describes the interleaved execution of the concurrent processes. The formula for the transition relation of process \( P_i \) is conjuncted with same \((V \setminus V_i) \land \text{same}(PC \setminus \{p_{c_i}\})\). 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 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.

\[ C(l, \ \text{wait}(b), l') \]  

is a disjunction of the following two formulas:

- \((p_{c_i} = l \land p_{c_i}' = l' \land \neg b \land \text{same}(V_i))\)
- \((p_{c_i} = l \land p_{c_i}' = l' \land b \land \text{same}(V_i))\)

- **Lock:** The statement lock\((v)\) is similar to the statement 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.

\[ C(l, \ \text{lock}(v), l') \]  

is a disjunction of the following two formulas:

- \((p_{c_i} = l \land p_{c_i}' = l' \land v = 1 \land \text{same}(V_i))\)
- \((p_{c_i} = l \land p_{c_i}' = l' \land v = 0 \land v' = 1 \land \text{same}(V_i \setminus \{v\}))\)

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

\[ C(l, \ \text{unlock}(v), l') \equiv p_{c_i} = l \land p_{c_i}' = l' \land v' = 0 \land \text{same}(V_i \setminus \{v\}) \]

### 2.3 Example of Program Translation

Consider a simple **mutual exclusion** program

\[ P = m : \text{cobegin } P_0 \parallel P_1 \text{ coend } m' \]
with two processes $P_0$ and $P_1$, where

$$\begin{align*}
P_0 &:: \ \ l_0: \quad \text{while True do} \\
&\quad NC_0: \ \ \text{wait}(turn = 0); \\
&\quad CR_0: \ \ \text{turn} := 1; \\
&\quad \text{end while}; \\
&\quad l'_0
\end{align*}$$

$$\begin{align*}
P_1 &:: \ \ l_1: \quad \text{while True do} \\
&\quad NC_1: \ \ \text{wait}(turn = 1); \\
&\quad CR_1: \ \ \text{turn} := 0; \\
&\quad \text{end while}; \\
&\quad l'_1
\end{align*}$$

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 $\bot$ the value of $pc$ when $P_1$ and $P_2$ 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 $\bot$. 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

$$S_0(V, PC) \equiv pc = m \land pc_0 = \bot \land pc_1 = \bot.$$

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

- $pc = m \land pc_0 = l_0 \land pc_1 = l_1 \land pc' = \bot$
- $pc_0 = l_0' \land pc_1 = l'_1 \land pc' = m' \land pc_0' = \bot \land pc_1' = \bot$
- $C(l_0, P_0, l'_0) \land same(V \setminus V_0) \land same(PC \setminus \{pc_0\})$, which is equivalent to $C(l_0, P_0, l'_0) \land same(pc, pc_1)$
- $C(l_1, P_1, l'_1) \land same(V \setminus V_1) \land same(PC \setminus \{pc_1\})$, which is equivalent to $C(l_1, P_1, l'_1) \land same(pc, pc_0)$
For each process $P_i$, $C(i, P_i)$ is the disjunction of:

- $p_{0i} = l_i$ \land $p_{1i} = NC_i \land true \land same(turn)$
- $p_{0i} = NC_i \land p_{1i} = CR_i \land true \land turn = i \land same(turn)$
- $p_{0i} = CR_i \land p_{1i} = l_i \land true \land turn = i \land same(turn)$
- $p_{0i} = CR_i \land p_{1i} = NC_i \land turn = i \land same(turn)$

The Kripke structure in Figure 2.2 is derived from the formulas $S_i$ and $R_i$ 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 fails to guarantee the required mutual exclusion property. However, this program fails to enter its critical region without ever being able to do so, while the other processes stay in their critical region forever. Later, we will see how to formulate and model such properties.

3.1 The Compu...

Temporal logic, designed for describ...
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 \( (\Sigma, Q, \Delta, Q^0, F) \) 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 \( \rho : \{0, 1, \ldots, |v|\} \rightarrow Q \) 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
word $v$ if and only if there exists an accepting run of $A$ on $v$. For example, the automaton in Figure 9.1 accepts the word $aabb$ because there is a run that passes through the states $q_1 q_2 q_3 q_4 q_2 q_1$.

The language of $A$, $L(A) \subseteq \Sigma^*$ consists of all the words accepted by $A$. The automaton in Figure 9.1 accepts the language described by the regular expression $e + (a + b)^*a$, that is, either the empty word $e$, 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 $A$ 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 $\inf(\rho)$ 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 $A$ over an infinite word is accepting if and only if $\inf(\rho) \cap F \neq \emptyset$, 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 $(b^*a)^\omega$.

