Finite-State Machines

In this chapter we study the simplest automaton: the finite-state machine. These machines, which have been studied in such diverse disciplines as computer design, neurophysiology, communications, linguistics, and the theory of computation, are widely known and well understood. The finite-state machine model arises naturally from physical settings in which information-denoting signals are processed.

Most complex devices, whether computers, engines, or communication systems, are realized by interconnecting many simple components. Only a finite number of physical components can be enclosed in a specified volume. The "symbols" used to signal events between components are often represented by the values of physical quantities such as mechanical positions, voltages, currents, temperatures, pressures, fluid flows, or chemical compositions. Because no signal can be measured with arbitrarily high accuracy to within arbitrarily small tolerances, designers of physical devices are limited to a small number of easily distinguishable values, both for the signals themselves and for the internal conditions (states) of the devices. In short, physical reality imposes an inescapable, bounded finiteness on physical systems.

A given signal or state requires a small but nonzero interval to measure. It follows that only a finite number of operations can reliably be performed in a finite amount of time. Physical reality imposes discreteness on systems, not only in the symbols that they may process or the states that they may assume, but also in the times at which they may change state.
Our natural tendency to decompose problem solutions into sequences of steps is manifested in our simple machine model by *sequential action*. Indeed, designing a machine to operate one step at a time is the simplest way known to make its behavior *deterministic*, that is, not subject to uncertainty. (Of what use is a telephone number that does not reach the same party each time dialed or a computer instruction whose outcome is different each time used?)

The properties of finiteness, discreteness, sequential action, and determinism are embodied in the finite-state machine model. Because of them, the model can be (and has been) applied in diverse settings, such as the following:

1. Mechanical systems (adding machines, for example) in which signals between components are represented by the positions of moving parts.
2. Digital electronic systems (computer logic circuits, for example) in which signals are represented by "high" or "low" voltage, or by the presence or absence of current.
3. Pneumatic or hydraulic systems, comprising networks of pipes and pressure-actuated valves, in which signals are represented by the presence or absence of pressure in a pipe. (Examples are found in many types of industrial control equipment.)
4. Chemical systems, in which signals are represented by the chemical compositions of materials. (An example is the transcription of the genetic code during cell reproduction.)

The finite-state machine is also useful for modeling physical systems in which combinations of these effects are found, as in relay switching networks (electromechanical action) or the human nervous system (electrochemical action).

Despite its strong roots in physical phenomena, the finite-state machine is not limited to modeling physical processes. It is frequently applied to information-processing procedures, such as encoding and decoding messages or syntax analysis of computer programs. It plays a central role in the more powerful models of computation, where it appears as the control unit of an abstract machine having access to a storage medium of unbounded capacity.

Finiteness implies that the (finite state) machine and any storage medium to which it has access are bounded. This imposes important limitations on the capabilities of the model. We shall encounter a multitude of information-processing activities for which there is no finite-state model. Despite such limitations, this model plays an important role in the theory of computation as one absolute measure of the complexity of computational processes.

Our approach to the mathematical formulation of the finite-state machine begins from the assumptions of finite numbers of parts and permissible signals. At the end of the chapter, we shall show how the same model follows from the assumption that the machine has "finite memory" of its past experience.
4.1 Properties of Finite-State Machines

Consider a transducer machine $M$ as shown in Figure 4.1. (Transducers were introduced in Chapter 1.) Assume that the machine operates at instants $t_0, t_1, \ldots, t_i, \ldots$, their origin ($t_0$) and spacing ($t_i - t_{i-1}, i \geq 1$) being arbitrary. For simplicity, we denote these time instants by the integers $t = 0, 1, 2, \ldots$. At $t = 0$ the parts of the machine are initialized to a known starting condition. At each subsequent moment $t$, $M$ receives an input symbol $s(t)$ through its input channel and transmits an output symbol $r(t)$ through its output channel. The input symbols are chosen from a finite input alphabet $S$:

$$s(t) \in S, \quad t = 1, 2, \ldots$$

Similarly, the output symbols $r(t)$ are chosen from a finite output alphabet $R$:

$$r(t) \in R, \quad t = 1, 2, \ldots$$

A sequence of input symbols presented to the machine is called a stimulus; the resulting sequence of output symbols is called the response of $M$ to the stimulus.

Suppose the transducer is constructed from a finite number of interconnected parts, each of which can assume any one of a finite number of states. The interconnections are paths along which signals pass from one part of the machine to another. Suppose that $M$ has $n$ parts and that $q^{(i)}(t)$ is the state assumed by the $i$th part at moment $t$. The total state of $M$ at time $t$ is the $n$-tuple

$$q(t) = (q^{(1)}(t), q^{(2)}(t), \ldots, q^{(n)}(t))$$

Since it is possible that each part might assume any one of its states independent of the other parts, the maximum possible number of total states of $M$ is the product of the numbers of states for each part. Thus, if each part can assume no more than $k$ states, there can be no more than $k^n$ total states of $M$. From now on, we use $q(t)$ to denote the (total) state of $M$ at moment $t$, and use $Q$ to denote the finite state set of $M$. 

![Figure 4.1. Finite-state transducer machine.](image-url)
Sec. 4.1  Properties of Finite-State Machines  91

Since the future behavior of $M$ clearly depends on the present state of $M$, it is natural to inquire about the state of $M$ prior to the presentation of any inputs. For the machine to have deterministic behavior, we must insist that its parts be placed in a fixed, known state before any stimulus is applied. The corresponding total state is called the initial state of $M$ and is denoted $q_r$. The "initialize" input in Figure 4.1 resets the parts of $M$ to the initial state, so that $q(0) = q_r$.

We make the further assumption that the machine changes state only at the moments $t = 0, 1, 2, \ldots$, so its behavior may be specified by giving the sequence of states

$q(0), q(1), q(2), \ldots, q(t), \ldots, \quad q(t) \in Q$

that describes the internal condition of $M$ at each moment.

The input channel enters $M$ and connects to certain parts of $M$. When input symbol $s(t + 1)$ arrives, it influences these parts [which are in a condition represented by state $q(t)$], and thereby establishes a new condition described by state $q(t + 1)$. The new state can depend only on the former state of $M$ and the input symbol $s(t + 1)$. Thus there exists a function $f$ that specifies the next state of $M$ in terms of its present state and the next arriving input symbol:

$q(t + 1) = f(q(t), s(t + 1)), \quad t \geq 0$

This function is called the state transition function of $M$. Its domain is the set of all state–symbol pairs, and its range is a subset of the states

$f: Q \times S \rightarrow Q$

The output channel of the machine conveys signals from certain parts of $M$. The output symbol generated by the arrival of an input symbol depends only on which symbol arrives and on which state $M$ is in just prior to its arrival. Thus there exists a function $g$ that specifies the output symbol produced by $M$ in terms of its present state and the arriving input symbol:

$r(t + 1) = g(q(t), s(t + 1)), \quad t \geq 0$

This function is called the output function of $M$. Its domain is the set of all state–input symbol pairs, and its range is the output alphabet

$g: Q \times S \rightarrow R$

In summary, the following are the properties of a finite-state machine:

1. The behavior of $M$ is defined only at the moments $t = 0, 1, 2, \ldots$.
2. The input symbols $s(t)$ are chosen from a finite input alphabet $S$.
3. The output symbols $r(t)$ are chosen from a finite output alphabet $R$.
4. The behavior of $M$ is uniquely determined by the sequence of input symbols presented.
5. The behavior of $M$ carries it through a sequence of states, each of which is a member of the state set $Q$.  
6. There is an initial state $q_I$ of $M$ that describes the condition of the parts of $M$ just before any stimulus is presented.

These properties lead to a mathematical description of a finite-state machine $M$ consisting of the following:

1. The finite sets $S$, $R$, and $Q$.
2. A state transition function $f$ that gives the next state of $M$ in terms of the current state and the next input symbol.
3. An output function $g$ that gives the next output symbol of $M$ in terms of the current state and next input symbol.
4. A predetermined initial state $q(0) = q_I$ in which $M$ is placed prior to instant $t = 0$.

These ideas are formalized in Definition 4.1.

4.1.1 Machines with Transition-Assigned Output

Definition 4.1: A *transition-assigned finite-state machine* is a six-tuple 

$$M = (Q, S, R, f, g, q_I)$$

where 
- $Q$ is a finite set of internal states
- $S$ is a finite input alphabet
- $R$ is a finite output alphabet
- $f$ is the state transition function
  $$f: Q \times S \rightarrow Q$$
- $g$ is the output function
  $$g: Q \times S \rightarrow R$$
- $q_I \in Q$ is the initial state

This type of machine, known in the literature as a *Mealy automaton*, is characterized by the association of output symbols with transitions between states. (Later in the chapter, machines with output symbols assigned to states are studied.) Because a finite-state machine deals with sequences of input and output symbols, and because its behavior is represented by a sequence of states, a finite-state machine is also known as a *sequential machine*. It is important to remember that a *finite-state machine includes specification of an initial state*.

To describe a particular finite-state machine, it is necessary to specify the state transition function and the output function. Two representations are standard: the *state table* and the *state diagram*. 
A state table is a tabular representation of the two functions, using one row for each state and one column for each input symbol. In the tabular position for state \( q \) and symbol \( s \), we write the next state \( q' = f(q, s) \) and the output symbol \( r = g(q, s) \), as shown in Figure 4.2a.

(a) State Table

(b) State Diagram

\[
\begin{array}{|c|c|}
\hline
s & \\
\hline
\cdots & q', r \\
\hline
\cdots & \\
\hline
\end{array}
\]

\[ q' = f(q, s) \]
\[ r = g(q, s) \]

Figure 4.2. State table and state diagram representations for transition-assigned machines.

A state diagram is a directed graph in which each node corresponds to a state of the machine and each directed arc indicates a possible transition from one state to another. Figure 4.2b shows how the transition and output functions are represented in a state diagram. Each arc is labeled with the input symbol that causes the transition and the output symbol that is generated. We write

\[ q \xrightarrow{s/r} q' \quad \text{to mean} \quad \begin{cases} f(q, s) = q' \\ g(q, s) = r \end{cases} \]

Unless otherwise specified, the first row of a state table will be assigned to the initial state of a machine. In a state diagram, the initial state is indicated by an arc having no node of origin, as shown in Figure 4.3.

Figure 4.3. Initial state of a machine.
Each directed path through the state diagram shows the behavior of the machine in response to the sequence of input symbols labeling the path. Suppose that \( q(0), q(1), \ldots, q(t) \) are successive nodes on a directed path (Figure 4.4) and the transitions are labeled \( s(1)/r(1), s(2)/r(2), \ldots, s(t)/r(t) \).

![Figure 4.4. Path of operation corresponding to an input sequence.](image)

Then the stimulus \( s(1)s(2) \ldots s(t) \) causes the machine to trace out the state sequence \( q(0), q(1), \ldots, q(t) \) and generate the response \( r(1)r(2) \ldots r(t) \). This is represented by the notation

\[
q(0) \xrightarrow{s(1)/r(1)} q(1) \xrightarrow{s(2)/r(2)} \ldots \xrightarrow{s(t)/r(t)} q(t)
\]

Note that the states of this sequence need not be distinct, for the path traced in the state diagram may contain loops.

