Structure and Interpretation of Signals and Systems
Edward Lee and Pravin Varaiya
ISBN: 0-201-74551-8
©2003 Pearson Education

Chapter 3

State Machines

Systems are functions that transform signals. The domain and the range of these functions are both signal spaces, which significantly complicates specification of the functions. A broad class of systems can be characterized using the concept of **state** and the idea that a system evolves through a sequence of changes in state, or **state transitions**. Such characterizations are called **state-space models**.

A state-space model describes a system procedurally, giving a sequence of step-by-step operations for the evolution of a system. It shows how the input signal drives changes in state, and how the output signal is produced. It is thus an imperative description. Implementing a system described by a state-space model in software or hardware is straightforward. The hardware or software simply needs to sequentially carry out the steps given by the model. Conversely, given a piece of software or hardware, it is often useful to describe it using a state-space model, which yields better to analysis than more informal descriptions.

In this chapter, we introduce state-space models by discussing systems with a finite (and relatively small) number of states. Such systems typically operate on event streams, often implementing control logic. For example, the decision logic of modem negotiation described in chapter 1 can be modeled using a finite state model. Such a model is much more precise than the English-language descriptions that are commonly used for such systems.

3.1 Structure of state machines

A description of a system as a function involves three entities: the set of input signals, the set of output signals, and the function itself, $F\colon InputSignals \to OutputSignals$. For a **state machine**, the input and output signals have the form

$$EventStream\colon Naturals_0 \to Symbols,$$

where $Naturals_0 = \{0, 1, 2, \cdots\}$, and *Symbols* is an arbitrary set. The domain of these signals represents *ordering* but not necessarily time (neither discrete nor continuous time). The ordering of the domain means that we can say that one event occurs *before* or *after* another event. But we cannot say how much time elapsed between these events. In chapter 5 we will study how state-space models can be used with functions of time.

A state machine constructs the output signal one symbol at a time by observing the input signal one symbol at a time. Specifically, a state machine *StateMachine* is a 5-tuple,

$$\mathit{StateMachine} = (\mathit{States}, \mathit{Inputs}, \mathit{Outputs}, \mathit{update}, \mathit{initialState}) \tag{3.1}$$

where $\mathit{States}, \mathit{Inputs}, \mathit{Outputs}$ are sets, *update* is a function, and $\mathit{initialState} \in \mathit{States}$. The meaning of these names is:

States is the **state space**,
Inputs is the **input alphabet**,
Outputs is the **output alphabet**,
$\mathit{initialState} \in \mathit{States}$ is the **initial state**, and
$\mathit{update}\colon \mathit{States} \times \mathit{Inputs} \to \mathit{States} \times \mathit{Outputs}$ is the **update function**.

This five-tuple is called the **sets and functions model** of a state machine.

Inputs and *Outputs* are the sets of possible input and output symbols. The set of **input signals** consists of all infinite sequences of input symbols,

$$\mathit{InputSignals} = [\mathit{Naturals}_0 \to \mathit{Inputs}].$$

The set of **output signals** consists of all infinite sequences of output symbols,

$$\mathit{OutputSignals} = [\mathit{Naturals}_0 \to \mathit{Outputs}].$$

Let $x \in \mathit{InputSignals}$ be an input signal. A particular symbol in the signal can be written $x(n)$ for any $n \in \mathit{Naturals}_0$. We write the entire input signal as a sequence

$$(x(0), x(1), \cdots, x(n), \cdots).$$

This sequence defines the function x in terms of symbols $x(n) \in \mathit{Inputs}$, which represent particular input symbols.

We reiterate that the index n in $x(n)$ does not refer to time, but rather to the **step number**. This is an **ordering constraint** only: step n occurs after step $n-1$ and before step $n+1$. The state machine evolves (i.e. moves from one state to the next) in **steps**.[1]

[1] Of course the steps could last a fixed duration of time, in which case there would be a simple relationship between step number and time. The relationship may be a mixed one, where some input symbols are separated by a fixed amount of time and some are not.

