form  $u v^\omega$  where  $u$  and  $v$  are finite words.

In some implementations such as SPIN [138, 140], the user is supposed to provide the automaton for the complement of  $\mathcal{S}$  directly instead of providing the automaton for  $\mathcal{S}$ . In this approach, the user specifies the bad behaviors rather than the good ones. Another possibility [162] is to use a different type of  $\omega$ -regular automata, for which complementation is easy.

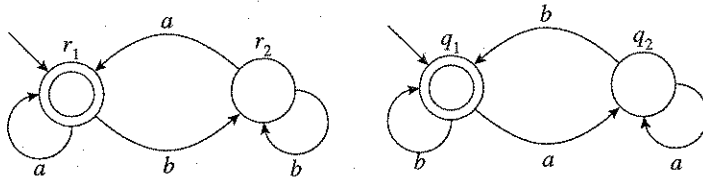
Finally, the automaton  $\mathcal{S}$  may be obtained using a translation from some specification language such as LTL. In this case, instead of translating a property  $\varphi$  into  $\mathcal{S}$  and then complementing  $\mathcal{S}$ , we can simply translate  $\neg\varphi$ , which immediately provides an automaton for the complement language, as required in (9.2). Later, we will provide an efficient translation from LTL to Büchi automata.

Let  $\mathcal{B}_1 = \langle \Sigma, Q_1, \Delta_1, Q_1^0, F_1 \rangle$  and  $\mathcal{B}_2 = \langle \Sigma, Q_2, \Delta_2, Q_2^0, F_2 \rangle$ . We can build an automaton that accepts  $\mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$  as follows:  $\mathcal{B}_1 \cap \mathcal{B}_2 = \langle \Sigma, Q_1 \times Q_2 \times \{0, 1, 2\}, \Delta, Q_1^0 \times Q_2^0 \times \{0\}, Q_1 \times Q_2 \times \{2\} \rangle$ . We have  $((r_i, q_j, x), a, (r_m, q_n, y)) \in \Delta$  if and only if the following conditions hold:

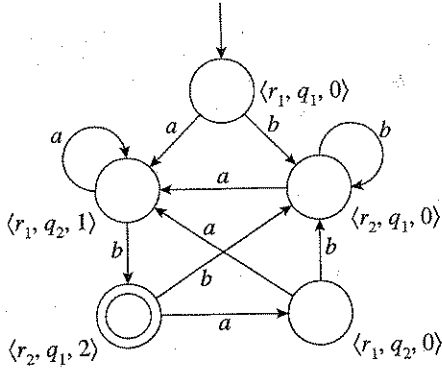
- $(r_i, a, r_m) \in \Delta_1$  and  $(q_j, a, q_n) \in \Delta_2$ , that is, the local components agree with the transitions of  $\mathcal{B}_1$  and  $\mathcal{B}_2$ .
- The third component is affected by the accepting conditions of  $\mathcal{B}_1$  and  $\mathcal{B}_2$ .
  - if  $x = 0$  and  $r_m \in F_1$ , then  $y = 1$ .
  - if  $x = 1$  and  $q_n \in F_2$ , then  $y = 2$ .
  - if  $x = 2$  then  $y = 0$ .
  - otherwise,  $y = x$ .

(9.1)

(9.2)



**Figure 9.5**  
An automaton for infinite number of *a*'s (left) and an automaton for an infinite number of *b*'s (right).



**Figure 9.6**  
An automaton for words with an infinite number of *a*'s and *b*'s.

The third component is responsible for guaranteeing that accepting states from both  $\mathcal{B}_1$  and  $\mathcal{B}_2$  appear infinitely often. Note that accepting states from both automata may appear together only finitely many times even if they appear individually infinitely often. Hence setting  $F = F_1 \times F_2$  does not work. The third component is initially 0. It changes from 0 to 1 when an accepting state of the first automaton is seen. It changes from 1 to 2 when an accepting state of the second automaton is seen, and in the next state, returns back to 0. The constructed automaton accepts exactly when infinitely many states from  $F_1$  and infinitely many states from  $F_2$  occur. The intersection of the automata in Figure 9.5 appears in Figure 9.6. Only nodes reachable from the initial state are shown.

A simpler intersection is obtained when all of the states of one of the automata are accepting. Such an intersection is used, for instance, in Equation 9.2, because all the states of the automaton for the modeled system are accepting. Assume all of the states of  $\mathcal{B}_1$  are accepting and that the acceptance set of  $\mathcal{B}_2$  is  $F_2$ . Their intersection will be defined as follows:

$$\mathcal{B}_1 \cap \mathcal{B}_2 = \langle \Sigma, Q_1 \times$$

The accepting states a state. Moreover,  $\langle r_i, \Delta_2$ .