If we denote the input string by \( \omega = s(1)s(2) \ldots s(t) \), and the output string by \( \varphi = r(1)r(2) \ldots r(t) \), then we may represent the machine's behavior by the more compact notation

\[
q = \omega/\varphi \rightarrow q'
\]

\[
\begin{align*}
q &= q(0) \\
q' &= q(t)
\end{align*}
\]

### 4.1.2 Examples

To make these ideas more concrete, we consider some examples of finite-state machines.
Example 4.1 (Modulo-3 Counter): We shall design a finite-state machine $M_1$ whose output tells the number of input symbols, modulo 3, that have been applied. (For integers $n \geq 0$ and $m > 0$, the value of $n \mod m$ is the remainder after dividing $n$ by $m$.) Let

$S = \{a\}$

$Q = \{A, B, C\}$

$R = \{0, 1, 2\}$

Let $t$ be the moment at which the last symbol of a stimulus $\omega$ is presented to $M_1$. Then the state $q(t)$ will be interpreted as follows:

$q(t) = A$ means $|\omega| \mod 3 = 0$

$q(t) = B$ means $|\omega| \mod 3 = 1$

$q(t) = C$ means $|\omega| \mod 3 = 2$

A state table and a state diagram for this machine are shown in Figure 4.5. The behavior of $M_1$ for the input sequence $aaaa$, for example, is

$$A \xrightarrow{a/1} B \xrightarrow{a/2} C \xrightarrow{a/0} A \xrightarrow{a/1} B$$

(a) State Diagram

(b) State Table

<table>
<thead>
<tr>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B, 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, 0</td>
</tr>
</tbody>
</table>

Figure 4.5. Modulo-3 counter.

Example 4.2 (Parity Checker): Let $\omega$ be a sequence of symbols in the binary alphabet $\{0, 1\}$. If $\omega$ contains an even number of 1's, it is said to have even parity; if $\omega$ contains an odd number of 1's, it is said to have odd parity.

We wish to design a finite-state machine $M_2$ that produces an output 1 whenever the parity of the input sequence is odd, and an output 0 whenever the parity of the input sequence is even. Clearly, we must have $S = R = \{0, 1\}$.

Observe that the parity of a string $\omega s$, $s \in \{0, 1\}$, is readily determined from $s$ and the parity of $\omega$: if $s = 0$, the parity of $\omega s$ is
the same as that of \( \omega \); if \( s = 1 \), the parity of \( \omega s \) is different from that of \( \omega \). Thus \( M_2 \) needs only two states, one indicating that the input sequence so far has even parity, the other indicating that it has odd parity. The machine \( M_2 \) is shown in Figure 4.6.