3.1.1 Updates

The interpretation of *update* is this. If $s(n) \in States$ is the current state at step n, and $x(n) \in Inputs$ is the current input symbol, then the current output symbol and the next state are given by

$$(s(n+1), y(n)) = update(s(n), x(n)).$$

Thus the *update* function makes it possible for the state machine to construct the output signal step by step by observing the input signal step by step.

The state machine *StateMachine* of (3.1) defines a function

$$F : InputSignals \to OutputSignals \tag{3.2}$$

such that for any input signal $x \in InputSignals$ the corresponding output signal is $y = F(x)$. However, it does much more than just define this function. It also gives us a procedure for evaluating this function on a particular input signal. The **state response** $(s(0), s(1), \cdots)$ and output signal y are constructed as follows:

$$\begin{aligned} s(0) &= initialState, && (3.3)\\ \forall n \geq 0,\ (s(n+1), y(n)) &= update(s(n), x(n)), && (3.4) \end{aligned}$$

Observe that if the initial state is changed, the function F will change, so the initial state is an essential part of the definition of a state machine.

Each evaluation of (3.4) is called a **reaction** because it defines how the state machine reacts to a particular input symbol. Note that exactly one output symbol is produced for each input symbol. Thus, it is not necessary to have access to the entire input sequence to start producing output symbols. This feature proves extremely useful in practice, since it is usually impractical to have access to the entire input sequence (it is infinite in size!). The procedure summarized by (3.3)–(3.4) is **causal**, in that the next state $s(n+1)$ and current output symbol $y(n)$ depend only on the initial state $s(0)$ and current and past input symbols $x(0), x(1), \cdots, x(n)$.

It is sometimes convenient to decompose *update* into two functions:

> $nextState : States \times Inputs \to States$ is the **next state function**,
> $output : States \times Inputs \to Outputs$ is the **output function**.

The interpretation is this. If $s(n)$ is the current state, and $x(n)$ is the current input symbol at step n, the next state is

$$s(n+1) = nextState(s(n), x(n)),$$

and the current output symbol is

$$y(n) = output(s(n), x(n)).$$

Evidently, for all $s(n) \in States$ and $x(n) \in Inputs$,

$$(s(n+1), y(n)) = update(s(n), x(n)) = (nextState(s(n), x(n)), output(s(n), x(n)).$$

3.1.2 Stuttering

A state machine produces exactly one output symbol for each input symbol. For each input symbol, it may also change state (of course, it could also remain in the same state by changing back to the same state). This means that with no input symbol, there is neither an output symbol nor a change of state.

Later, when we compose simpler state machines to construct more complicated ones, it will prove convenient to be explicit in the model about the fact that no input triggers no output and no state change. We do that by insisting that the input and output symbol sets include a **stuttering symbol**, typically denoted *absent*. That is,

$$absent \in Inputs, \text{ and } absent \in Outputs.$$

Moreover, we require that for any $s \in States$,

$$update(s, absent) = (s, absent). \tag{3.5}$$

This is called a **stuttering reaction** because no progress is made. An absent input symbol triggers an absent output symbol and no state change. Now any number of *absent* symbols may be inserted into the input sequence, anywhere, without changing the non-absent output symbols. Stuttering reactions will prove essential for hybrid systems models, considered in chapter 6.

Example 3.1: Consider a 60-minute parking meter. There are three (non-stuttering) input symbols: *in5* and *in25* which represent feeding the meter 5 and 25 cents respectively, and *tick* which represents the passage of one minute. The meter displays the time in minutes remaining before the meter expires. When *in5* occurs, this time is incremented by 5, and when *in25* occurs it is incremented by 25, up to a maximum of 60 minutes. When *tick* occurs, the time is decremented by 1, down to a minimum of 0. When the remaining time is 0, the display reads *expired*.

