FINITE STATE MACHINES 8

8.1 INTRODUCTION

At a low level of abstraction, a protocol is often most easily understood as a state machine. Design criteria can also easily be expressed in terms of desirable or undesirable protocol states and state transitions. In a way, the protocol state symbolizes the assumptions that each process in the system makes about the others. It defines what actions a process is allowed to take, which events it expects to happen, and how it will respond to those events.

The formal model of a communicating finite state machine plays an important role in three different areas of protocol design: formal validation, protocol synthesis, and conformance testing. This chapter introduces the main concepts. First the basic finite state machine model is discussed. There are several, equally valid, ways of extending this basic model into a model for communicating finite state machines. We select one of those models and formalize it in a definition of a generalized communicating finite state machine. The model can readily be applied to represent PROMELA specifications and to build an automated validator.

There exist many variations of the basic finite state machine model. Rather than list them all, we conclude this chapter with a discussion of two of the more interesting examples: the Petri Net and the FIFO Net.

8.2 INFORMAL DESCRIPTION

A finite state machine is usually specified in the form of a transition table, much like the one shown in Table 8.1 below.

Table 8.1 — Mealy[1]

Condition		Effect	
Current State	In	Out	Next State
q0	–	1	q2
q1	–	0	q0
q2	0	0	q3
q2	1	0	q1
q3	0	0	q0
q3	1	0	q1

For each control state of the machine the table specifies a set of transition rules. There is one rule per row in the table, and usually more than one rule per state. The example table contains transition rules for control states named q0, q1, q2, and q3. Each transition rule has four parts, each part corresponding to one of the four columns in the table. The first two are conditions that must be satisfied for the transition rule to be executable. They specify

- The control state in which the machine must be
- A condition on the ‘‘environment’’ of the machine, such as the value of an input signal

The last two columns of the table define the effect of the application of a transition rule. They specify

- How the ‘‘environment’’ of the machine is changed, e.g., how the value of an output signal changes
- The new state that the machine reaches if the transition rule is applied

In the traditional finite state machine model, the environment of the machine consists of two finite and disjoint sets of signals: input signals and output signals. Each signal has an arbitrary, but finite, range of possible values. The condition that must be satisfied for the transition rule to be executable is then phrased as a condition on the value of each input signal, and the effect of the transition can be a change of the values of the output signals. The machine in Table 8.1 illustrates that model. It has one input signal, named In, and one output signal, named Out.

A dash in one of the first two columns is used as a shorthand to indicate a ‘‘don’t care’’ condition (that always evaluates to the boolean value *true*). A transition rule, then, with a dash in the first column applies to all states of the machine, and a transition rule with a dash in the second column applies to all possible values of the input signal. Dashes in the last two columns can be used to indicate that the execution of a transition rule does not change the environment. A dash in the third column means

1. This example first appeared in two seminal papers on finite state machines, published by George H. Mealy [1955] and Edward F. Moore [1956].

that the output signal does not change, and similarly, a dash in the fourth column means that the control state remains unaffected.

In each particular state of the machine there can be zero or more transition rules that are executable. If no transition rule is executable, the machine is said to be in an *end-state*. If precisely one transition rule is executable, the machine makes a deterministic move to a new control state. If more than one transition rule is executable a nondeterministic choice is made to select a transition rule. A *nondeterministic* choice in this context means that the selection criterion is undefined. Without further information either option is to be considered equally likely. From here on, we will call machines that can make such choices *nondeterministic machines.*[2] Table 8.2 illustrates the concept. Two transition rules are defined for control state q1. If the input signal is one, only the first rule is executable. If the input signal is zero, however, both rules will be executable and the machine will move either to state q0 or to state q3.

Table 8.2 — Non-Determinism

Current State	In	Out	Next State
q1	–	0	q0
q1	0	0	q3

The behavior of the machine in Table 8.1 is more easily understood when represented graphically in the form of a state transition diagram, as shown in Figure 8.1.

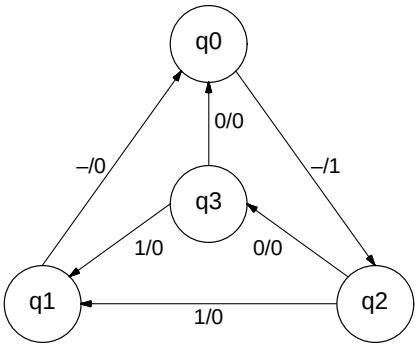


Figure 8.1 — State Transition Diagram

The control states are represented by circles, and the transition rules are specified as directed edges. The edge labels are of the type *c/e*, where *c* specifies the transition condition (e.g., the required set of input values) and *e* the corresponding effect (e.g., a new assignment to the set of output values).

2. The nondeterministic formal automata (NFA) from automata theory are often defined differently. (See for instance, Aho, Sethi and Ullman [1986, p. 114].) Unlike our nondeterministic machines, an NFA can be in more than one state at the same time.

TURING MACHINES

The above definition of a finite state machine is intuitively the simplest. There are many variants of this basic model that differ in the way that the environment of the machines is defined and thus in the definition of the conditions and the effects of the transition rules. For truly finite state systems, of course, the environment must be finite state as well (e.g., it could be defined as another finite state machine). If this requirement is dropped, we obtain the well-known *Turing Machine* model. It is used extensively in theoretical computer science as the model of choice in, for instance, the study of computational complexity. The Turing machine can be seen as a generalization of the finite state machine model, although Turing's work predates that of Mealy and Moore by almost two decades.

The ''environment'' in the Turing machine model is a tape of infinite length. The tape consists of a sequence of squares, where each square can store one of a finite set of tape symbols. All tape squares are initially blank. The machine can read or write one tape square at a time, and it can move the tape left or right, also by one square at a time. Initially the tape is empty and the machine points to an arbitrary square. The condition of a transition rule now consists of the control state of the finite state machine and the tape symbol that can be read from the square that the machine currently points to. The effect of a transition rule is the potential output of a new tape symbol onto the current square, a possible left or right move, and a jump to a new control state.

The tape is general enough to model a random access memory, be it an inefficient one. Table 8.3 illustrates this type of finite state machine.