(a) State Diagram

(b) State Table

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A, 0</td>
<td>B, 1</td>
</tr>
<tr>
<td>B</td>
<td>B, 1</td>
<td>A, 0</td>
</tr>
</tbody>
</table>
```

Figure 4.6. Parity checker.

For the input sequence

\[ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \]

the behavior of \( M_2 \) is given by

\[ A \xrightarrow{0/0} A \xrightarrow{1/1} B \xrightarrow{0/1} B \xrightarrow{1/0} A \xrightarrow{1/1} B \]

producing the output sequence

\[ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \]

**Example 4.3 (Two-Unit Delay):** The input and output alphabets of machine \( M_3 \) are \{0, 1\}. The output sequence is to be a replica of the input sequence delayed by two time units:

\[ r(t) = s(t - 2), \quad t > 2 \]

We do not care what \( r(1) \) and \( r(2) \) are since no input symbols were applied before \( t = 1 \). The states of \( M_3 \) distinguish among input sequences according to the last two symbols presented, \( s(t - 1) \) and \( s(t - 2) \), as follows:

<table>
<thead>
<tr>
<th>( s(t - 1), s(t - 2) )</th>
<th>State of ( M_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>A</td>
</tr>
<tr>
<td>0 1</td>
<td>B</td>
</tr>
<tr>
<td>1 1</td>
<td>C</td>
</tr>
<tr>
<td>1 0</td>
<td>D</td>
</tr>
</tbody>
</table>

The state diagram is shown in Figure 4.7. A typical stimulus and the resulting response are

**Stimulus:** 0 0 0 1 1 0 1 0 0

**Response:** 0 0 0 0 0 1 1 0 1
for which $M_2$ has the state sequence

$$A \xrightarrow{0/0} A \xrightarrow{0/0} A \xrightarrow{1/0} D \xrightarrow{1/0} C \xrightarrow{0/1} B \xrightarrow{1/1} D \xrightarrow{0/0} B \xrightarrow{0/1} A$$

Note that an $n$-unit delay would require $2^n$ states.

### 4.1.3 Machines with State-Assigned Output

It is frequently convenient to specify the output of a finite-state machine just in terms of the state of the machine. For example, in the parity machine $M_2$ the output is uniquely determined by the state to which a transition leads. In a state-assigned machine the output function assigns an output symbol to each state.

**Definition 4.2:** A state-assigned finite-state machine is a six-tuple

$$M = (Q, S, R, f, h, q_i)$$

where

- $Q$ is a finite set of internal states
- $S$ is a finite input alphabet
- $R$ is a finite output alphabet
- $f$ is a state transition function
  $$f: Q \times S \rightarrow Q$$
- $h$ is an output function
  $$h: Q \rightarrow R$$
- $q_i \in Q$ is the initial state

This type of machine is known in the literature as a *Moore automaton*.

Figure 4.8 shows how state-assigned machines are represented by state tables and state diagrams. For state-assigned machines we write state
sequences as (state, output symbol) pairs joined by arrows labeled with the input symbol causing the indicated transition:

\[(q, r) \xrightarrow{s} (q', r')\] means \[
\begin{cases}
    f(q, s) = q' \\
    h(q) = r \\
    h(q') = r'
\end{cases}
\]

Later in the chapter we shall inquire whether these two models, one with transition-assigned outputs and the other with state-assigned outputs, are equivalent in computing power.

**Example 4.4: (Modulo-4 Up-Down Counter):** Machine \(M_4\) will analyze input sequences in the binary alphabet \(S = \{0, 1\}\). Let \(\omega = s(1)s(2) \ldots s(t)\) be an input string and define

\[
N_0(\omega) = \text{number of } 0\text{'s in } \omega \\
N_1(\omega) = \text{number of } 1\text{'s in } \omega
\]

so that \(|\omega| = N_0(\omega) + N_1(\omega) = t\). The last output of \(M_4\) is to be

\[
r(t) = [N_1(\omega) - N_0(\omega)] \mod 4
\]

Hence the output alphabet is \(R = \{0, 1, 2, 3\}\). The machine is specified in Figure 4.9 and is called an up-down counter owing to the appearance of its state diagram. Typical stimulus and response sequences are

**Stimulus:** 1 1 0 1 1 1 0 0
**Response:** 0 1 2 1 2 3 0 3 2

The corresponding state sequence is
Example 4.5 (Language Recognizer): Machine $M_5$ is to accept a binary string if and only if it begins with a 1 and contains exactly one 0. (The set of all such strings is the set $X = 11^*01^*$. ) A state diagram for $M_5$ appears in Figure 4.10. The machine has an output of 1, indicating acceptance, if and only if the input string leads it into state C. Observe that once $M_5$ enters state D, no subsequent sequence of input symbols can ever cause the machine to leave state D. Thus any string that is accepted must avoid leading the machine to state D. Such a state is called a trap state. Any input string not beginning with a 1 or having more than a single 0 leads $M_5$ to the trap state. States B and C distinguish between input sequences con-
taining no 0's and one 0, respectively. Thus each string in the set \( X \)
leads \( M_2 \) to its accepting state \( C \), and only such strings lead \( M_2 \) to \( C \).

### 4.1.4 Machine Complexity

It is interesting to study the relationship between the number of states of a machine \( M \) and the number of physical components required to realize \( M \). Suppose that \( M \) is built of \( n \) parts, each of which can assume either of two states. Then each total state of \( M \) is an \( n \)-tuple

\[
q = (q^{(1)}, q^{(2)}, \ldots, q^{(n)})
\]

in which \( q^{(n)} \) represents the state of the \( n \)th part. The number of total states is \( 2^n \). Therefore,

\[
\text{(number of parts)} \sim \log_2 \text{(number of states)}
\]

Adding another two-state part to a machine doubles the number of distinct total states. Thus the number of states grows exponentially with the number of parts.

How many states does a computer have? For simplicity, let us just consider the active memory of a computer, which in certain machines is of the order of \( 2^{15} \) words of 32 bits apiece. Since each bit may be either 0 or 1, and the memory consists of \( N = (32)(2^{15}) = 2^{20} \) bits, the memory has

\[
2^N = 2^{2^{20}} \approx 2^{1,000,000} \approx 10^{300,000}
\]

states!

It would hardly be practical to draw a state diagram for such a machine. The finite-state model, therefore, is not especially useful for studying the behavior of an entire computer. Indeed, for all practical purposes a computer has an infinite number of states, and that is why machine models that allow for infinite numbers of states are of great practical interest. The finite-state model has proved quite useful, however, in the design of subunits of a computer that have relatively few states.

### 4.2 State Sequences

We have seen that a sequence of states on a path through the state diagram of a machine corresponds to the application of a string of input symbols and the emission of a string of output symbols. We now give careful definitions for the notions of successor states and state sequences. These definitions are needed for discussing equivalence of machines and for studying their relation to languages in Chapter 5. The definitions hold for both state-assigned and transition-assigned machines.

**Definition 4.3:** Let \( M \) be a finite-state machine with transition function \( f : Q \times S \rightarrow Q \). If \( f(q, s) = q' \), we say that state \( q' \) is the
s-successor of state $q$ and write
\[ q \overset{r}{\rightarrow} q' \]
If a string of input symbols
\[ \omega = s(1)s(2) \ldots s(t) \]
takes $M$ from state $q = q(0)$ to state $q' = q(t)$, that is, if
\[ q(0) \overset{s(1)}{\rightarrow} q(1) \overset{s(2)}{\rightarrow} \ldots \overset{s(t)}{\rightarrow} q(t) \]
we say that state $q'$ is the $\omega$-successor of state $q$ and write
\[ q \overset{\omega}{\rightarrow} q' \]
Under these conditions
\[ q(0)q(1) \ldots q(t) \]
is called an admissible state sequence for $\omega$.

By these definitions we have effectively extended the domain of the state transition function to include all input strings, rather than just individual input symbols. That is, we now have
\[ f : Q \times S^* \rightarrow Q \]
where
\[ f(q, \omega) = q' \text{ if and only if } q \overset{\omega}{\rightarrow} q' \]
and
\[ f(q, \lambda) = q \text{ for all } q \in Q \]

Note that these four statements are equivalent:

1. There is an admissible state sequence from $q$ to $q'$ for input string $\omega = s(1)s(2) \ldots s(t)$.
2. $q \overset{\omega}{\rightarrow} q'$.
3. $f(q, \omega) = q'$.
4. There is a directed path from $q$ to $q'$ in the state diagram for $M$ with transitions labeled by input symbols $s(1), s(2), \ldots, s(t)$.

**Definition 4.4:** Let $M$ be a finite-state machine and suppose that
\[ q_1 = q(0), q(1), \ldots, q(t) = q \]
is an admissible state sequence for the input sequence
\[ \omega = s(1)s(2) \ldots s(t) \]

a. If $M$ is transition assigned with output function $g : Q \times S \rightarrow \mathbb{R}$, then
\[ \varphi = r(1)r(2) \ldots r(t) \]
is the response of $M$ to stimulus $\omega$ where
\[ r(i) = g(q(i - 1), s(i)), \quad i = 1, 2, \ldots, t \]
b. If $M$ is state assigned with output function $h : Q \rightarrow \mathbb{R}$, then
\[ \varphi = r(0)r(1)r(2) \ldots r(t) \]
is the response of $M$ to stimulus $\omega$ where

$$r(i) = h(q(i)), \quad i = 0, 1, \ldots, t$$

### 4.3 Conversion Between Transition- and State-Assigned Machines

Definition 4.4 points out an important distinction between transition- and state-assigned automata: a state-assigned machine has a response to the empty string! Our aim in this section is to show that, aside from this point, these two machine types are equivalent abstract models of computation.

How shall we decide whether the two models are equivalent? The natural question to ask is whether a machine of one type can always be designed to mimic the responses of a given machine of the other type.

**Definition 4.5:** A transition-assigned machine $M_t$ and a state-assigned machine $M_s$ are similar if, for each possible stimulus, the response of $M_s$ is exactly that of $M_t$ preceded by one arbitrary, but fixed, symbol.

The meaning of this definition is suggested by Figure 4.11. The same string $\omega$ is applied to both $M_t$ and $M_s$. The response of $M_t$ is the string $\varphi$. The response of $M_s$ is the string $r_0\varphi$ for some fixed symbol $r_0$ determined by $M_t$.

![Figure 4.11](image)

**Figure 4.11.** Illustrating the definition of similarity.

**Theorem 4.1:** For each state-assigned machine $M_s$ there exists a similar transition-assigned machine $M_t$. Conversely, for transition-assigned machine $M_t$ there exists a similar state-assigned machine $M_s$.

Proving each of these two assertions consists of constructing a machine of one type from any given machine of the other, and demonstrating that the constructed machine is similar to the original.
The simpler problem is to obtain $M_r$ from $M_s$. The states and transitions of $M_s$ are chosen to be the same as those of $M_r$. Whenever state $q$ in $M_s$ has output $r$, each transition of $M_s$ into state $q$ is labeled with output $r$. The construction is as follows: if

$$M_s = (Q, S, R, f, h, q_1)$$

then

$$M_r = (Q, S, R, f, g, q_1)$$

in which

$$g(q, s) = h(f(q, s)), \quad \forall q \in Q, s \in S$$

Since the states and transitions of both $M_s$ and $M_r$ are identical, it is clear that a state sequence is admissible in $M_s$ for a stimulus $\omega$ if and only if it is admissible in $M_r$ for $\omega$. The similarity of $M_s$ and $M_r$ follows directly from this fact and the relationship of the output functions.

**Example 4.6:** Figure 4.12 illustrates the construction of a transition-assigned machine from a state-assigned machine. The property that $M_r$ has output 1 in its initial state is lost in $M_r$.

![Figure 4.12. Construction of $M_r$ from $M_s$.](image-url)