The general algorithm fairness constraints. It accepting.

### 9.2.1 Nondetermini

For both regular and deterministic. That is, the nondeterministic finite deterministic automaton, the subset construction construct an equivalent  $\Delta' \subseteq 2^Q \times \Sigma \times 2^Q$  con

$$Q_2 = \bigcup_{q \in Q_1} \{q' \mid (q, a, q'$$

The set  $F'$  is defined as be represented as a func states that  $\mathcal{M}$  can reach

Complementing a nc first determinizing it usi the nonaccepting states. Büchi automaton has an by a deterministic Büc  $v \in \Sigma^\omega$ : If there are infin state, then  $v$  is in the la run for each finite prefix whose finite runs reach the automaton on  $v$ . By

Consider the automaton  $\Sigma = \{a, b\}$  that have only is no deterministic autom Büchi automaton that cou

$$\mathcal{B}_1 \cap \mathcal{B}_2 = \langle \Sigma, Q_1 \times Q_2, \Delta', Q_1^0 \times Q_2^0, Q_1 \times F_2 \rangle$$

The accepting states are pairs from  $Q_1 \times F_2$  in which the second component is an accepting state. Moreover,  $(\langle r_i, q_j \rangle, a, \langle r_m, q_n \rangle) \in \Delta'$  if and only if  $(r_i, a, r_m) \in \Delta_1$  and  $(q_j, a, q_n) \in \Delta_2$ .

The general algorithm for computing intersection is useful for verifying systems with fairness constraints. In this case, some of the states of the system automaton  $\mathcal{B}_1$  may not be accepting.

### 9.2.1 Nondeterministic Büchi Automata

For both regular and Büchi automata, we allow the transition relation  $\Delta$  to be nondeterministic. That is, there can be transitions  $(q, a, l), (q, a, l') \in \Delta$ , where  $l \neq l'$ . Any nondeterministic finite automaton on *finite words* can be translated into an equivalent deterministic automaton, that is, one that accepts the same language. This is done using the *subset construction*. For a nondeterministic automaton  $\mathcal{M} = \langle \Sigma, Q, \Delta, Q^0, F \rangle$ , we construct an equivalent deterministic automaton  $\mathcal{M}' = \langle \Sigma, 2^Q, \Delta', \{Q^0\}, F' \rangle$ , such that  $\Delta' \subseteq 2^Q \times \Sigma \times 2^Q$  contains  $(Q_1, a, Q_2)$  where

$$Q_2 = \bigcup_{q \in Q_1} \{q' \mid (q, a, q') \in \Delta\}.$$

The set  $F'$  is defined as  $\{Q' \mid Q' \subseteq Q \wedge Q' \cap F \neq \emptyset\}$ . Because  $\mathcal{M}'$  is deterministic,  $\Delta'$  can be represented as a function  $\Delta' : 2^Q \times \Sigma \rightarrow 2^Q$ . Each state of  $\mathcal{M}'$  corresponds to the set of states that  $\mathcal{M}$  can reach after reading some given input sequence.

Complementing a nondeterministic automaton over finite words can be performed by first determinizing it using the subset construction. Then, we interchange the accepting and the nonaccepting states. However, for Büchi automata the situation is different. Not every Büchi automaton has an equivalent deterministic Büchi automaton. A language recognized by a deterministic Büchi automaton  $\mathcal{B}$  satisfies the following condition for each word  $v \in \Sigma^\omega$ : If there are infinitely many finite prefixes of  $v$  whose finite runs reach an accepting state, then  $v$  is in the language. If the automaton is deterministic then there is a unique run for each finite prefix of a word. Suppose there are infinitely many finite prefixes of  $v$  whose finite runs reach accepting states. Then, these runs are prefixes of the unique run of the automaton on  $v$ . By definition, this run must be accepting.

Consider the automaton in Figure 9.7. It accepts the language of infinite words over  $\Sigma = \{a, b\}$  that have only finitely many  $a$ 's. This is a nondeterministic automaton, but there is no deterministic automaton that can recognize this language. If there were a deterministic Büchi automaton that could recognize this language, it would have to reach some accepting

ber of  $b$ 's (right).

g states from both  $\mathcal{B}_1$  automata may appear infinitely often. Hence  $v \neq 0$ . It changes from 0 to 1 when it goes from 1 to 2 when it reaches state, returns back to 0 by states from  $F_1$  and in Figure 9.5 appears

of the automata are because all the states in all of the states of  $\mathcal{B}_1$  on will be defined as