Table 8.3 — Busy Beaver[3]

Condition		Effect	
Current State	In	Out/Move	Next State
q0	0	1/L	q1
q0	1	1/R	q2
q1	0	1/R	q0
q1	1	1/L	–
q2	0	1/R	q1
q2	1	1/L	q3
q3	–	–	–

This machine has two output signals: one is used to overwrite the current square on the tape with a new symbol, and one is used to move the tape left or right one square. State q3 is an *end* state.

3. This table is Tibor Rado's classic entry into the busy beaver game. The object of the game is to create an N-state (here N = 3) finite state machine that, when started on an empty tape (i.e., with all squares zero) reaches a known end state in a finite number of steps, leaving the longest possible sequence of ones on the tape.

It is fairly hard to define an extension of this variant of the model with a practical method for modeling the controlled interaction of multiple finite state machines. The obvious choice would be to let one machine read a tape that is written by another, but this is not very realistic. Furthermore, the infinite number of potential states for the environment means that many problems become computationally intractable. For the study of protocol design problems, therefore, we must explore other variants of the finite state machine.

COMMUNICATING FINITE STATE MACHINES

Consider what happens if we allow overlap of the sets of input and output signals of a finite state machine of the type shown in Table 8.1. In all fairness, we cannot say what will happen without first considering in more detail what a ''signal'' is.

We assume that signals have a finite range of possible values and can change value only at precisely defined moments. The machine executes a two-step algorithm. In the first step, the input signal values are inspected and an arbitrary executable transition rule is selected. In the second step, the machine changes its control state in accordance with that rule and updates its output signals. These two steps are repeated forever. If no transition rule is executable, the machine will continue cycling through its two-step algorithm without changing state, until a change in the input signal values, effected by another finite state machine, makes a transition possible. A signal, then, has a state, much like a finite state machine. It can be interpreted as a variable that can only be evaluated or assigned to at precisely defined moments.

The behavior of the machine from Table 8.1 is now fully defined, even if we assume a feedback from the output to the input signal. In this case the machine will loop through the following sequence of three states forever: `q0`, `q2`, `q1`. At each step, the machine inspects the output value that was set in the previous transition. The behavior of the machine is independent of the initial value of the input signal.

We can build elaborate systems of interacting machines in this way, connecting the output signals of one machine to the input signals of another. The machines must share a common ''clock'' for their two-step algorithm, but they are not otherwise synchronized. If further synchronization is required, it must be realized with a subtle system of handshaking on the signals connecting the machines. This problem, as we saw in Chapter 5, has three noticeable features: it is a hard problem, it has been solved, and, from the protocol designer's point of view, it is irrelevant. Most systems provide a designer with higher-level synchronization primitives to build a protocol. An example of such synchronization primitives are the send and receive operations defined in PROMELA.

ASYNCHRONOUS COUPLING

In protocol design, finite state machines are most useful if they can directly model the phenomena in a distributed computer system. There are two different and equally valid ways of doing this, based on an asynchronous or a synchronous communication model. With the asynchronous model, the machines are coupled via bounded FIFO

(first-in first-out) message queues. The signals of a machine are now abstract objects called messages. The input signals are retrieved from input queues, and the output signals are appended to output queues. All queues, and the sets of signals, are still finite, so we have not given up the finiteness of our model.

Synchronization is achieved by defining both input and output signals to be conditional on the state of the message queues. If an input queue is empty, no input signal is available from that queue, and the transition rules that require one are unexecutable. If an output queue is full, no output signal can be generated for that queue, and the transition rules that produce one are similarly unexecutable.

From this point on we restrict the models we are considering to those with no more than one synchronizing event per transition rule; that is, a single rule can specify an input or an output, but not both. The reason for this restriction is twofold. First, it simplifies the model. We do not have to consider the semantics of complicated composites of synchronizing events that may be inconsistent (e.g., two outputs to the same output queue that can accommodate only one of the two). Second, it models the real behavior of a process in a distributed system more closely. Note that the execution of a transition rule is an atomic event of the system. In most distributed systems a single send or receive operation is guaranteed to be an atomic event. It is therefore appropriate not to assume yet another level of interlocking in our basic system model.

Table 8.4 — Sender

State	In	Out	Next State
q0	–	mesg0	q1
q1	ack1	–	q0
q1	ack0	–	q2
q2	–	mesg1	q3
q3	ack0	–	q2
q3	ack1	–	q0

As an example of asynchronous coupling of finite state machines, Tables 8.4 and 8.5 give transition table models for a simple version of the alternating bit protocol (see also Chapter 4, Figure 4.8). The possibility of a retransmission after a timeout is not modeled in Table 8.4. We could do so with spontaneous transitions, by adding two rules:

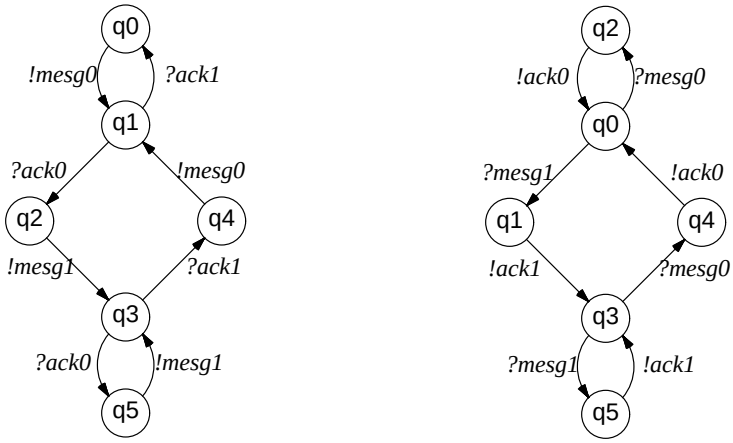
State	In	Out	Next State
q1	–	mesg0	–
q3	–	mesg1	–

The table can model the *possibility* of retransmissions in this way, though not their *probability*. Fortunately, this is exactly the modeling power we need in a system that must analyze protocols independently of any assumptions on the timing or speed of individual processes (see also Chapter 11).