The construction of $M_r$ from a given machine $M_s$ is somewhat more complicated. We cannot simply reverse the construction given above, because $M_s$ may contain a state $q$ whose input transitions are labeled by more than one output symbol. To circumvent this difficulty, we let the states of $M_r$ be the set of all state-output pairs $Q_r \times R$ in $M_s$. Machine $M_r$ will enter state $(q, r)$ whenever $M_s$ enters state $q$ and emits output symbol $r$. The construction is as follows: if

$$M_r = (Q_r, S, R, f_r, g, q_i)$$
then

$$M_s = (Q_r, S, R, f_s, h, (q_i, r_0))$$
where

$$Q_r = Q_s \times R$$

The functions $f_r$ and $h$ are defined as follows: whenever $M_r$ has a transition

$$q \xrightarrow{u/r} q'$$
then $M_s$ has the transition

$$((q, r'), r') \xrightarrow{u} ((q', r), r)$$
for each $r' \in R$. Whenever $M_s$ has the transition

$$q_i \xrightarrow{u/r} q'$$
then $M_r$ has the corresponding transition

$$((q_i, r_0), r_0) \xrightarrow{u} ((q', r), r)$$
where $r_0 \in R$ is an arbitrarily chosen output symbol for the initial state of $M_r$.

Given an input string $\omega = s(1) \ldots s(n)$ and an admissible state sequence $q(0) \ldots q(n)$ for $\omega$ in $M_s$, an admissible state sequence for $\omega$ in $M_r$ will consist of the pairs

$$(q(i), g(q(i - 1), s(i))), \quad i = 1, 2, \ldots, n$$

By construction, the response of $M_r$ will be the same as that of $M_s$ with the exception of the initial symbol $r_0$.

Example 4.7: Figure 4.13 illustrates the construction of a state-assigned machine from a transition-assigned machine. We have arbitrarily chosen 0 as the output specified for the initial state of $M_s$.

We conclude that the two finite-state models are of equivalent computational power, with the minor exception that a state-assigned machine has a specific response for the empty input string. Since this feature is important in connection with the study of machines as language recognizers, we shall use the state-assigned model extensively in later chapters.
4.4 Equivalence of Finite-State Machines

An important question that recurs throughout discussions of automata is that of equivalence: under what circumstances do two automata exhibit identical behavior? Regarded as language recognizers, two automata are equivalent if and only if they recognize the same language; as generators, if and only if they generate the same language; as transducers, if and only if they produce identical transductions of each input string.

We shall see later that it is not always possible to decide whether two automata have equivalent behavior. In this section we shall find that the question can be answered in every case if the automata are finite-state machines. Indeed, each such automaton can be put into an essentially unique standard form.

Solving the equivalence problem for finite-state machines also resolves several issues arising naturally in the solution of design problems:

1. Given the state diagram of a machine, is it possible to detect and eliminate redundant states without altering the machine’s behavior?
2. Is it possible, by eliminating redundant states, to obtain a unique minimum-state machine equivalent to the original?

We shall find affirmative answers to both questions. Although using the minimum number of states need not lead directly to a physical machine of simplest construction, the minimum-state machine is often a good starting point for hardware design.

We begin with a definition of equivalent machines:

**Definition 4.6:** Two machines $M_1$ and $M_2$ are *equivalent* if and only if

1. Their input alphabets and their output alphabets are the same: $S_1 = S_2$, $R_1 = R_2$.
2. For each stimulus, $M_1$ and $M_2$ produce identical responses. That is, if $s_1(i) = s_2(i)$, $t \geq 1$, then $r_1(i) = r_2(i)$, $t \geq 1$.

If $M_1$ and $M_2$ are equivalent, we write $M_1 \sim M_2$.

According to this definition, equivalent machines operated as in Figure 4.14 must produce identical output sequences, regardless of the input sequence presented.

![Figure 4.14. Conceptual test for equivalent machines.](image)

Definition 4.6 establishes an equivalence relation on the class of finite-state automata. To confirm this, we must show that the properties of reflexivity, symmetry, and transitivity are satisfied:

1. **Reflexivity:** $M \sim M$. Each machine is clearly equivalent to itself.
2. **Symmetry:** If $M_1 \sim M_2$, then $M_2 \sim M_1$. Interchanging subscripts in the body of Definition 4.6 does not alter the meaning of equivalence.
3. **Transitivity:** If $M_1 \sim M_2$ and $M_3 \sim M_2$, then $M_1 \sim M_3$. Whenever $r_1(i) = r_2(i)$, $t \geq 1$, and $r_2(i) = r_3(i)$, $t \geq 1$, it follows that $r_1(i) = r_3(i)$, $t \geq 1.$
The relation of machine equivalence partitions the class of finite-state automata into collections of mutually equivalent machines.

4.5 Equivalent States

To solve the equivalence problem for finite-state automata, we must relate the equivalence of two automata to their internal structure as expressed by state diagrams or state tables. We consider two states of a machine to be equivalent if it is impossible to distinguish two copies of the machine, one started in each state.

**Definition 4.7:** Two states $q_a$ and $q_b$ of a transition-assigned machine $M = (Q, S, R, f, g, q_I)$ are **equivalent states** if and only if the machines

$$M_a = (Q, S, R, f, g, q_a)$$

and

$$M_b = (Q, S, R, f, g, q_b)$$

are equivalent. An analogous statement applies to state-assigned machines. If $q_a$ and $q_b$ are equivalent states, we write $q_a \sim q_b$.

The notion of state equivalence is illustrated by Figure 4.15. If states $q_a$ and $q_b$ are equivalent in machine $M$, the two copies of $M$ cannot be distinguished by presenting stimuli and comparing the responses.

![Figure 4.15. Conceptual test for equivalent states.](image)

Because state equivalence is expressed in terms of machine equivalence, it is easy to see that the relation of state equivalence partitions the states of a machine into equivalence classes such that

1. All states in the same class are mutually equivalent.
2. Any two states in different classes are not equivalent.
Note that if two states are not equivalent, there must be an input sequence which, when applied to two copies of a machine as depicted in Figure 4.15, will produce different output sequences.

If two states are equivalent, one of them is clearly redundant, since all transitions into one of the states may be switched to the equivalent state without affecting the behavior of the machine. To identify redundant states in a machine, we need a method of constructing the equivalence classes of states from the state diagram or state table of the machine. Before developing such a procedure, however, we shall explore some consequences of the definition.

An interpretation of state equivalence in terms of the structure of a machine is provided by the next theorem.

**Theorem 4.2:** States $q_a$ and $q_b$ of a finite-state machine $M$ are equivalent if and only if

1a. Transition-assigned: For all $s \in S$, $g(q_a, s) \sim g(q_b, s)$. That is, the outputs for corresponding transitions from the two states are identical.

1b. State-assigned: $h(q_a) = h(q_b)$. That is, the output symbols for the two states are identical.

2. For all $s \in S$, $f(q_a, s) \sim f(q_b, s)$. That is, the $s$-successors of the two states are themselves equivalent states.

**Proof:** We give the proof for the case of transition-assigned machines, and leave the proof for state-assigned machines as an exercise for the reader.

**If:** We must show that if conditions 1a and 2 hold, then $q_a \sim q_b$.

Let $s \omega$ be an arbitrarily chosen input string consisting of the single letter $s$ followed by some string $\omega \in S^*$. Consider $M$’s behavior when $s \omega$ is applied:

$$
q_a \xrightarrow{s/r} q'_a \xrightarrow{u/p} q''_a \\
q_b \xrightarrow{s/r'} q'_b \xrightarrow{u/p'} q''_b
$$

To show that $q_a \sim q_b$, we must show that the responses are identical; that is,

$$r \varphi = r' \varphi'$$

Condition 1a asserts that $r = r'$. Condition 2 asserts that $q_a \sim q_b'$ and thus that $\varphi = \varphi'$.

**Only if:** We must show that, if $q_a \sim q_b$, then conditions 1a and 2 hold. Again, let $s \omega$ be an arbitrarily chosen input string, as above, and consider $M$’s behavior when $s \omega$ is applied:

$$
q_a \xrightarrow{s/r} q'_a \xrightarrow{u/p} q''_a \\
q_b \xrightarrow{s/r'} q'_b \xrightarrow{u/p'} q''_b
$$
Since \( q_a \sim q_b \), the responses to \( s \omega \) must be identical. That is,

\[ r \varphi = r' \varphi' \]

from which we conclude that \( r = r' \) (thus condition 1a is satisfied), and that \( \varphi = \varphi' \) (thus \( q'_a \sim q'_b \), and condition 2 is satisfied).

In the machine of Figure 4.16 we see that states A and B are equivalent because

\[
\begin{align*}
g(A, 0) &= g(B, 0) = 0 \\
g(A, 1) &= g(B, 1) = 0
\end{align*}
\]

and

\[
\begin{align*}
f(A, 0) &= B & f(A, 1) &= C \\
f(B, 0) &= B & f(B, 1) &= C
\end{align*}
\]

and certainly \( B \sim B \) and \( C \sim C \).

![Figure 4.16. Machine with A \sim B.](image)

Now let us try to use Theorem 4.2 to decide if \( B \sim C \) in Figure 4.17. We have

\[
\begin{align*}
g(B, 0) &= g(C, 0) = 0 \\
g(B, 1) &= g(C, 1) = 1
\end{align*}
\]

![Figure 4.17. Machine with B \sim C.](image)
so the outputs are identical. Also

\[ f(B, 1) = A \]
\[ f(C, 1) = A \]

and certainly \( A \sim A \). However,

\[ f(B, 0) = C \]
\[ f(C, 0) = B \]

We have the result that \( B \sim C \) if \( B \sim C \)! This circularity reduces the value of Theorem 4.2 as a test for state equivalence: it is true that \( B \sim C \), but this does not follow directly from Theorem 4.2.

Figure 4.18 shows two copies of the same machine side by side. Viewed as a single state diagram of a machine \( M \), it is obvious that

\[ A \sim C \]
\[ B \sim D \]

since the behavior of \( M \) is the same whether it is started in either of \( A \) or \( C \), or either of \( B \) or \( D \). Again, these state equivalences cannot be deduced through direct application of Theorem 4.2. We need an equivalence test that will conclude that \( A \sim C \) and \( B \sim D \) in this type of disconnected machine, and that \( B \sim C \) in a connected machine such as the one in Figure 4.17.

Theorem 4.2 is more useful for deciding when two states are not equivalent. In principle, we can test the states of a machine in pairs to decide which states are not equivalent, by default discovering which states are equivalent. This is precisely what is done in Section 4.6, where we develop an orderly procedure for sorting states into equivalence classes.

### 4.6 State Reduction and Equivalence Testing

The methods discussed in the remainder of this chapter are applicable to both transition- and state-assigned machines. The presentation is made in
terms of the transition-assigned model. The reader should be able to supply
the minor modifications required for the state-assigned model.

4.6.1 Reduced, Connected Machines

Intuitively, a machine is in its simplest form once all redundant and
unusable states have been removed.

**Definition 4.8:** A finite-state machine is *reduced* if it contains no pair
of equivalent states.

**Definition 4.9:** A state $q$ in a finite-state automaton is *accessible* if
there is some input string $\omega$ such that $q_1 \xrightarrow{\omega} q$. A finite-state machine
is *connected* if every state is accessible from the initial state.

The reader should convince himself that if state $q$ in an $n$-state machine is
accessible, there exists an input string $\omega$ such that $|\omega| < n$ and $q_1 \xrightarrow{\omega} q$.

It should be evident that no state can be removed from a reduced and
connected machine without affecting its behavior. Inaccessible states can