We can construct a finite state machine model for this parking meter. The set of states is

$$States = \{0, 1, 2, ..., 60\}.$$

The input and output alphabets are

$$Inputs = \{in5, in25, tick, absent\},$$

$$Outputs = \{expired, 1, 2, ..., 60, absent\}.$$

The initial state is

$$initialState = 0.$$

The update function

$$update\colon States \times Inputs \to States \times Outputs$$

is given by, $\forall\ s(n) \in States,\ x(n) \in Inputs$,

$$update(s(n),x(n)) = \begin{cases} (0, expired) & \text{if} \quad x(n) = tick \wedge (s(n) = 0 \vee s(n) = 1) \\ (s(n)-1, s(n)-1) & \text{if} \quad x(n) = tick \wedge s(n) > 1 \\ (\min(s(n)+5, 60), \min(s(n)+5, 60)) & \text{if} \quad x(n) = in5 \\ (\min(s(n)+25, 60), \min(s(n)+25, 60)) & \text{if} \quad x(n) = in25 \\ (s(n), absent) & \text{if} \quad x(n) = absent \end{cases}$$

where min is a function that returns the minimum of its arguments.

If the input sequence is $(in25, tick^{20}, in5, tick^{10}, \cdots)$, for example, then the output sequence is[2]

$$(expired, 25, 24, ..., 6, 5, 10, 9, 8, \cdots, 2, 1, expired, \cdots).$$

3.2 Finite state machines

Often, *States* is a finite set. In this case, the state machine is called a **finite state machine**, abbreviated **FSM**. FSMs yield to powerful analytical techniques because, in principle, it is possible to explore all possible sequences of states. The parking meter above is a finite state machine. The remainder of this chapter and the next chapter will focus on finite state machines. We will return to infinite state systems in chapter 5. Lab C.3 considers software implementation of finite state machines.

When the number of states is small, and the input and output alphabets are finite (and small), we can describe the state machine using a very readable and intuitive diagram called a **state transition diagram**.

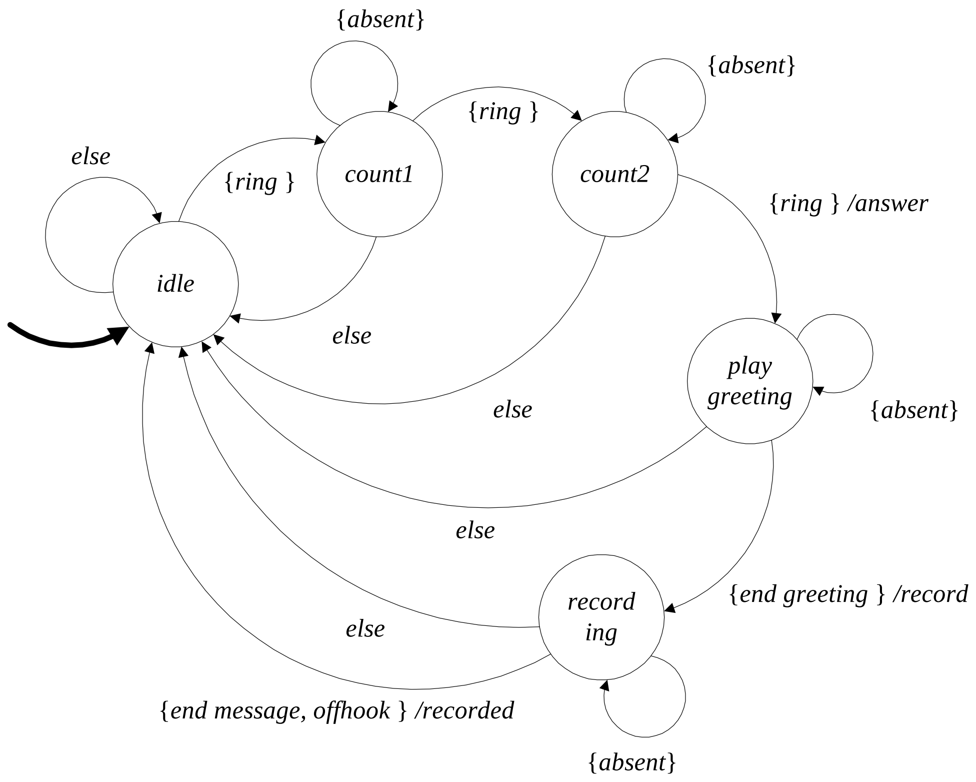
Example 3.2: A verbal description of an automatic telephone answering machine might go like this.