Table 8.5 — Receiver

State	In	Out	Next State
q0	mesg1	–	q1
q0	mesg0	–	q2
q1	–	ack1	q3
q2	–	ack0	q0
q3	mesg0	–	q4
q3	mesg1	–	q5
q4	–	ack0	q0
q5	–	ack1	q3

The last received message can be accepted as correct in states `q1` and `q4`. A state transition diagram for Tables 8.4 and 8.5 is given in Figure 8.2. The timeout option in the sender would produce and extra self-loop on states `q1` and `q3`.



Sender (Table 8.4) *Receiver (Table 8.5)*

Figure 8.2 — State Transition Diagrams, Tables 8.4 and 8.5

We do not have parameter values in messages just yet. In the above model the value of the alternating bit is therefore tagged onto the name of each message.

SYNCHRONOUS COUPLING

The second method for coupling machines is based on a synchronous model of communication, like the one discussed briefly in Chapter 5. The transition conditions are now the ‘‘selections’’ that the machine can make for communication. Again we allow only one synchronizing event per transition rule. The machine can select either an input or an output signal for which a transition rule is specified. To make a move, a signal has to be selected by precisely two machines simultaneously, in one machine as

an output and in the other as an input. If such a match on a signal occurs, both machines make the corresponding transition simultaneously and change their selections in accordance with the new states they reach.

Tables 8.6 and 8.7 give an example of synchronously coupled finite state machines. The machine in Table 8.6 can make just one input selection P in state q0 and one output selection V in state q1.

Table 8.6 — User

State	In	Out	Next State
q0	P	–	q1
q1	–	V	q0

The second machine is almost the same as the first, but has the inputs and outputs swapped (Table 8.7).

Table 8.7 — Server

State	In	Out	Next State
q0	–	P	q1
q1	V	–	q0

If we create two machines of type 8.6 and combine them with one machine of type 8.7, we can be certain that for all possible executions the first two machines cannot both be in state q1 at the same time. Note that synchronous communication was defined to be binary: exactly two machines must participate, one with a given input selection and the other with the matching output selection. Typically, a parameter value will be passed from sender to receiver in the synchronous handshake. The value transfer, however, is not in the model just yet.

We can again consider the synchronous communication as a special case of asynchronous communication with a queue capacity of zero slots (see also Chapters 5 and 11). In the remainder of this chapter we therefore focus on the more general case of a fully asynchronous coupling of finite state machines.

8.3 FORMAL DESCRIPTION

Let us now see if we can tidy up the informal definitions discussed so far. A communicating finite state machine can be defined as an abstract demon that accepts input symbols, generates output symbols, and changes its inner state in accordance with some predefined plan. For now, these symbols or ‘‘messages’’ are defined as abstract objects without contents. We will consider the extensions required to include value transfer in Section 8.8. The finite state machine demons communicate via bounded FIFO queues that map the output of one machine upon the input of another. Let us first formally define the concept of a queue.

A *message queue* is a triple (S, N, C), where:

S is a finite set called the queue vocabulary,

N is an integer that defines the number of slots in the queue, and
C is the queue contents, an ordered set of elements from S.

The elements of S and C are called messages. They are uniquely named, but otherwise undefined abstract objects. If more than one queue is defined we require that the queue vocabularies be disjoint. Let M be the set of all messages queues, a superscript $1 \le m \le |M|$ is used to identify a single queue, and an index $1 \le n \le N$ is used to identify a slot within the queue. C_n^m, then, is the n-th message in the m-th queue. A system vocabulary V can be defined as the conjunction of all queue vocabularies, plus a null element that we indicate with the symbol ε. Given the set of queues M, numbered from 1 to $|M|$, the system vocabulary V is defined as

$$V = \bigcup_{m=1}^{|M|} S^m \bigcup \varepsilon$$

Now, let us define a communicating finite state machine.

A *communicating finite state machine* is a tuple (Q, q_0, M, T), where
Q is a finite, non-empty set of states,
q_0 is an element of Q, the *initial state*,
M is a set of message queues, as defined above, and
T is a state transition relation.

Relation T takes two arguments, $T(q,a)$, where q is the current state and a is an action. So far, we allow just three types of actions: inputs, outputs, and a null action ε. The executability of the first two types of actions is conditional on the state of the message queues. If executed, they both change the state of precisely one message queue. Beyond this, it is immaterial, at least for our current purposes, what the precise definition of an input or an output action is.

The transition relation T defines a set of zero or more possible successor states in set Q for current state q. This set will contain precisely one state, unless nondeterminism is modeled, as in Table 8.2. When $T(q,a)$ is not explicitly defined, we assume $T(q,a) = \varnothing$.

$T(q,\varepsilon)$ specifies spontaneous transitions. A sufficient condition for these transitions to be executable is that the machine be in state q.

8.4 EXECUTION OF MACHINES

Consider a system of P finite state machines, with overlapping sets of message queues. The union of the sets of all message queues is again called M. This system of communicating finite state machines is executed by applying the following rules, assuming asynchronous coupling only. The elements of finite state machine i are referred to with a superscript i.

ALGORITHM 8.1 — FSM EXECUTION

1. Set all machines in their initial state, and initialize all message queues to empty:

$$\forall(i), 1 \le i \le P \rightarrow q^i = q_0^i$$

$$\forall(i), 1 \le i \le |M| \;\rightarrow\; C^i = \varnothing$$

2. Select an arbitrary machine i and an arbitrary transition rule T^i with

$$T^i(q, {}^i a) \neq \varnothing \quad \text{and} \quad a \textit{ is executable}$$

and execute it.

3. If no executable transition rules remain, the algorithm terminates.

Action a can be an input, an output, or it can be the null action ε. Let $1 \le d(a) \le |M|$ be destination queue of an action a, and let $m(a)$ be the message that is sent or received, $m(a) \in S^{d(a)}$. Further, let N^i represent the number of slots in message queue i. In an asynchronous system, for instance, the following three rules can be used to determine if a is executable.

$$a = \varepsilon \tag{1}$$

or

$$a \text{ is an input and } m(a) = C_1^{d(a)} \tag{2}$$

or

$$a \text{ is an output and } |C^{d(a)}| < N^{d(a)} \tag{3}$$

Algorithm 8.1 does not necessarily terminate.

8.5 MINIMIZATION OF MACHINES

Consider the finite state machine shown in Table 8.8, with the corresponding state transition diagram in Figure 8.3.