always be removed from a machine, since the machine can never enter such
states.

4.6.2 Distinguishing Sequences and $k$-Equivalence

If states $q_a$ and $q_b$ of machine $M$ are equivalent, no input string applied
to the experiment of Figure 4.15 can evoke different responses from the two
copies of $M$. Conversely, if the states $q_a$ and $q_b$ are not equivalent, there is
an input sequence which, when applied to the experiment of Figure 4.15, will
evoke responses differing in at least one symbol. In the latter case, $q_a$ and $q_b$
are said to be *distinguishable* states.

**Definition 4.10:** States $q_a$ and $q_b$ of a transition-assigned machine
$M = (Q, \Sigma, R, f, g, q_0)$ are *$k$-distinguishable* if there exists a string
$\omega \in \Sigma^*$ with $|\omega| \leq k$, such that the responses of

$$M_a = (Q, \Sigma, R, f, g, q_a)$$

and

$$M_b = (Q, \Sigma, R, f, g, q_b)$$

to $\omega$ differ in at least one symbol. Such a string $\omega$ is called a *distingui-
shing sequence* for states $q_a$ and $q_b$. If states $q_a$ and $q_b$ are not $k$-
distinguishable, we say that they are *$k$-equivalent*.

A distinguishing sequence for states $q_a$ and $q_b$, if used as input in the experi-
ment of Figure 4.15, would enable us to decide which copy of machine $M$
was started in state $q_a$ and which was started in $q_b$. 
Theorem 4.3: Two states of a finite-state machine are \( k \)-equivalent if and only if

1. They are 1-equivalent.
2. For each input symbol \( s \), their \( s \)-successors are \( (k - 1) \)-equivalent.

The proof of Theorem 4.3 is similar to that of Theorem 4.2, except that the input string \( \omega \) is restricted to being of length less than \( k \); we leave it to the reader to fill in the details.

Clearly, two states of a machine are equivalent if and only if they are \( k \)-equivalent for all \( k \geq 1 \).

### 4.6.3 Partitioning the State Set

Theorem 4.3 serves as the basis for the machine-reduction procedure developed below. Our objective is to partition the machine's state set into blocks of mutually equivalent states.

Some additional terminology is useful. Recall that an equivalence relation on a set partitions the set into mutually exclusive, collectively exhaustive blocks. If \( P_1 \) and \( P_2 \) are partitions of a set \( X \), and if each block of \( P_2 \) is a subset of exactly one block of \( P_1 \), then \( P_2 \) is a refinement of \( P_1 \). That is, if

\[
P_1 = \{A_1, \ldots, A_n\} \quad \text{and} \quad P_2 = \{B_1, \ldots, B_m\}
\]

are partitions of a set \( X \), and \( P_2 \) is a refinement of \( P_1 \), then for each block \( B_j \) in \( P_2 \) there is a block \( A_i \) in \( P_1 \) such that

\[
B_j \subseteq A_i
\]

The number of blocks in \( P_2 \) is never less than the number in \( P_1 \); thus, \( m \geq n \). According to this definition, a partition is a refinement of itself.

The partitioning of a state set into equivalence classes is based on the conceptual experiment depicted in Figure 4.19. Suppose that \( M = (Q, S, R, f, g, q_i) \) is a machine whose state set \( Q \) is to be partitioned into blocks of equivalent states. Let \( Q = \{q_1, \ldots, q_n\} \), and let

\[
M_i = (Q, S, R, f, g, q_i)
\]

stand for a copy of \( M \) started in \( q_i \). States \( q_i \) and \( q_j \) belong to the same block of the partition of \( Q \) just if no stimulus applied to the experiment will yield different responses from machines \( M_i \) and \( M_j \). We construct this partition of \( Q \) by forming a succession of partitions

\[
P_1, P_2, \ldots, P_m, P_{m+1}
\]

such that each block of partition \( k \), \( 1 \leq k \leq m + 1 \), contains only states that are mutually \( k \)-equivalent. In terms of Figure 4.19, states \( q_i \) and \( q_j \) will belong to the same block of partition \( P_k \) if and only if machines \( M_i \) and \( M_j \) cannot be distinguished by any stimulus of \( k \) or fewer symbols. Since states
that are \((k+1)\)-equivalent are certainly \(k\)-equivalent, each block of \(P_{k+1}\) will be contained in some block of partition \(P_k\). Hence each partition is a refinement of its predecessor. Theorem 4.3 provides the basis for constructing the sequence of partitions, according to the following partitioning algorithm:

**Step 1:** Form an initial partition \(P_1\) of \(Q\) by grouping together states that are 1-equivalent, that is, states that produce identical outputs for each input symbol. States \(q\) and \(q'\) are in the same block of \(P_1\) if and only if, for each \(s \in S\), \(g(q, s) = g(q', s)\).

**Step 2:** Obtain \(P_{k+1}\) from \(P_k\) as follows: states \(q\) and \(q'\) are in the same block of \(P_{k+1}\) if and only if

1. They are in the same block of \(P_k\).
2. For each \(s \in S\), their \(s\)-successors \(f(q, s)\) and \(f(q', s)\) are in the same block of \(P_k\).

**Step 3:** Repeat step 2 until \(P_{m+1} = P_m\) for some \(m\). We call \(P_m\) the final partition of \(Q\).

**Theorem 4.4:** There is an effective procedure for partitioning the states of a finite-state machine into blocks of equivalent states.

**Proof:** Theorem 4.3 guarantees that each partition \(P_k\) constructed during the above procedure has blocks of mutually \(k\)-equivalent
states. Also, by construction, $P_{k+1}$ is a refinement of $P_k$ for all $k \geq 1$. We shall show that the procedure must terminate with a partition $P_m$ such that $P_k = P_m$ for all $k \geq m$:

1. If $P_{k+1} = P_k$, then $P_{k+j} = P_k$, all $j \geq 0$: if $P_{k+1} = P_k$, then from step 2 of the procedure, $P_{k+2} = P_{k+1}$, and the assertion follows by induction.

2. If the machine has $n$ states, then $P_{n+1} = P_n$: for all $k \geq 1$, $P_{k+1}$ is a refinement of $P_k$. Thus the number of blocks in $P_{k+1}$ is greater than the number of blocks in $P_k$ unless $P_k$ is final. Since the number of blocks in a partition cannot exceed the number of states in $M$, partition $P_n$ must be final, and therefore $P_{n+1} = P_n$.

### 4.6.4 Example

Figure 4.20 shows the state table of a machine $M$. Applying the partitioning procedure to $M$ produces the four partitions shown, as the reader

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, 0</td>
<td>C, 0</td>
</tr>
<tr>
<td>B</td>
<td>C, 1</td>
<td>D, 1</td>
</tr>
<tr>
<td>C</td>
<td>D, 0</td>
<td>E, 0</td>
</tr>
<tr>
<td>D</td>
<td>C, 1</td>
<td>B, 1</td>
</tr>
<tr>
<td>E</td>
<td>F, 1</td>
<td>E, 1</td>
</tr>
<tr>
<td>F</td>
<td>G, 0</td>
<td>C, 0</td>
</tr>
<tr>
<td>G</td>
<td>F, 1</td>
<td>G, 1</td>
</tr>
<tr>
<td>H</td>
<td>J, 1</td>
<td>B, 0</td>
</tr>
<tr>
<td>J</td>
<td>H, 1</td>
<td>D, 0</td>
</tr>
</tbody>
</table>

$P_1$: $\{A, C, F\}, \{B, D, E, G\}, \{H, J\}$

$P_2$: $\{A, F\}, \{C\}, \{B, D, F, G\}, \{H, J\}$

$P_3$: $\{A, F\}, \{C\}, \{B, D\}, \{E, G\}, \{H, J\}$

$P_4$: $\{A\}, \{F\}, \{C\}, \{B, D\}, \{E, G\}, \{H, J\}$

Figure 4.20. Example of the partitioning procedure.
should verify. The final partition \( P_4 \) shows that \( B \sim D \), \( E \sim G \), and \( H \sim J \).

States \( A \) and \( F \), among other pairs, are distinguishable because they are in different blocks of \( P_4 \). How can we construct a distinguishing sequence for \( A \) and \( F \)? Since states \( A \) and \( F \) lie in the same block of \( P_3 \), they are not 3-distinguishable, and the shortest distinguishing sequence must be of length four. Construction of the sequence \( \omega = s(1)s(2)s(3)s(4) \) proceeds as follows:

1. States \( A \) and \( F \) are 4-distinguishable; hence \( s(1) \) must take \( A \) and \( F \) into states that are 3-distinguishable. That is, the \( s(1) \)-successors of \( A \) and \( F \) must lie in different blocks of \( P_3 \): we choose

\[
s(1) = 0 \quad \text{so that} \quad \begin{cases} 
A \xrightarrow{0/0} B \\
F \xrightarrow{0/0} G 
\end{cases}
\]

2. The \( s(2) \)-successors of \( B \) and \( G \) must be 2-distinguishable: we choose

\[
s(2) = 0 \quad \text{so that} \quad \begin{cases} 
B \xrightarrow{0/1} C \\
G \xrightarrow{0/1} F 
\end{cases}
\]

3. The \( s(3) \)-successors of \( C \) and \( F \) must be 1-distinguishable: we choose

\[
s(3) = 1 \quad \text{so that} \quad \begin{cases} 
C \xrightarrow{1/0} E \\
F \xrightarrow{1/0} C 
\end{cases}
\]

4. States \( E \) and \( C \) are 1-distinguishable, so we choose \( s(4) \) to cause different outputs. Here we may choose either \( s(4) = 0 \) or \( s(4) = 1 \):

\[
\begin{align*}
E \xrightarrow{0/1} F & \quad C \xrightarrow{0/0} D \\
E \xrightarrow{1/1} E & \quad C \xrightarrow{1/0} E
\end{align*}
\]

Therefore, the stimuli

\[
\omega = 0 \ 0 \ 1 \ 0
\]

or

\[
\omega = 0 \ 0 \ 1 \ 1
\]

are both distinguishing sequences for \( A \) and \( F \). For the case \( \omega = 0010 \), we have

\[
\begin{align*}
A & \xrightarrow{0/0} B \xrightarrow{0/1} C \xrightarrow{1/0} E \xrightarrow{0/1} F \\
F & \xrightarrow{0/0} G \xrightarrow{0/1} F \xrightarrow{1/0} C \xrightarrow{0/0} D
\end{align*}
\]

The machine responds with 0101 if initially in state \( A \), and with 0100 if initially in state \( F \). For \( \omega = 0011 \), we have

\[
\begin{align*}
A & \xrightarrow{0/0} B \xrightarrow{0/1} C \xrightarrow{1/0} E \xrightarrow{1/1} E \\
F & \xrightarrow{0/0} G \xrightarrow{0/1} F \xrightarrow{1/0} C \xrightarrow{1/0} E
\end{align*}
\]
The machine responds with 0101 if initially in state A, and with 0100 if initially in state F. We observe that the shortest length distinguishing sequence need not be unique.

### 4.6.5 Construction of a Reduced Machine

Let $M = (Q, S, R, f, g, q_i)$ be a transition-assigned machine. We want to construct an equivalent reduced machine $M' = (Q', S, R, f', g', q'_i)$. The states of $M'$ will represent the state equivalence classes of machine $M$; that is, they will correspond to the blocks of the final partition $P_f$ resulting from applying the partitioning procedure to $M$. The initial state $q'_i$ corresponds to the block containing the initial state of $M$. The state table of $M'$ is obtained by applying two rules:

1. To find the $s$-successor of a state $q'$ in $M'$, select any state in the block of the partition $P_f$ corresponding to $q'$ and find the block containing its $s$-successor; the corresponding state of $M'$ is the $s$-successor of $q'$.

2. The output for an $s$-transition from state $q'$ of $M'$ is the output for an $s$-transition from any state in the block corresponding to $q'$.

Figure 4.21 shows the reduced machine equivalent to the machine in Figure 4.20. Note that state $Z$ in the reduced machine is inaccessible when $U$ is the initial state; thus a reduced machine need not be connected, and any inaccessible states must be eliminated separately.

### 4.6.6 Isomorphism of Equivalent Machines

The following theorem expresses the uniqueness of reduced finite-state machines. This important result means that the equivalence question for the class of finite-state automata is answerable in every case by a well-defined procedure.

**Theorem 4.5:** Let $M_1$ and $M_2$ be reduced, connected finite-state machines. Then the state graphs of $M_1$ and $M_2$ are isomorphic if and only if $M_1$ and $M_2$ are equivalent.

For two state graphs to be isomorphic, we require that they be identical except for the names assigned to the states. In precise terms, the state graphs of $M_1$ and $M_2$ are isomorphic if and only if there exists a one-to-one correspondence

$$T: Q_1 \rightarrow Q_2$$

between the states of $M_1$ and the states of $M_2$ such that the transition and output functions of $M_1$ are consistent with those of $M_2$: 
(a) $P_4$:  \[
\begin{align*}
\{A\} & \quad \{F\} & \quad \{C\} & \quad \{B, D\} & \quad \{E, G\} & \quad \{H, J\} \\
U & \quad V & \quad W & \quad X & \quad Y & \quad Z