> When a call arrives, the phone rings. If the phone is not picked up, then on the third ring, the machine answers. It plays a pre-recorded greeting requesting that the caller leave a message ("Hello, sorry I can't answer your call right now ... Please leave a message after the beep"), then records the caller's message, and then automatically hangs up. If the phone is answered before the third ring, the machine does nothing.

Figure 3.1 shows a state transition diagram for the state machine model of this answering machine.

You can probably read the diagram in figure 3.1 without any further explanation. It is sufficiently intuitive. Nonetheless, we will explain it precisely.

[2] We are using the common notation $tick^{10}$ to mean a sequence of 10 consecutive *ticks*.

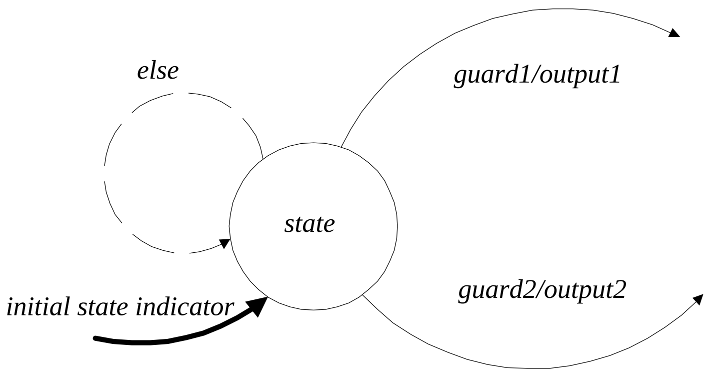


states
idle: nothing is happening *count1*: one ring has arrived *count2*: two rings have arrived *play greeting*: playing the greeting message *recording* : recording the message

inputs
ring - incoming ringing signal *offhook* - a telephone extension is picked up *end greeting* - greeting message is finished playing *end message* - end of message detected (e.g. dialtone) *absent* - no input of interest.

outputs
answer - answer the phone and start the greeting message *record* - start recording the incoming message *recorded*- recorded an incoming message *absent* - default output when there is nothing interesting to say

Figure 3.1: State transition diagram for the telephone answering machine.



state machine:
$(States, Inputs, Outputs, update, initialState)$
$update \colon States \times Inputs \to States \times Outputs$
$initialState \in States$

elements:
$state \in States$
$output1, output2 \in Outputs$
$guard1, guard2 \subset Inputs$
$else = \{i \in Inputs \mid i \notin (guard1 \cup guard2)\}$

determinacy: (There is at most one possible reaction to an input symbol)
$guard1 \cap guard2 = \emptyset$

Figure 3.2: Summary of notation in state transition diagrams, shown for a single state with two outgoing arcs and one self loop.

3.2.1 State transition diagrams

Figure 3.1 consists of bubbles linked by arcs. (The arcs are also called arrows.) In this bubbles-and-arcs syntax each bubble represents one state of the answering machine, and each arc represents a **transition** from one state to another. The bubbles and arcs are annotated, i.e. they are labeled with some text. The execution of the state machine consists of a sequence reactions, where each reaction involves a transition from one state to another (or back to the same state) along one of the arcs. The tables at the bottom of the figure are not part of the state transition diagram, but they improve our understanding of the diagram by giving the meanings of the names of the states, input symbols, and output symbols.