Table 8.8 — Receiver-II

Condition		Effect	
Current State	In	Out	Next State
q0	mesg1	–	q1
q0	mesg0	–	q2
q1	–	ack1	q0
q2	–	ack0	q0

Though this machine has three states fewer than the machine from Table 8.5, it certainly looks like it behaves no differently. Two machines are said to be *equivalent* if they can generate the same sequence of output symbols when offered the same sequence of input symbols. The keyword here is *can*. The machines we study can make nondeterministic choices between transition rules if more than one is executable at the same time. This nondeterminism means that even two equal machines *can* behave differently when offered the same input symbols. The rule for equivalence is that the machines must have equivalent choices to be in equivalent states.

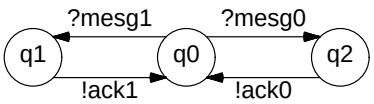


Figure 8.3 — State Transition Diagram for Table 8.8

States within a single machine are said to be equivalent if the machine can be started in any one of these states and generate the same set of possible sequences of outputs when offered any given test sequence of inputs. The definition of an appropriate equivalence relation for states, however, has to be chosen with some care. Consider the following PROMELA process.

```
proctype A()
{       if
        :: q?a -> q?b
        :: q?a -> q?c
        fi
}
```

Under the standard notion of language equivalence that is often defined for deterministic finite state machines, this would be equivalent to

```
proctype B()
{       q?a;
        if
        :: q?b
        :: q?c
        fi
}
```

since the set of all input sequences (the language) accepted by both machines is the same. It contains two sequences, of two messages each:

```
{ q?a;q?b , q?a;q?c }
```

The behavior of the two processes, however, is very different. The input sequence `q?a;q?b`, for instance, is always accepted by process `B` but may lead to an unspecified reception in process `A`. For nondeterministic communicating finite state machines, therefore processes `A` and `B` are not equivalent. The definitions given below will support that notion.

In the following discussion of equivalence, state minimization, and machine composition, we will focus exclusively on the set of control states Q and the set of transitions T of the finite state machines. Specifically, the internal ‘‘state’’ of the message queues in set M is considered to be part of the environment of a machine and not contributing to the state of the machine itself. That this is a safe assumption needs some motivation. Consider, as an extreme case, a communicating finite state machine that accesses a private message queue to store internal state information. It can do so by appending messages with state information in the queue and by retrieving that information later. The message queue is internal and artificially increases the number of states of the machine.

When we consider the message queue to be part of the environment of a machine in the definitions that follow, we ignore the fact that the information that is retrieved from such a private queue is always fixed (i.e., it can only have been placed in the queue by the same machine in a previous state). If we say that two states of this machine are equivalent if they respond to the same input messages in the same way, we do in fact place a *stronger* requirement on the states than strictly necessary. We require, for instance, that the two states would respond similarly to messages that could never be in a private queue for the given state. To suppress state information that could be implicit in the messages queue contents therefore does not relax the equivalence requirements. As we will see, it does lead to simpler algorithms.

Using this approach, the set of control states of a communicating finite state machine can be minimized, without changing the external behavior of the machine, by replacing every set of equivalent states with a single state. More formally, we can say that this equivalence relation defines a partitioning of the states into a finite set of disjoint equivalence classes. The smallest machine equivalent to the given one will have as many states as the original machine has equivalence classes.

We can now define a procedure for the minimization of an arbitrary finite state machine with $|Q|$ states.

ALGORITHM 8.2 — FSM MINIMIZATION

1. Define an array E of $|Q| \times |Q|$ boolean values. Initially, every element $E[i,j]$ of the array is set to the truth value of the following condition, for all actions a:

$$T(i,a) \neq \varnothing \;\Leftrightarrow\; T(j,a) \neq \varnothing$$

Two states are not equivalent unless the corresponding state transition relations are defined for the same actions.

2. If the machine considered contains only deterministic choices, T defines a unique successor state for all *true* entries of array E. Change the value of all those entries $E[i,j]$ to the value of

$$\forall(a),\;\; E[T(i,a),T(j,a)]$$

It means that states are not equivalent unless their successors are also equivalent. When $T(i,a)$ and $T(j,a)$ can have more than one element, the relation is more complicated. The value of $E[i,j]$ is now set to *false* if either of the following two conditions is *false* for any action a.

$$\forall(p),\; p \in T(i,a) \;\rightarrow\; \exists(q), q \in T(j,a) \text{ and } E[p,q]$$
$$\forall(q),\; q \in T(j,a) \;\rightarrow\; \exists(p), p \in T(i,a) \text{ and } E[q,p]$$

This means that states i and j are not equivalent unless for every possible successor state p of state i there is at least one equivalent successor state q of state j, and vice versa.

3. Repeat step 2 until the number of *false* entries in array E can no longer be increased.

The procedure always terminates since the number of entries of the array is finite and each entry can only be changed once, from *true* to *false*, in step 2. When the

procedure terminates, the entries of the array define a partitioning of the $|Q|$ states into equivalence classes. State i, $1 \le i \le |Q|$, is equivalent with all states j, $1 \le j \le |Q|$, with $E[i,j] = true$.

Table 8.9 — Equivalence

	q0	q1	q2	q3	q4	q5
q0	1	0	0	1	0	0
q1	0	1	0	0	0	1
q2	0	0	1	0	1	0
q3	1	0	0	1	0	0
q4	0	0	1	0	1	0
q5	0	1	0	0	0	1

If we apply this procedure to the finite state machine in Table 8.5, we obtain the stable array of values for E shown in Table 8.9 after a single application of the first two steps. A one in the table represents the boolean value *true*. From the table we see that state pairs (q0, q3), (q1, q5), and (q2, q4) are equivalent. We can therefore reduce Table 8.5 to the three-state finite state machine that was shown in Table 8.8. It is necessarily the smallest machine that can realize the behavior of Table 8.5.

The procedure above can be optimized by noting, for instance, that array E is symmetric: for all values of i and j we must have $E(i,j) = E(j,i)$. Trivially, every state is equivalent with itself.