\end{align*}
\]

New Names in $M'$

(b) \[
\begin{array}{c|c|c|c}
& \text{Blocks} & 0 & 1 \\
\hline
U & \{A\} & \{B\}, 0 & \{C\}, 0 \\
V & \{F\} & \{G\}, 0 & \{C\}, 0 \\
W & \{C\} & \{D\}, 0 & \{E\}, 0 \\
X & \{B, D\} & \{C\}, 1 & \{D, B\}, 1 \\
Y & \{E, G\} & \{F\}, 1 & \{E, G\}, 1 \\
Z & \{H, J\} & \{I, H\}, 1 & \{B, D\}, 0 \\
\end{array}
\]

(c) $M'$: \[
\begin{array}{c|c|c}
& 0 & 1 \\
\hline
U & X, 0 & W, 0 \\
V & Y, 0 & W, 0 \\
W & X, 0 & Y, 0 \\
X & W, 1 & X, 1 \\
Y & V, 1 & Y, 1 \\
Z & Z, 1 & X, 0 \\
\end{array}
\]

Figure 4.21. Construction of a reduced machine.

$$T(f_1(q, s)) = f_2(T(q), s) \text{ all } q \in Q_1$$

$$g_1(q, s) = g_2(T(q), s) \text{ all } s \in S$$

$$T(q_{t1}) = q_{t2}$$

From Theorem 4.5 we have a procedure for testing the equivalence of
two finite-state machines. We remove inaccessible states and then find reduced machines \( M_1 \) and \( M_2 \) from the given machines; the given machines are equivalent if and only if machines \( M_1 \) and \( M_2 \) are equivalent. By Theorem 4.5, this is the case if and only if the state diagrams of \( M_1 \) and \( M_2 \) are isomorphic.

For the proof of Theorem 4.5 it is convenient to extend the notion of equivalence to encompass states in different machines.

**Definition 4.11:** Let \( M_1 \) and \( M_2 \) be finite-state machines, and assume the states are so named that \( Q_1 \cap Q_2 = \emptyset \). The direct-sum machine for \( M_1 \) and \( M_2 \) is the result of considering the state graphs of \( M_1 \) and \( M_2 \) as constituting a single machine with state set \( Q = Q_1 \cup Q_2 \). The initial state of the direct-sum machine is unspecified. A state \( q \) in \( M_1 \) is equivalent to a state \( q' \) in \( M_2 \) just if \( q \) and \( q' \) are equivalent in the direct-sum machine.

**Proof of Theorem 4.5**

*Only if:* If the state diagrams of \( M_1 \) and \( M_2 \) are isomorphic, it is obvious that \( M_1 \sim M_2 \).

*If:* We must show that, if \( M_1 \sim M_2 \) and both machines are reduced and connected, then their state graphs are isomorphic. To do this, we shall construct the required isomorphism \( T \): \( Q_1 \to Q_2 \). We apply the partitioning procedure to obtain a final partition \( P_f \) of the direct-sum state set \( Q_1 \cup Q_2 \). We claim that each block of \( P_f \) is a pair \((q, q')\) in which \( q \in Q_1 \), and \( q' \in Q_2 \):

First, no block of \( P_f \) can contain more than one state from \( Q_1 \) because this would contradict the assertion that \( M_1 \) is reduced. Similarly, no block of \( P_f \) can contain more than one state from \( Q_2 \). Hence each block of \( P_f \) contains at most two states.

Second, no block of \( P_f \) contains exactly one state. To see this, suppose that some block of \( P_f \) contains exactly one state, say \( q \in Q_1 \). Since \( M_1 \) is connected, there exists an input sequence \( \omega \) for which

\[ q_{11} \xrightarrow{\omega} q \]

This same sequence, applied to \( M_2 \), must leave \( M_2 \) in some state \( q' \):

\[ q_{12} \xrightarrow{\omega} q' \]

Since \( q \) is assumed to be alone in a block of \( P_f \), it must be distinguishable from \( q' \), and there is a distinguishing sequence \( \varphi \) for \( q \) and \( q' \). But then the sequence \( \omega \varphi \) will distinguish \( q_{11} \) and \( q_{12} \), contradicting the assertion \( M_1 \sim M_2 \).

Thus the final partition \( P_f \) of \( Q_1 \cup Q_2 \) is a set of pairs of corresponding equivalent states. This establishes a one-to-one correspon-
dence $T: Q_1 \rightarrow Q_2$. The equivalence of each pair of states in $P_f$, together with Theorem 4.2, shows that the state graphs of $M_1$ and $M_2$ are isomorphic.

### 4.6.7 Machine Containment

The proof of Theorem 4.5 suggests that it is not necessary to construct reduced machines in order to test for equivalence.

**Definition 4.12:** Let $M_1$ and $M_2$ be finite-state machines. If each state in $M_1$ is equivalent to a state in $M_2$, we say that $M_2$ contains $M_1$.

If $M_2$ contains $M_1$, then for a given initial state of $M_1$ we may choose an initial state for $M_2$ that causes $M_2$ to mimic the behavior of $M_1$. If $M_1$ contains $M_2$ as well, then the two machines are equivalent whenever they are initialized to equivalent states.

If we apply the partitioning procedure to the direct-sum machine for $M_1$ and $M_2$, the final partition will group states that are equivalent in the two machines. If each block of this partition contains at least one state from $M_1$, then $M_1$ contains $M_2$. If each block of this partition contains at least one state from $M_2$, then $M_2$ contains $M_1$. If both these statements are true, then $M_1$ and $M_2$ are equivalent whenever their initial states are equivalent, and a reduced machine equivalent to both $M_1$ and $M_2$ can be obtained using the blocks of the final partition.

Thus the following is a procedure for deciding whether $M_1 \sim M_2$ and, if so, for obtaining a reduced machine $M$ equivalent to $M_1$ and $M_2$:

1. Eliminate states of $M_1$ and $M_2$ inaccessible from their respective initial states.
2. Partition the state set of the direct-sum machine. Let $P_f$ denote the final partition.
3. Ascertain whether each block of $P_f$ contains at least one state from each of $M_1$ and $M_2$. If so, $M_1 \sim M_2$ just if the initial states of $M_1$ and $M_2$ are in the same block of $P_f$.
4. If $M_1 \sim M_2$, an equivalent reduced machine $M$, whose states correspond to the blocks of $P_f$, is given by the construction of Section 4.6.5.

**Example 4.8:** Figure 4.22 shows the state tables of two machines $M_1$ and $M_2$. The state table of the direct-sum machine is given, and the partitioning procedure yields the two partitions shown. We see that

$$A \sim D, \quad B \sim E, \quad \text{and} \quad C \sim F$$
(a) $M_1$:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, 0</td>
<td>A, 0</td>
</tr>
<tr>
<td>B</td>
<td>A, 0</td>
<td>C, 1</td>
</tr>
<tr>
<td>C</td>
<td>C, 1</td>
<td>A, 0</td>
</tr>
</tbody>
</table>

(b) $M_2$:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>E, 0</td>
<td>D, 0</td>
</tr>
<tr>
<td>E</td>
<td>D, 0</td>
<td>F, 1</td>
</tr>
<tr>
<td>F</td>
<td>F, 1</td>
<td>D, 0</td>
</tr>
<tr>
<td>G</td>
<td>E, 0</td>
<td>H, 1</td>
</tr>
<tr>
<td>H</td>
<td>D, 1</td>
<td>G, 0</td>
</tr>
</tbody>
</table>

(c) Sum of $M_1$ and $M_2$:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, 0</td>
<td>A, 0</td>
</tr>
<tr>
<td>B</td>
<td>A, 0</td>
<td>C, 1</td>
</tr>
<tr>
<td>C</td>
<td>C, 1</td>
<td>A, 0</td>
</tr>
<tr>
<td>D</td>
<td>E, 0</td>
<td>D, 0</td>
</tr>
<tr>
<td>E</td>
<td>D, 0</td>
<td>F, 1</td>
</tr>
<tr>
<td>F</td>
<td>F, 1</td>
<td>D, 0</td>
</tr>
<tr>
<td>G</td>
<td>E, 0</td>
<td>H, 1</td>
</tr>
<tr>
<td>H</td>
<td>D, 1</td>
<td>G, 0</td>
</tr>
</tbody>
</table>

$P_1$: \{A, D\}, \{B, E, G\}, \{C, F, H\}

$P_2$: \{A, D\}, \{B, E\}, \{C\}, \{C, F\}, \{H\}

$P_3 = P_2$

Figure 4.22. Testing two machines for equivalence.

Hence $M_2$ contains $M_1$. States G and H of $M_2$ are not equivalent to any states of $M_1$, so $M_1$ does not contain $M_2$, and it appears that the two machines are not equivalent. However, there is no way in $M_2$ to get from state D, E, or F to either of states G and H. If D, E, or F is specified as the starting state of $M_2$, then states G and H are inaccessible and may be eliminated from the machine. If, correspondingly, A, B, or C is made the initial state of $M_1$, the two machines will be equivalent.
4.7 Machine Histories and Finite Memory

We conclude this chapter with an alternative derivation of the finite-state machine. In Section 4.1, we obtained the model by assuming that the machine comprises a finite number of interconnected parts, each capable of assuming a finite number of distinct states. We consider now another finiteness assumption, that of finite memory, and show that it leads to exactly the same machine model.

Suppose that we know these facts about a machine M:

1. It is deterministic.
2. It operates at discrete instants in time, at each instant receiving a symbol from an input alphabet S and emitting a symbol from an output alphabet R.

Since M is deterministic, its behavior is uniquely determined by its input string, which may therefore be regarded as representing M's “history.” Input strings $\omega$ and $\varphi$ are said to represent equivalent histories of M if it is impossible to distinguish, based on its response to a future stimulus, a copy of M that has received the string $\omega$ from a copy that has received the string $\varphi$.

The notion of equivalent histories can be used to define a relation on the input universe $S^*$: two input strings are equivalent with respect to M's future behavior just if they represent equivalent histories of M. The reader may verify that this relation is reflexive, symmetric, and transitive, and therefore partitions $S^*$ into equivalence classes, which we refer to as classes of machine histories or simply history classes. These history classes are the basis of our second development of the finite-state machine model.

By saying a machine has finite memory, we mean that it is capable of sorting its input strings into only a finite number of distinct history classes (like the postal clerk who tosses all letters into a fixed number of bins regardless of the variety of addresses). Now suppose that we know, in addition to the two facts stated earlier, that M has finite memory. Let the corresponding set of history classes be

$$H = \{H_1, H_2, \ldots, H_n\}$$

These nonempty classes, being a partition of $S^*$, are mutually disjoint and collectively exhaustive:

$$H_i \cap H_j = \emptyset, \quad i \neq j$$

$$\bigcup_{i=1}^{n} H_i = S^*$$

That is, each string in $S^*$ belongs to precisely one class in H.
Consider two strings \( \omega \) and \( \phi \) that are in the same history class of \( S^* \), say \( H_j \). For any symbol \( s \) in \( S \), the strings \( \omega s \) and \( \phi s \) must also be in the same history class of \( S^* \); if \( \omega s \) and \( \phi s \) were in different history classes, some input string (that is, the single symbol \( s \)) would allow us to determine whether \( M \) has initially received \( \omega \) or \( \phi \), contradicting the assumption that \( \omega \) and \( \phi \) are together in \( H_j \). Therefore, for each input symbol \( s \) and each history class \( H_j \), there exists a unique class \( H_j \) such that

\[
H_j s = \{ \omega s \mid \omega \in H_j \} \subseteq H_j
\]

and thus there exists a function

\[
f : H \times S \rightarrow H
\]

defined by

\[
f(H_j, s) = H_j \quad \text{just if} \quad H_j s \subseteq H_j
\]

Since \( f \) specifies the manner in which input symbols cause transitions of input strings from one history class to another, we refer to \( f \) as the transition function of \( M \).

Because \( M \) is a deterministic machine, its output is determined solely by its input. In particular, the response of \( M \) to an input symbol \( s \) depends only on \( s \) and the sequence \( \omega \) of symbols presented previously to \( M \). But the sequence \( \omega \) cannot be distinguished, on the basis of \( M \)'s response to \( s \) (or any other input symbol), from the other members of its history class. Thus there exists a function

\[
g : H \times S \rightarrow R
\]

that specifies \( M \)'s response to a symbol in terms of that symbol and the history class of \( M \)'s previous input; we refer to \( g \) as the output function of \( M \).

In summary, an automaton with finite memory is a five-tuple \( M = (H, S, R, f, g) \), where

- \( H \) is a finite set of history classes