The notation for state transition diagrams is summarized in figure 3.2. Each bubble is labeled with the name of the **state** it represents. The state names can be anything, but they must be distinct. The state machine of figure 3.1 has five states. The state names define the state space,

$$States = \{idle, count1, count2, play\ greeting, recording\}.$$

Each arc is labeled by a **guard** and (optionally) an **output**. If an output symbol is given, it is separated from the guard by a forward slash, as in the example $\{ring\}/answer$ going from state *count2* to *play greeting*. A guard specifies which input symbols might trigger the associated transition. It is a subset of the *Inputs*, the input alphabet, which for the answering machine is

$$Inputs = \{ring, offhook, end\ greeting, end\ message, absent\}.$$

In figure 3.1, some guards are labeled "*else*." This special notation designates an arc that is taken when there is no match on any other guard emerging from a given state. The arc with the guard *else* is called the **else arc**. Thus, *else* is the set of all input symbols not included in any other guard emerging from the state. More precisely, for a given state, *else* is the complement with respect to *Inputs* of the union of the guards on emerging arcs. For example in figure 3.1, for state *recording*,

$$else = \{ring, offhook, end\ greeting\}.$$

For the example in figure 3.2, *else* is defined by

$$else = \{i \in Inputs \mid i \notin (guard1 \cup guard2)\}.$$

If no *else* arc is specified, and the set *else* is not empty, then the *else* arc is implicitly a **self loop**, as shown by the dashed arc in figure 3.2. A self loop is an arc that transitions back to the same state. When the else arc is a self loop, then the stuttering symbol may be a member of the set *else*.

Initially, the system of figure 3.1 is in the *idle* state. The **initial state** is indicated by the bold arc on the left that leads into the state *idle*. Each time an input symbol arrives, the state machine reacts. It checks the guards on arcs going out of the current state and determines which of them contains the input symbol. It then takes that transition.

Two problems might occur.

- The input symbol may not be contained in the guard of any outgoing arc. In our state machine models, for every state, there is at least one outgoing transition that matches the input symbol (because of the *else* arc). This property is called **receptiveness**; it means that the machine can always react to an input symbol. That is, there is always a transition out of the current state that is enabled by the current input symbol. (The transition may lead back to current state if it is a self loop.) Our state machines are said to be **receptive**.
- More than one guard going out from the current state may contain the input symbol. A state machine that has such a structure is said to be **nondeterministic**. The machine is free to choose any arc whose guard contains the input symbol, so more than one behavior is possible for the machine. Nondeterministic state machines will be discussed further below. Until then, we assume that the guards are always defined to give deterministic state machines. Specifically, the guards on outgoing arcs from any state are mutually exclusive. In other words, the intersection of any two guards on outgoing arcs of a state is empty, as indicated in figure 3.2. Of course, by the definition of the *else* set, for any *guard* that is not *else*, it is true that $guard \cap else = \emptyset$.

A sequence of input symbols thus triggers a sequence of state transitions. The resulting sequence of states is called the **state response**.

> **Example 3.3:** In figure 3.1, if the input sequence is
>
> $$(ring, ring, offhook, \cdots)$$
>
> then the state response is
>
> $$(idle, count1, count2, idle, \cdots).$$
>
> The ellipsis ("$\cdots$") are there because the answering machine generally responds to an infinite input sequence, and we are showing only the beginning of that response. This behavior can be compactly represented by a **trace**,
>
> $$idle \xrightarrow{ring} count1 \xrightarrow{ring} count2 \xrightarrow{offhook} idle \cdots$$
>
> A trace represents the state response together with the input sequence that triggers it. This trace describes the behavior of the answering machine when someone picks up a telephone extension after two rings.[3]
>
> A more elaborate trace illustrates the behavior of the answering machine when it takes a message:
>
> $$\begin{aligned} & idle \xrightarrow{ring} count1 \xrightarrow{ring} count2 \xrightarrow{ring} play\ greeting \\ & \xrightarrow{end\ greeting} recording \xrightarrow{end\ message} idle \quad \cdots \end{aligned} \tag{3.6}$$