8.6 THE CONFORMANCE TESTING PROBLEM

The procedure for testing equivalence of states can also be applied to determine the equivalence of two machines. The problem is then to determine that every state in one machine has an equivalent in the other machine, and vice versa. Of course, the machines need not be equal to be equivalent.

A variant of this problem is of great practical importance. Suppose we have a formal protocol specification, in finite state machine form, and an implementation of that specification. The two machines must be equivalent, that is the implementation, seen as a black box, should respond to input signals exactly as the reference machine would. We cannot, however, know anything with certainty about the implementation's true internal structure. We can try to establish equivalence by systematically probing the implementation with trial input sequences and by comparing the responses with those of the reference machine. The problem is now to find just the right set of test sequences to establish the equivalence or non-equivalence of the two machines. This problem is known in finite state machine theory as the *fault detection* or conformance testing problem. Chapter 10 reviews the methods that have been developed for solving this problem.

Carrying this one step further, we may also want to determine the internal structure of an unknown finite state machine, just by probing it with a known set of input signals

and by observing its responses. This problem is known as the *state verification* problem. Without any further knowledge about the machine, that problem is alas unsolvable. Note, for instance, that in Figure 8.1 state q3 cannot be distinguished from state q2 by any test sequence that starts with an input symbol one. Similarly, state q1 cannot be distinguished from state q3 by any sequence starting with a zero. Since every test sequence has to start with either a one or a zero there can be no single test sequence that can tell us reliably in which state this machine is.

8.7 COMBINING MACHINES

By collapsing two separate finite state machines into a single machine the complexity of formal validations based on finite state machine models may be reduced. The algorithm below is referred to in Chapter 11 in the discussion of an incremental protocol validation method, and in Chapter 14 in the discussion of methods for stepwise abstraction.

The problem is to find a tuple (Q, q_0, M, T) for the combined machine, given two machines (Q^1, q_0^1, M^1, T^1) and (Q^2, q_0^2, M^2, T^2).

ALGORITHM 8.3 — FSM COMPOSITION

1. Define the product set of the two sets of states of the two state machines. If the first machine has $|Q^1|$ states and the second machine has $|Q^2|$ states the product set contains $|Q^1| \times |Q^2|$ states. We initially name the states of the new machine by concatenating the state names of the original machines in a fixed order. This defines set Q of the combined machine. The initial state q_0 of the new machine is the combination $q_0^1 q_0^2$ of the initial states of the two original machines.

2. The set of message queues M of the combined machine is the union of the sets of queues of the separate machines, $M^1 \cup M^2$. The two original sets need not be disjoint. The vocabulary V of the new machine is the combined vocabulary of M^1 and M^2, and the set of actions a is the union of all actions that the individual machines can perform.

3. For each state $q^1 q^2$ in Q, define transition relation T for each action a as the nondeterministic choice of the corresponding relations of M^1 and M^2 separately, when placed in the individual states q^1 and q^2. This can be written:

$$\forall(q^1 q^2)\forall(a), \ \rightarrow \ T(q^1 q^2, a) = T^1(q^1, a) \cup T^2(q^2, a)$$

The combined machine can now be minimized using Algorithm 8.2. Algorithm 8.3 can be readily adapted to combine more than two machines.

The greatest value of the composition technique from the last section is that it allows us to simplify complex behaviors. In protocol validations, for instance, we could certainly take advantage of a method that allows us to collapse two machines into one. One method would be to compose two machines using Algorithm 8.3, remove all their internal interactions, i.e., the original interface between the two machines, and minimize the resulting machine.

There are two pieces missing from our finite state machine framework to allow us to

apply compositions and reductions in this way. First, the finite state machine model we have developed so far cannot easily represent PROMELA models. In the next section we show how the basic finite state machine model can be extended sufficiently to model PROMELA models elegantly. The second piece that is missing from our framework is a method for removing internal actions from a machine without disturbing its external behavior. We discuss such methods in Section 8.9.

8.8 EXTENDED FINITE STATE MACHINES

The finite state machine models we have considered so far still fall short in two important aspects: the ability to model the manipulation of variables conveniently and the ability to model the transfer of arbitrary values. These machines where defined to work with abstract objects that can be appended to and retrieved from queues and they are only synchronized on the access to these queues.

We make three changes to this basic finite state machine model. First, we introduce an extra primitive that is defined much like a queue: the variable. Variables have symbolic names and they hold abstract objects. The abstract objects, in this case, are integer values. The main difference from a real queue is that a variable can hold only one value at a time, selected from a finite range of possible values. Any number of values can be appended to a variable, but only the last value that was appended can be retrieved.

The second change is that we will now use the queues specifically to transfer integer values, rather than undefined abstract objects. Third, and last, we introduce a range of arithmetic and logical operators to manipulate the contents of variables.

Table 8.10 — Finite State Variable

Current State	In	Out	Next State
q0	s0	–	–
q0	s1	–	q1
q0	s2	–	q2
q0	rv	–	r0
r0	–	0	q0
q1	s0	–	q0
q1	s1	–	–
q1	s2	–	q2
q1	rv	–	r1
r1	–	1	q1
q2	s0	–	q0
q2	s1	–	q1
q2	s2	–	–
q2	rv	–	r2
r2	–	2	q2

The extension with variables, provided that they have a finite range of possible values, does not increase the computational power of finite state machines with bounded FIFO queues. A variable with a finite range can be simulated trivially by a finite state machine. Consider the six-state machine shown in Table 8.10, that models a variable with the range of values from zero to two. The machine accepts four different input messages. Three are used to set the pseudo variable to one of its three possible values. The fourth message, rv, is used to test the current value of the pseudo variable. The machine will respond to the message rv by returning one of the three possible values as an output message.

Thus, at the expense of a large number of states, we can model any finite variable without extending the basic model, as a special purpose finite state machine. The extension with explicit variables, therefore, is no more than a modeling convenience.

Recall that the transition rules of a finite state machine have two parts: a condition and an effect. The conditions of the transition rules are now generalized to include boolean expressions on the value of variables, and the effects (i.e. the actions) are generalized to include assignment to variables.