- \( S \) is a finite input alphabet
- \( R \) is a finite output alphabet
- \( f : H \times S \rightarrow H \) is a transition function
- \( g : H \times S \rightarrow R \) is an output function

Note that the application of an input string to \( M \) leaves \( M \) in a state of existence that identifies the history class of the input. If we consider the classes in \( H \) as representing "states" of the machine \( M \), the definition above becomes identical to Definition 4.1, except that it does not specify explicitly an initial state. But the initial state of \( M \) is the state of \( M \) prior to the presentation of any input; that is, it is the state of the machine corresponding to the history
class of the empty string. If we specify this state explicitly as a sixth component of the tuple, the definition above becomes identical to Definition 4.1. The finite-state model of this chapter can be used, therefore, to describe any model characterized by "finite memory."

Notes and References

The notion of a finite-state device is usually attributed to McCulloch and Pitts [1943], who developed a model of "nerve nets" to aid in describing the neurological behavior of the brain. The formalisms used in this chapter were developed more than a decade later, the transition-assigned machine by G. H. Mealy in 1955, the state-assigned machine by E. F. Moore in 1956. The contributions of Moore are of particular interest to us here, since he demonstrated that the reduced form of a state-assigned machine is unique up to a labeling of states (an equivalent result follows from the study by Huffman [1954] of sequential switching circuits), and described a partitioning procedure for obtaining a reduced machine from an arbitrary state-assigned device. His procedure is essentially that described in Section 4.6. The applicability of these results to transition-assigned machines followed from a demonstration of the equivalence of the Moore and Mealy models (see, for example, Ibarra [1967]).

The characterization of finite-state machines in terms of equivalence classes is due to Myhill [1957] and Nerode [1959]. Discussions of it on our level are found in Minsky [1967] and Hopcroft and Ullman [1969]. (Minsky's work contains detailed discussions of many aspects of the theory of finite automata, and provides useful supplementary reading for this and the next three chapters.) The characterization is discussed on an abstract level by Arbib [1969] and Gunzberg [1968].

Problems

4.1. Present the state diagram of a transition-assigned machine \( M = (Q, S, R, f, g, q_0) \) with \( S = R = \{0, 1, 2, 3\} \) and the following behavior: for \( t > 2 \), \( r(t) = m(t) + n(t) \), where

\[
m(t) = \begin{cases} 
2 & \text{if } s(t-1) \text{ is 0 or 2} \\
0 & \text{otherwise}
\end{cases}
\]

\[
n(t) = \begin{cases} 
1 & \text{if } s(t-2) \text{ is 1 or 3} \\
0 & \text{otherwise}
\end{cases}
\]

Define \( r(1) \) and \( r(2) \) as if \( s(-1) = s(0) = 0 \).
4.2. Let $S = \{a, b, c\}$ and define, for each symbol in $S$ and any string $\omega$ in $S^*$,

$$N_s(\omega) = \text{number of occurrences of symbol } s \text{ in } \omega$$

a. Display the state table of a transition-assigned machine $M = (Q, S, R, f, g, q_i)$ whose last output symbol in response to an input string $\omega$ is

$$r = (N_a(\omega) + 2N_b(\omega) - 3N_c(\omega)) \text{ mod } 5$$

(The set $R$ will be $\{0, 1, 2, 3, 4\}$.) A typical input–output pair might be

Input: $a$ $b$ $b$ $c$ $c$ $b$ $a$ $a$ $b$ $c$
Output: $1$ $3$ $0$ $2$ $4$ $1$ $2$ $3$ $0$ $2$

b. Suppose that we ask now that the last output symbol be

$$r = N_a(\omega) + 2N_b(\omega) - 3N_c(\omega)$$

Is it still possible to construct a machine $M$ with the appropriate behavior? Suppose that the last output symbol is to be

$$r = \min(100, N_a(\omega) + 2N_b(\omega) - 3N_c(\omega))$$

Is it possible in this case to construct the machine $M$?

4.3. Design a Mealy automaton $M = (Q, S, R, f, g, q_i)$ with $S = R = \{0, 1\}$ and the following output behavior:

a. $r(1) = r(2) = 0$. For $t > 2$,

$$r(t) = 1 \iff ((s(t) = 0) \land (s(t - 1) = 1) \land (s(t - 2) = 1))$$

b. $r(1) = r(2) = 0$. For $t > 2$,

$$r(t) = 1 \iff (((s(t) = 0) \land (s(t - 1) = s(t - 2) = 1))$$

$$\lor ((s(t) = 1) \land (s(t - 1) = 0 \lor s(t - 2) = 0)))$$

4.4. Describe informally the input–output behavior and the form of the state diagram of a Mealy $n$-unit delay, for arbitrary $n$. (A 2-unit delay was constructed in Example 4.3.)

4.5. Let $M = (Q, S, R, f, h, q_i)$ be a Moore machine with $S = R = \{0, 1\}$. An input sequence $\omega$ is said to be accepted by $M$ just if $M$'s response to $\omega$ ends with 1. Present the state diagram of a machine $M$ that accepts
a. All strings $\omega \in S^*$ in which the substring 101 appears.
b. All strings $\omega \in S^*$ in which the substring 101 does not appear.
c. All strings $\omega \in S^*$ in which

$$(N_0(\omega) - N_1(\omega)) \mod 4 = 0$$

(See Problem 4.2.)

4.6. Let $S = R = \{0, 1\}$, and let $\omega$ be a binary string. We say that $\omega$ has property $A$ if each occurrence of the symbol 0 in $\omega$ is followed immediately by another 0 or by a string of at least two consecutive 1's. We say that $\omega$ has property $B$ if the number of occurrences of the symbol 1 in $\omega$ is odd. Present the state table of a Moore machine $M = (Q, S, R, f, h, q_i)$ that accepts (see Problem 4.5) a string $\omega$ if and only if $\omega$ has

a. Property $A$.
b. Property $B$.
c. Both properties $A$ and $B$.
d. Either property $A$ or property $B$ or both.
e. Either property $A$ or property $B$, but not both.
f. Neither property $A$ nor property $B$.

Construct a transition-assigned machine similar (in the sense of Definition 4.5) to the following state-assigned machine:

\[
\begin{array}{c|ccc}
0 & 1 \\
\hline
A & B & C & 0 \\
B & C & B & 1 \\
C & A & C & 0 \\
\end{array}
\]

Construct a state-assigned machine similar to the following transition-assigned machine:

\[
\begin{array}{c|cc}
0 & 1 \\
\hline
A & B & 0 & C & 1 \\
B & C & 1 & B & 1 \\
C & A & 1 & C & 0 \\
\end{array}
\]
4.8. Find a Moore machine that accepts (see Problem 4.5) just those strings accepted by the following Mealy machine:

\[
\begin{array}{c|cc}
0 & 1 \\
\hline
A & A & 1 \\
B & C & 1 \\
C & B & 0 \\
D & D & 1
\end{array}
\]

Describe the set of strings accepted.

4.9. Consider a machine partially specified by the following table:

\[
\begin{array}{c|cc}
0 & 1 \\
\hline
A & B & 0 \\
B & A & 1 \\
C & C & 0 \\
D & E & 1 \\
E & A & 0
\end{array}
\]

It is known that starting in state A the machine exhibits a response ending in 0 to the input sequence 0110. Following this, application of the sequence 101 results in a response again ending in 0. What can be told about the missing entry?

4.10. The following input–output behavior was exhibited by a transition-assigned machine M known to contain three states. Find an appropriate state table for M. Is the table unique?

Input 0 0 0 0 1 0 0 0 1 0 0 0 1 0
Output 0 1 0 1 0 0 0 1 0 1 0 0 1

Repeat the above for the following input–output behavior, given that M is a transition-assigned machine with at most three states:

Input 1 1 1 0 1 1 1 0 1 0 1 0 0 0 1 0 0 1 1 0
Output 0 1 0 1 1 0 1 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 1
4.11. a. A certain four-state machine exhibits the following input–output behavior:

<table>
<thead>
<tr>
<th>Input</th>
<th>0 1 0 1 0 1 0 1 0 1 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0 0 0 0 0 0 0 0 0 0 0 1</td>
</tr>
</tbody>
</table>

Prove that the state assumed by the machine at the point marked "*" must have been visited earlier in the sequence.

b. Let \( M \) be a three-state machine about which nothing is known except that its input alphabet contains symbols 0 and 1. Let \( q \) denote the state of \( M \) after presentation of the input sequence \((01)^{100}\). What can be said about the state of \( M \) after presentation of the input sequence \((01)^{400}\)?

4.12. Let \( M \) be a finite-state machine with state set \( Q \). Suppose that \( Q \) contains \( n \) states, including initial state \( q_i \). For any state \( q \in Q \), prove that \( q \) is accessible in \( M \) only if \( q \) is the \( \omega \)-successor of \( q_i \) for some string \( \omega \) of length less than \( n \).

4.13. We say that a finite-state machine is strongly connected if every state in the machine is accessible from every other state in the machine. Let \( M \) be an \( n \)-state, reduced, strongly connected, finite-state machine. Prove there exists an input string \( \omega, |\omega| \leq n(n - 1)/2 \), such that \( M \) assumes each of its states at least once in response to \( \omega \). (Hint: See Problem 4.12.)

4.14. A \((q, r, s)\)-machine is a transition-assigned machine \( M = (Q, S, R, f, g, q_i) \) in which

\[
\begin{align*}
\bar{Q} &= q \\
\bar{R} &= r \\
\bar{S} &= s
\end{align*}
\]

Let \( \mathcal{E}(q, r, s) \) denote the class of \((q, r, s)\)-machines. Show that

\[
\mathcal{E}(q, r, s) = (rg)^a
\]

A simply minimal \((q, r, s)\)-machine is a \((q, r, s)\)-machine in which, for all \( i, j \) \( (i \neq j) \), there exists at least one \( k \) such that

\[
g(q_i, s_k) \neq g(q_j, s_k)
\]

Let \( \mathcal{E}'(q, r, s) \) be the class of simply minimal \((q, r, s)\)-machines. Show that \( \mathcal{E}'(q, r, s) = q^a(r^* - 1)(r^* - 2) \ldots (r^* - n + 1) \). Does any simply minimal \((q, r, s)\)-machine contain a pair of equivalent states?
4.15. Consider the following machine $M_1$:

$$
\begin{array}{c|cc}
0 & 1 \\
A & C & 0 \\
B & E & 0 \\
C & A & 0 \\
D & G & 0 \\
E & F & 1 \\
F & E & 0 \\
G & D & 0 \\
\end{array}
$$

Find a minimal length distinguishing sequence for states C and G. Is the sequence unique? Is there a sequence that distinguishes C from G and also distinguishes A from D?

4.16. Consider the following machines $M_1$ and $M_2$:

$$
\begin{array}{c|cc}
0 & 1 \\
A & B & A & 0 \\
B & C & D & 0 \\
C & E & C & 0 \\
D & F & B & 0 \\
E & G & E & 0 \\
F & H & F & 0 \\
G & I & G & 0 \\
II & J & II & 0 \\
I & A & K & 1 \\
J & K & J & 0 \\
K & A & K & 1 \\
\end{array}
\begin{array}{c|cc}
0 & 1 \\
A & B & A & 0 \\
B & C & B & 0 \\
C & D & C & 0 \\
D & E & D & 0 \\
E & F & E & 0 \\
F & B & F & 1 \\
\end{array}
$$
a. Reduce $M_1$. Find a minimal length distinguishing sequence for states $A$ and $B$ in $M_1$.
b. Reduce $M_2$. Find a minimal length distinguishing sequence for states $A$ and $B$ in $M_2$.
c. $M_1$ and $M_2$ are started in their respective states $A$. Find a minimal length distinguishing sequence for machines $M_1$ and $M_2$.
d. In parts a and b the lengths of the distinguishing sequences were less than the number of states in the respective reduced machines. In part c, however, a sequence is needed that has a length greater than the number of states in either reduced machine. Why?

4.17. Let $M$ be the following machine, and let $A$ be its initial state:

$$
\begin{array}{c|cc}
 & 0 & 1 \\
\hline
A & B & E \\
B & C & D \\
C & B & A \\
D & B & B \\
E & B & A \\
F & G & G \\
G & F & H \\
H & I & G \\
I & G & G \\
\end{array}
$$

Give the state table of a reduced, connected, Moore machine equivalent to $M$. Give the state table of a reduced, connected, Mealy machine similar to $M$.

4.18. Consider the following machines $M_1$ and $M_2$. $M_1$ has initial state $A$, and the initial state of $M_2$ is unspecified. Can the machines be made equivalent by the correct choice of initial state for $M_2$? If so, which state(s) can be chosen? (Establish the results by partitioning the states of the direct-sum machine.)
4.19. Write out the proof of Theorem 4.3.

4.20. Consider the following machine $M_2$ with unspecified initial state.

\[ M_2: \]