9.2. Model Checking

Finite automata can be $Q$, or the alphabet $\Sigma$ as advantages of using all the specification are rep to an $\omega$-regular automa of a system $M$ is the $L$ Kripke structure $(S, R, A = (\Sigma, S \cup \{t\}, \Delta, t$ and only if $(s, s') \in R$ a $a = L(s)$. Figure 9.2 sh

The specification can $L(S)$ is the set of allo expressed using Büchi a Figure 2.2. In these exam subset of the proposition transition corresponds to For example, when $AP$ labeled with $(X, Z)$ and include $Y$ but may or ma

The set of atomic prop $CR_0$ and $CR_1$ of the mut
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 $L(A)$ of the corresponding automaton $A$. Specifically, a Kripke structure $\langle S, R, S_0, L \rangle$ where $L : S \to 2^{AP}$, can be transformed into an automaton $A = (\Sigma, S \cup \{i\}, \Delta, \{i\}, S \cup \{i\})$, where $\Sigma = 2^{AP}$. We have $(s, a, s') \in \Delta$ for $s, s' \in S$ if and only if $(s, s') \in R$ and $\alpha = L(s')$. In addition, $(i, a, s) \in \Delta$ 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 $S$, over the same alphabet. Then, $L(S)$ 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 $X \land Y$ 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
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 $G \neg(CR_0 \land CR_1)$. 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 $\mathcal{S}$ when

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

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

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

This means that there is no behavior of $\mathcal{A}$ that is disallowed by $\mathcal{S}$. 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 formulation checking procedure

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

If the intersection is empty, the system is correct. If the intersection is non-empty, the union of the behaviors in the intersection must provide a counterexample, and can be represented as $u \sqcup v^\omega$, where $u$ and $v$ are infinite paths.

In some implementations, the counterexample is to be found in the input stream $I$.

Finally, the automaton $\mathcal{A}$ can be complemented $\mathcal{A}^c$ to $\mathcal{S}^c$.

Let $B_1 = (Q_1, \Sigma, \delta_1, Q_1, 0)$ be the automaton that accepts $Q_1^1 \times Q_2^2 \times \{0\}$, $Q_1^1$ following condition:

- $(r, a, r_m) \in \Delta_1$ actions of $B_2$ and $B_3$.
- The third component
  - if $x = 0$ and $r_m \in s$, $x = 1$ and $q_2 \in i$, $x = 2$ then $y = 0$.
  - otherwise, $y = x$. 

Model Checking and
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 \( \mathcal{S}' \) that recognizes the
language \( L(\varepsilon) \).

2. Construct the automaton that accepts the intersection of the languages \( L(\mathcal{A}) \) and \( L(\mathcal{S}') \).

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
\( w \uparrow \uparrow \), where \( w \) and \( \uparrow \) 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 pos-
sibility [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 \( B_1 = (\Sigma, Q_1, \Delta_1, Q_1^0, F_1) \) and \( B_2 = (\Sigma, Q_2, \Delta_2, Q_2^0, F_2) \). We can build an au-
tomaton that accepts \( L(B_1) \cap L(B_2) \) as follows: \( B_1 \cap B_2 = \langle \Sigma, \times Q_1 \times Q_2 \times \times \times 0, 1, 2, \Delta, Q_1 \times Q_2 \times \times \times 2 \rangle \). We have \( (x, q, a, y) \in \Delta \) if and only if the
following conditions hold:

- \( (r, a, r, m) \in \Delta_1 \) and \( (q, a, q, q, y) \in \Delta_2 \), that is, the local components agree with the transi-
tions of \( B_1 \) and \( B_2 \).

- The third component is affected by the accepting conditions of \( B_1 \) and \( B_2 \):
  - if \( x = 0 \) and \( r, m \in F_1 \), then \( y = 1 \).
  - if \( x = 1 \) and \( q, m \in F_2 \), then \( y = 2 \).
  - if \( x = 2 \) then \( y = 0 \).
  - otherwise, \( y = x \).
The third component is responsible for guaranteeing that accepting states from both $B_1$ and $B_2$ 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 $B_1$ are accepting and that the acceptance set of $B_2$ is $F_2$. Their intersection will be defined as follows:
\[ B_1 \cap B_2 = (\Sigma, Q_1 \times Q_2, \Delta', Q_1^0 \times Q_2^0, Q_1 \times F_2) \]

The accepting states are pairs from \( Q_1 \times F_2 \) in which the second component is an accepting state. Moreover, \((r, q_j, a, (r_m, q_n)) \in \Delta'\) if and only if \((r, a, r_m) \in \Delta_1\) 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 \( B_1 \) may not be accepting.

### 9.2.1 Nondeterministic Büchi Automata

For both regular and Büchi automata, we allow the transition relation \( \Delta \) to be non-deterministic. 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 \( M = (\Sigma, Q, \Delta, Q^0, F) \), we construct an equivalent deterministic automaton \( M' = (\Sigma, 2^Q, \Delta', \{Q^0\}, F') \), 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 \( \{ Q' \mid Q' \subseteq Q \land Q' \cap F \neq \emptyset \} \). Because \( M' \) is deterministic, \( \Delta' \) can be represented as a function \( \Delta': 2^Q \times \Sigma \to 2^Q \). Each state of \( M' \) corresponds to the set of states that \( M \) 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 \( B \) 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