An *extended finite state machine* can now be defined as a tuple (Q, q_0, M, A, T), where A is the set of variable names. Q, q_0, and M are as defined before. The state transition relation T is unchanged. We have simply defined two extra types of actions: boolean conditions on and assignments to elements of set A. A single assignment can change the value of only one variable. Expressions are built from variables and constant values, with the usual arithmetic and relational operators.

In the spirit of the validation language PROMELA, we can define a condition to be executable only if it evaluates to *true*, and let an assignment always be executable. Note carefully that the extended model of communicating finite state machines is a *finite* state model, and almost all results that apply to finite state machines also apply to this model.

EXTENDED I/O

Input and output actions can now be generalized as well. We will define I/O actions as finite, ordered sets of values. The values can be expressions on variables from A, or simply constants. By definition the first value from such an ordered set defines the destination queue for the I/O, within the range $1..|M|$. The remaining values define a data structure that is appended to, or retrieved from, the queue when the I/O action is performed. The semantics of executability can again be defined as in PROMELA.

EXAMPLE

Consider the following PROMELA fragment, based on an example from Chapter 5.

```
proctype Euclid
{       pvar x, y;

        In?x,y;
```

```
        do
        :: (x >  y) -> x = x - y
        :: (x <  y) -> y = y - x
        :: (x == y) -> break
        od;
        Out!x
}
```

The process begins by receiving two values into variables x and y, and it completes by returning the greatest common divisor of these two values to its output queue. The matching extended finite state machine is shown in Table 8.11, where we combine all conditions, assignments and I/O operations in a single column.

Table 8.11 — Extended Finite State Machine

Current State	Action	Next State
q0	In?x,y	q1
q1	x>y	q2
q1	x<y	q3
q1	x=y	q4
q2	x=x-y	q1
q3	y=y-x	q1
q4	Out!x	q5
q5	–	–

Set *A* has two elements, *x* and *y*.

We now have a simple mapping from PROMELA models to extended finite state machines. Algorithm 8.3, for instance, can now be used to express the combined behavior of two PROMELA processes by a single process. We noted before that this technique could be especially useful in combination with a *hiding* method that removes internal actions from a machine without disturbing its external behavior. We take a closer look at such methods in the next section.

8.9 GENERALIZATION OF MACHINES

Consider the following PROMELA model.

```
 1 proctype generalize_me(chan ans; byte p)
 2 {   chan internal[1] of { byte };
 3     int r, q;
 4
 5     internal!cookie;
 6     r  = p/2;
 7     do
 8     :: (r <= 0) -> break
 9     :: (r >  0) ->
10             q = (r*r + p)/(2*r);
11             if
12             :: (q != r) -> skip
```

```
13              :: (q == r) -> break
14              fi;
15              r = q
16      od;
17      internal?cookie;
18      if
19      :: (q <  p/3) -> ans!small
20      :: (q >= p/3) -> ans!great
21      fi
22  }
```

A process of this type will start by sending a message `cookie` to a local message channel. It will then perform some horrible computation, using only local variables, read back the message from the channel `internal`, and send one of two possible messages over an external message channel `ans`.

Now, for starters, nothing detectable will change in the external behavior of this process if we remove lines 2, 5 and 17. The message channel is strictly local, and there is no possible behavior for which any of the actions performed on the channel can be unexecutable. Lines 5 and 17 are therefore equivalent to `skip` operations and can be deleted from the model. Reductions, or *prunings*, of this type produce machines that have a equivalent external behavior to the non-reduced machines. This is not true for the next type of reduction we discuss: generalization.

The horrible computation performed by the process, between lines 6 and 16, does not involve any global variables or message interactions. Once the initial value of variable `p` is chosen, the resulting message sent to channel `ans` is fixed. If we are interested in just the external behavior of processes of type `generalize_me`, independently of the precise value of `p`, the model could be rewritten as

```
proctype generalized(chan ans; byte p)
{
        if
        :: ans!small
        :: ans!great
        fi
}
```

This specification merely says that within a finite time after a process of this type is instantiated, it sends either a message of type `small` or a message of type `great` and terminates. To justify the reduction we must of course show that the loop in the original specification will always terminate. If this is not the case, or cannot be proven, the correct reduction would be

```
proctype generalized(chan ans; byte p)
{
        if
        :: (0)
        :: ans!small
        :: ans!great
        fi
}
```

where the possibility of blocking is preserved explicitly.

We call a reduction of this type, where uninteresting but strictly local, behavior is removed, a *generalization*. A process of type `generalized` can do everything that a process of type `generalize_me` can do, but it can do more. The generalized process can, for instance, for any given parameter `p`, return either of the two messages, while the non-generalized processes will pick only one. The generalized processes is *only* more general in the way it can produce output, not in the way it can accept input, or in general in the way other processes can constrain its behavior via global objects.

The usefulness of generalizations in protocol validation can be explained as follows. Consider two protocol modules `A` and `B` whose combined behavior is too complex to be analyzed directly. We want to validate a correctness requirement for the processes in module `A`. We can do this by simplifying the behavior in module `B`, for instance by combining, pruning, generalizing, and minimizing machines. If the behavior in module `B` is generalized as discussed above, the new module `B` will still be capable of behaving precisely like the unmodified module `B`, but it can do more. If we can prove the observance of a correctness requirement for module `A` in the presence of the generalized module `B`, which may be easier, the result will necessarily also hold for the original, more complex, module `B`, because the original behavior is a subset of the new behavior.

Two things should be noted. First, if we are interested in proving a property of module `A` we simplify its environment, which in this case is module `B`. We do not change module `A` itself. Second, it is important that the modified behavior of module `B` does not, by virtue of the modifications, allow module `A` to pass its test. This is guaranteed by the fact that the generalized module `B` continues to adhere to all constraints that can be imposed by `A`, via global objects, such as message channels and variables. The validation, then, gives us the best of both worlds. It performs a stronger test, since it validates properties for more general conditions than defined in the original protocol, yet it is easier to perform, since a generalized process can be smaller than its original.

A general method for the reduction of an arbitrary PROMELA `proctype` definition can be described as follows.

- Identify selection and repetition structures in which all the guards are conditions on local variables only, and in which the union of all guards is `true`.
- Replace each of the guards identified in the first step with the PROMELA statement `skip`.