\[
\begin{array}{cc}
0 & 1 \\
A & A 0 & C 0 \\
B & D 0 & E 1 \\
C & B 0 & C 0 \\
D & A 1 & B 0 \\
E & C 1 & D 0 \\
\end{array}
\]

a. Show that by observing only the output associated with the input sequence 011 it is possible to determine which state the machine is
in after application of the sequence, without knowing its initial state. Such an input sequence is known as a homing sequence.

b. Devise an effective procedure to determine if a particular sequence is a homing sequence for an arbitrary finite-state machine. Use your method to find all homing sequences of length three for $M_2$ above.

*c. Prove that for every reduced finite-state machine $M$ there exists some homing sequence for $M$. (Hint: Suppose that $M$ is a finite-state machine with $n$ states. Show that, if $M$ has no homing sequence of length less than $n^2$, $M$ must have a pair of equivalent states.)

For a detailed discussion of homing sequences and related topics the reader is referred to Hennie [1968].

4.21. Prove that for any integer $k \geq 1$, there exist two state-assigned machines with $k$ states that are indistinguishable by all input strings of length less than $2k - 2$.

4.22. Let $S$ be a set of finite-state machines. If we were to apply simultaneously all input sequences of length $n$ to each machine in the set, recording all responses, we would say that we had performed an experiment of length $n$ on $S$.

a. Let $M$ be an $n$-state Mealy machine and, for any state $q$ in $M$, let $(M, q)$ denote a copy of $M$ started in state $q$. Prove that, for any states $q_i$ and $q_j$ in $M$, $(M, q_i)$ and $(M, q_j)$ can be distinguished by an experiment of length less than $n$ if they can be distinguished at all, that is, if they are not equivalent.

*b. Let $M_1, \ldots, M_k$ be finite-state machines, and let $n_1, \ldots, n_k$ be the cardinalities of their respective state sets. Find an upper bound on the maximum length of the shortest experiment that distinguishes all the machines, given that the machines are mutually distinguishable.

4.23. $M$ is a partially specified finite-state machine. Note that the transition caused by an input of 0 while $M$ is in state $E$ is unspecified.

\[\begin{array}{c|cc}
M: & 0 & 1 \\
\hline
A & B & 0 & C & 1 \\
B & A & 0 & D & 0 \\
C & E & 1 & A & 0 \\
D & B & 0 & E & 1 \\
E & - & D & 0 \\
\end{array}\]
An input sequence is applicable to state \( q \) if it does not cause \( M \), started in state \( q \), to go through an unspecified transition. We say that a state \( p \) contains a state \( q \) just if

1. Any sequence applicable to \( q \) is applicable to \( p \).
2. The response of \( M \) to a sequence applicable to \( q \) is the same whether \( M \) is started in \( q \) or in \( p \).

a. Modify the partitioning procedure to obtain a procedure for determining state containment, rather than state equivalence. Is state \( E \) of the machine above contained by any other state?

b. Prove that two states in a finite-state machine are equivalent if and only if they contain each other.

c. Prove that state containment is a transitive relation; that is, prove that \( p \) contains \( q \) and \( q \) contains \( r \) only if \( p \) contains \( r \).

d. One way to complete a machine with unspecified transitions is to replace an unspecified transition for any state \( q \) with the corresponding transition of a state containing \( q \). Use this technique to complete the specification of \( M \). Must this specified machine be equivalent to the original unspecified machine?

e. Repeat part d for machine \( M' \):

\[
M':
\begin{array}{c|cc}
0 & 1 \\
\hline
A & B & C \\
B & A & D \\
C & C & D \\
D & D & -
\end{array}
\]

\[4.24\] Let \( M_1 = (Q_1, S_1, R_1, f_1, g_1, q_{1f}) \) and \( M_2 = (Q_2, S_2, R_2, f_2, g_2, q_{2f}) \) be finite-state machines such that \( R_1 \subseteq S_2 \). The cascade machine \( M_1 \cdot M_2 \) is diagrammed as follows:

That is, the output of \( M_1 \) is used as input to \( M_2 \), and the output of \( M_2 \) is the output of \( M_1 \cdot M_2 \).

a. Show that \( M_1 \cdot M_2 \) is a finite-state machine by specifying the six-tuple

\[
M_1 \cdot M_2 = (Q, S, R, f, g, q_f)
\]
b. Given that $M_1$ and $M_2$ have $n_1$ and $n_2$ states, respectively, how many states are in $M_1 \cdot M_2$?

Let $M_1$ and $M_2$ be as follows:

\[
\begin{array}{c|cc}
M_1: & 0 & 1 \\
\hline
A & B & 1 \\
B & 0 & B \\
\end{array}
\quad
\begin{array}{c|cc}
M_2: & 0 & 1 \\
\hline
C & D & 0 \\
D & E & 1 \\
E & 0 & E \\
\end{array}
\]

c. Viewed as language accepters (see Problem 4.5), what strings are accepted by $M_1$? by $M_2$? by the cascade machine $M_1 \cdot M_2$?

d. Let $M_1$ and $M_2$ be arbitrary finite-state machines with input alphabets and output alphabets of $\{0, 1\}$. Might it be true that $M_1 \cdot M_2 \sim M_2 \cdot M_1$? Must it be true that $M_1 \cdot M_2 \sim M_2 \cdot M_1$?

The technique of combining machines to form a cascade machine has many useful applications. In later chapters, this technique is used in establishing closure properties of regular and context-free languages.

4.25. Let $M_1 = (Q_1, S, \{0, 1\}, f_1, h_1, q_{11})$ and $M_2 = (Q_2, S, \{0, 1\}, f_2, h_2, q_{12})$ be Moore machines, and consider the following parallel machine $M_1 \cup M_2$:

![Parallel Machine Diagram]

The output $r$ of the parallel machine is the Boolean sum of $r_1$ and $r_2$.

a. Show that $M_1 \cup M_2$ is a finite-state machine by specifying the six-tuple $M_1 \cup M_2 = (Q, S, R, f, h, q_t)$

b. Viewing $M_1$ and $M_2$ as accepters (see Problem 4.5), let $L(M_1)$ denote the strings accepted by $M_1$, and let $L(M_2)$ denote the strings accepted by $M_2$. In terms of $L(M_1)$ and $L(M_2)$, what strings are accepted by the parallel machine above?
c. Suppose that we replace the ∪-module shown in the figure with a ∩-module, indicating that \( r \) is the Boolean product of \( r_1 \) and \( r_2 \), rather than the Boolean sum. In terms of \( L(M_1) \) and \( L(M_2) \), what strings are accepted by the parallel machine?

Like the cascade machines of Problem 4.24, parallel machines are often a useful tool in the study of language closure properties.

4.26. Any finite-state machine \( M \) may be regarded as a mapping \( M : S^* \rightarrow R^* \), defined by \( M(\omega) = \varphi \), where \( \varphi \) is the response of \( M \) to the input sequence \( \omega \). A machine \( M \) is information lossless if, for each output string, it is possible to determine uniquely the corresponding input string.

a. What does information lossless mean in terms of the map \( M : S^* \rightarrow R^* \)? In terms of the state diagram of \( M \)?

b. Consider \( M_1, M_2 \) as follows:

\[
\begin{array}{c|cc}
& 0 & 1 \\
\hline
A & A & 0 \\
B & C & 0 \\
C & C & 1 \\
D & D & 0 \\
\end{array}
\quad
\begin{array}{c|cc}
& 0 & 1 \\
\hline
A & B & 1 \\
B & A & 0 \\
C & D & 0 \\
D & C & 1 \\
\end{array}
\]

Is \( M_1 \) information lossless? Is \( M_2 \) information lossless?

For a machine \( M \) that is information lossless, it is possible to construct an inverse machine \( M^{-1} \) that responds with \( \varphi \) to input \( \omega \) if and only if \( M(\varphi) = \omega \).

c. For each information lossless machine above, construct the inverse machine.

d. Generalize the above procedure; that is, given an arbitrary information lossless machine \( M \), describe a procedure for constructing \( M^{-1} \).

For a detailed discussion of information lossless machines, the reader is referred to Hennie [1968]. The algebraic theory of these machines is treated in depth by Kurmit [1974].

4.27. Let \( M \) be a finite-state machine with input alphabet \( S \). Let \( \sim \) be the equivalence relation on \( S^* \) defined in Section 4.7. For each \( \omega \in S^* \), let \( [\omega] = \{ \varphi \mid \varphi \sim \omega \} \), and let \( T(M) = \{ [\omega] \mid \omega \in S^* \} \).

a. Show that

\[
\alpha \in [\omega] \quad \text{and} \quad \beta \in [\varphi] \Rightarrow \alpha\beta \in [\omega\varphi]
\]
Let this motivate a definition of multiplication \( m \):
\[
m([\alpha], [\beta]) = [\alpha \beta]
\]
b. Prove that \( \overline{T(M)} \subseteq \overline{Q} \).
c. Prove that \( \overline{T(M)} = \overline{Q} \) if and only if \( M \) is reduced and connected.
d. Prove that \( (T(M), m) \) is a semigroup.† Does it have an identity?

The set \( T(M) \) is known as the semigroup of \( M \). Note that each finite-state machine has a semigroup associated with it in this way.

4.28. A **semiautomaton** is a triple \( M = (Q, S, F) \) in which \( Q \) is a finite state set, \( S \) is a finite input alphabet, and \( F = \{ f_s | f_s : Q \to Q, s \in S \} \) is a set of transition functions, each function \( f_s \) representing the transitions of \( M \) under symbol \( s \) (that is, \( f_s(q) = q' \) just if \( q' \) is the \( s \)-successor of \( q \) in \( M \)). Thus a semiautomaton is a finite-state machine with no specified outputs or initial state.

We may extend the transition functions from symbols to strings of symbols as follows:

1. \( f_s \) is the identity function.
2. \( f_{rs} = f_r \circ f_s \) for any \( r, s \in S \), where \( \circ \) denotes functional composition (see Problem 2.20).
3. \( f_{s\omega} = f_s \circ f_\omega \), for \( s \in S, \omega \in S^* \).

Let \( T(M) = \{ f_\omega | \omega \in S^* \} \) be the set of extended transition mappings.
a. Show that \( T(M) \) is finite.
Suppose that we define a multiplication \( m \) to be
\[
m(f_\omega, f_\rho) = f_{\omega \rho}
\]
b. Prove that \( (T(M), m) \) is a semigroup with identity. (The set \( T(M) \) is known as the semigroup of the semiautomaton \( M \).)

*c. Let \( (X, m) \) be any finite semigroup with identity. Describe a semiautomaton \( M_x \) that has semigroup \( X \). [Hint: \( M_x \) is of the form \( M = (X, X, F_X) \).]

The ideas of this problem are the starting point of the algebraic theory of automata. The beginning of the theory for finite-state automata is found in Hartmanis and Stearns [1966]. The complete theory may be found in the monograph by Ginzburg [1968].

†A **binary operation** on a set \( S \) is a function \( f : S \times S \to S \). Such an operation is **associative** if, for all \( x, y, z \) in \( S \), \( f(x, f(y, z)) = f(f(x, y), z) \). A **semigroup** is an ordered pair \( (S, m) \), where \( S \) is a set and \( m \) is an associative binary operation on \( S \), often referred to as a **multiplication** in \( S \). If \( S \) contains an element \( e \) such that \( m(e, s) = m(s, e) = s \) for each \( s \in S \), then \( e \) is an identity element and \( (S, m) \) is a semigroup with identity.

Arbib [1968, 1969] provides a wealth of material on semigroups, particularly as they relate to abstract machines.
4.29. An equivalence relation $\rho$ on a set $A$ is said to be right-invariant if, for all $x \in A$, $\alpha p \beta \Rightarrow (\alpha x)p(\beta x)$.

a. Let $[\alpha]$ denote the equivalence class of $\alpha$ with respect to the equivalence relation $\rho$. Show that $\rho$ right-invariant and $\alpha \in [\omega]$ implies $\alpha x \in [\alpha x]$, any $x \in A$.

b. Let $M$ be an arbitrary finite-state machine, $S$ its input alphabet, and $\sim$ the equivalence relation on $S$ defined in Section 4.7. Show that this relation is right-invariant.

c. Let $\rho$ be any right-invariant equivalence relation of finite index (that is, with finitely many equivalence classes) on $S^*$, and let $X$ be its set of equivalence classes. Show that $X$ is a semigroup with identity for an appropriately chosen multiplication. Describe the construction of a finite-state machine with semigroup $X$ (See Problem 4.27.)