- Replace all assignments to local variables that are no longer part of any condition, with `skip`.
- Remove all redundant declarations and minimize or simplify the new `proctype` body, for instance, by combining equal clauses in selection and repetition structures, and by removing redundant `skip` statements.

If we apply this method to the process type `generalize_me`, after pruning away the interactions on channel `internal`, we can reduce it to

```
proctype generalized_2(chan ans; byte p)
{
        do
        :: break
        :: skip
        od;
        if
        :: ans!small
        :: ans!great
        fi
}
```

which is similar to and has the same external behavior as the (second) version we derived earlier with a little more handwaving. Note that the loop in the above version does not necessarily terminate.

A more substantial application of this generalization technique and the resulting reduction in complexity is discussed in Chapter 14.

8.10 RESTRICTED MODELS

To conclude this chapter, we look at two other interesting variants of the basic finite state machine model that have been applied to the study of protocol problems. Many variations of the basic finite state machine model have been used for the analysis of protocol systems, both restrictions and extensions. The restricted versions have the advantage, at least in principle, of a gain in analytical power. The extended versions have the advantage of a gain in modeling power. The most popular variants of the finite state machine are formalized token nets, often derived from the Petri Net model. Below we briefly review the Petri Net model and discuss one of the variations, the FIFO Net.

PETRI NETS

A Petri Net is a collection of *places*, *transitions*, and directed *edges*. Every edge connects a place to a transition or vice versa. Places are graphically represented by circles, transitions by bars, and edges by directed arcs. Informally, a place corresponds to a condition and a transition corresponds to an event. The *input places* of transition *T* are the places that are directly connected to *T* by one or more edges. The input places correspond to conditions that must be fulfilled before the event corresponding to *T* can occur. The output places of a transition similarly correspond to the effect of the event on the conditions represented by the places.

Each place that corresponds to a fulfilled condition is marked with one or more *tokens* (sometimes called a *stone*). The occurrence of an event is represented in the Petri Net as the *firing* of a transition. A transition is enabled if there is at least one token in each of its input places. The effect of a firing is that one token is added to the markings of all output places of the firing transition, and one token is removed from the markings of all its input places.

Two transitions are said to *conflict* if they share at least one input place. If the shared place contains precisely one token, both transitions may be enabled to fire, but the firing of one transition disables the other. By definition the firing of any combination of two transitions is always mutually exclusive: only one transition can fire at a time.

By assigning zero or more tokens to each place in the net we obtain an initial marking. Each firing creates a new marking. A series of firings is called an *execution sequence*. If for a given initial marking all possible execution sequences can be made infinitely long, the initial marking, and trivially all subsequent markings, are said to be *live*. If in a certain marking no transition is enabled to fire, the net is said to *hang*. An initial making is said to be *safe* if no subsequent execution sequence can produce a marking where any place has more than one token.

A number of properties has been proven about Petri Nets, but mostly about still further simplified versions. Two examples of such variants are:

- Petri Nets in which precisely one edge is directed to and from each place. Such nets are called *marked graphs*. In a marked graph there can be no conflicting transitions.
- Petri Nets in which all transitions have at most one input place and one output place. These nets are called *transition diagrams*.

Figure 8.4 gives an example of a Petri Net modeling a deadlock problem. Initially, the two top transitions *t1* and *t2* are enabled. After *t1* fires, transition *t3* becomes enabled. If it fires, all is well. If in this marking, however, transition *t2* fires, the net will *hang*.

A token in a Petri Net symbolizes more than the fulfillment of a *condition*, as described above. It also symbolizes a *control flow point* in the program, and it symbolizes a *privilege* to proceed beyond a certain point. A token models a shared resource that can be claimed by more than one transition. All these abstractions symbolize the enforcement of partial orderings on the set of possible execution sequences in the system modeled. Especially relevant to the protocol modeling problem is that mixing these abstractions can make it more difficult than necessary to distinguish computation from communication in a Petri Net model.

The complexity of a Petri Net representation rises rapidly with the size of the problem being modeled. It is virtually impossible to draw a clear net for protocol systems that include more than two or three processes. This makes the Petri Net models relatively weak in modeling power compared to communicating finite state machines, without offering an increase in analytical power. There are, for instance, no standard procedures, other than reachability analysis, to analyze a Petri Net for the presence of

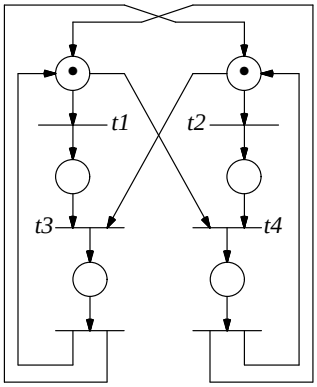


Figure 8.4 — Petri Net with hang state

hang states. Neither are there standard procedures for simplifying a large Petri Net into one or more smaller, equivalent ones.

One final note on the modeling power of basic Petri Nets. We observed above that the places in a Petri Net can be used to model conditions. It is fairly easy to model logical *and* and *or* tests on places using multiple edges, but there is no general way to model a logical *not*-operation (negation). With a logical *not*-operation it would be possible to define that a transition can fire if a place holds no tokens.

Of course, there are many good applications of Petri Net theory. They have been applied successfully to the study of a range of theoretical problems in parallel computation. For the above pragmatic reasons, however, we conclude that Petri Nets do not give us an advantage in the study of protocol design and validation problems.

FIFO NETS

FIFO Nets are an interesting generalization of Petri Nets and a relatively recent addition to the range of tools proposed for studying distributed systems (see Bibliographic Notes).

A FIFO Net, like a Petri Net, has places, edges, and transitions, but the places contain symbols rather than tokens. The symbols are appended to and reclaimed from the places by transition firings. They are stored by the places in FIFO queues. Both incoming and outgoing edges of transitions are labeled with symbol names. A transition can only fire if the queue of each of its input places can deliver the symbol that corresponds to the edge connecting the transition to that place. Upon firing the labels on the outgoing edges specify which symbols are to be appended to the queues of the corresponding output places.

The generalization of FIFO Nets is strong enough to make them equivalent in computational power to the finite state machines that we defined earlier. Alas, there are no

better procedures to analyze FIFO Nets for interesting protocol errors, such as deadlock. In some cases, procedures do exist for restricted versions of FIFO Nets, but again the restrictions generally reduce the modeling power too severely to make them of interest as a general tool for designing or analyzing protocol systems.

8.11 SUMMARY

The formal model of a finite state machine was developed in the early 1950s for the study of problems in computational complexity and, independently, for the study of problems in the design of combinatorial and sequential circuits. There are almost as many variants of the basic model of a finite state machine as there are applications. For the study of protocol design problems we need a formalism in which we can model the primitives of process interactions as succinctly as possible. With this in mind we developed an extended finite state machine model that can directly model message passing and the manipulation of variables. Its semantics are closely linked to the semantics of PROMELA.

There are three main criteria for evaluating the adequacy of formal modeling tools:

- Modeling power
- Analytical power
- Descriptive clarity

The main purpose of the modeling is to obtain a gain in analytical power. It should be easier to analyze the model than it is to analyze the original system being modeled. We have chosen the finite state machine as our basic model. There is a small set of useful properties that can easily be established with a static analysis of finite state machine models. More importantly, however, the manipulation of finite state machines can be automated, and more sophisticated dynamic analysis tools can be developed. We study such tools in Part IV of this book. The descriptive clarity of the finite state machines is debatable. It can well be argued that they trade descriptive clarity for analytical power. By using PROMELA as an intermediate form of an extended finite state machine, however, we can circumvent this problem.

The Turing machine model falls short on all three criteria listed above when applied to the study of protocol problems. In particular, the definition of the ‘‘environment’’ is hard to exploit in the modeling of communications. Perhaps even more importantly, many problems of interest, such as absence of deadlock, are intractable for Turing machine models. The model is too powerful for our purpose.

Petri Nets have been used for the study of distributed systems since their inception in the early 1960s. The Petri Net and the FIFO Net have an appealing conceptual simplicity that is mostly based on the graphical representation of the mechanism of process interaction. This advantage in descriptive clarity, however, is lost when the size of the problem exceeds a modest limit. Beyond roughly fifty states per process, the nets become inscrutable. Another, more subtle problem is to distinguish synchronization aspects from the control flow aspects in a Petri Net model. Both are modeled with the same tool: the token. It can be argued that descriptive clarity is traded here

for conceptual simplicity. For the modeling of protocol systems this turns out to be an unfortunate trade-off. Protocols of a realistic size typically have many times the numbers of states beyond which a Petri Net becomes unusable. The restrictions of the model imply a loss of modeling power that is not offset by a comparable gain in analytical power.

EXERCISES

8-1. Explain the difference between the dash introduced as a notational convenience in Section 8.2 and the ε introduced in Section 8.3. □

8-2. Apply Algorithm 8.1 to Table 8.1. □

8-3. Define the rules for the executability of *a* in Algorithm 8.1, assuming a synchronous instead of asynchronous coupling of machines. □

8-4. Change Algorithm 8.3 to combine any number of machines. □

8-5. Implement Algorithms 8.1 to 8.3 in your favorite programming language. Invent a syntax for specifying a system of finite state machines. Specify Tables 8.4 and 8.5 in this formalism and use your programs to minimize the corresponding machines, to combine them into one single machine, and to simulate the execution of the resulting description. □

8-6. Model the behavior of Tables 8.6 and 8.7 in PROMELA. □

8-7. Do the *run* and *chan* operators in PROMELA make the systems modeled unbounded? □

8-8. Find an algorithm that detects which message queues from the definition of a communicating finite state machine are only used internally, to store state information, and that removes them from the specification by increasing the number of states. □

8-9. (S. Purushothaman) Are two machine states equivalent if one of the two states contains an unexecutable transition that the other state lacks (cf. a receive from an always-empty message queue) ? □

8-10. Derive a formal finite state machine description for the example processes `A` and `B` on page 172 and show that they are not equivalent. □

BIBLIOGRAPHIC NOTES

The theory of finite state machines has a long history and at least parts of it can be found in many computer science text books, e.g., Aho, Hopcroft and Ullman [1974], Aho, Sethi and Ullman [1986]. The original idea of the finite state machine is attributed to McCulloch and Pitts [1943]. Most tightly connected to the theory that was subsequently developed are the names of D.A. Huffman, G.H. Mealy, E.F. Moore and A.M. Turing. The original paper on Turing machines is Turing [1936]. For a more recent discussion see, for instance, Kain [1972]. Huffman's early work on the concept of finite state machines and state equivalence was published in Huffman [1954] and reprinted in Moore [1964]. Edward Moore's first paper on finite state machines is Moore [1956]. In Moore's model the output of a finite state machine depends only on its current state, not on the transition that produced it. The early papers by Moore are collected in Moore [1964]. George Mealy's original paper, on the finite state machine model can be found in Mealy [1955]. Mealy's model is slightly more general than

Moore's. In his model the output of a finite state machine depends on the last transition that was executed, not necessarily on the current state.

For a more general introduction to the basic theory and its application to circuit design, refer to, for instance, Harrison [1965], Hartmanis and Stearns [1966], Kain [1972], Shannon and McCarthy [1956]. The ''busy beaver problem'' was introduced in Rado [1962] and further studied in Lin and Rado [1965].

The formal model of a finite state machine has been applied to the study of communication protocols since the very first publications, e.g., Bartlett, Scantlebury and Wilkinson [1969]. It has long been the method of choice in almost all formal modeling and validation techniques, cf. Bochmann and Sunshine [1980]. The model was first applied to a protocol validation problem in Zafiropulo [1978]. A very readable introduction the theory of communicating finite state machines can be found in Brand and Zafiropulo [1983].

An excellent overview of various methods for deriving equivalence relations for concurrent processes, and the complexity of the corresponding algorithms, can be found in Kanellakis and Smolka [1990]. The generalization of machines is closely related to the concept of a ''protocol projection'' that was introduced in Lam and Shankar [1984].

Petri's model was first described in Petri [1962]. See also Agerwala [1975] for a discussion of the Petri Net's modeling power and for some extensions. A discussion of FIFO Nets can be found in Finkel and Rosier [1987]. There are, of course, many other interesting analytical models for concurrent systems. An overview and assessment can be found in, e.g., Holzmann [1979].