A state machine also produces outputs. In figure 3.1, the output alphabet is

$$Outputs = \{answer, record, recorded, absent\}.$$

An output symbol is produced as part of a reaction. The output symbol that is produced is indicated after a slash on an arc. If the arc annotation shows no output symbol, then the output symbol is **absent**.

> **Example 3.4:** The output sequence for the trace (3.6) is
>
> $$(absent, absent, answer, record, recorded, \cdots).$$
>
> There is an output symbol for every input symbol, and some of the output symbols are *absent*.

It should be clear how to obtain the state response and output sequence for any input sequence. We begin in the initial state and then follow the state transition diagram to determine the successive state transitions for successive input symbols. Knowing the sequence of transitions, we also know the sequence of output symbols.

[3]When you lift the handset of a telephone to answer, your phone sends a signal called 'offhook' to the telephone switch. The reason for the name 'offhook' is that in the earliest telephone designs, the handset hung from a hook on the side of the phone. In order to answer, you had to pick the handset off the hook. When you finished your conversation you replaced the handset on the hook, generating an 'onhook' signal. The onhook signal is irrelevant to the answering machine, so it is not included in the model.

Shorthand

State transition diagrams can get very verbose, with many arcs with complicated labels. A number of shorthand options can make a diagram clearer by reducing the clutter.

- If no guard is specified on an arc, then that transition is always taken when the state machine reacts and is in the state from which arc emerges, as long as the input is not the stuttering symbol. That is, giving no guard is equivalent to giving the entire set *Inputs* as a guard, minus the stuttering symbol. The stuttering symbol, recall, always triggers a transition back to the same state, and always produces a stuttering symbol on the output.
- Any clear notation for specifying subsets can be used to specify guards. For example, if $Inputs = \{a,b,c\}$, then the guard $\{b,c\}$ can be given by $\neg a$ (read "not a").
- An *else* transition for a state need not be given explicitly. It is an implied self-loop if it is not given. This is why it is shown with a dashed line in figure 3.2.
- The output symbol is the stuttering symbol of *Outputs* if it is not given.

These shorthand notations are not always a good idea. For example, the *else* transitions often correspond to exceptional (unexpected) input sequences, and staying in the same state might not be the right behavior. For instance, in figure 3.1, all *else* transitions are shown explicitly, and all exceptional input sequences result in the machine ending up in state *idle*. This is probably reasonable behavior, allowing the machine to recover. Had we left the *else* transitions implicit, we would likely have ended up with less reasonable behavior. Use your judgment in deciding whether or not to explicitly include *else* transitions.

3.2.2 Update table

An alternative way to describe a finite state machine is by an **update table**. This is simply a tabular representation of the state transition diagram.

For the diagram of figure 3.1, the table is shown in figure 3.3. The first column lists the current state. The remaining columns list the next state and the output symbol for each of the possible input symbols.

The first row, for example, corresponds to the current state *idle*. If the input symbol is *ring*, the next state is *count1* and the output symbol is *absent*. Under any of the other input symbols, the state remains *idle* and the output symbol remains *absent*.

Types of State Machines

The type of state machines introduced in this section are known as **Mealy machines**, after G. H. Mealy, who studied them in 1955. Their distinguishing feature is that output symbols are associated

current state	(*next state, output symbol*) under specified input symbol				
	ring	*offhook*	*end greeting*	*end message*	*absent*
idle	(*count1*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)
count1	(*count2*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)	(*count1*, *absent*)
count2	(*play greeting*, *answer*)	(*idle*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)	(*count2*, *absent*)
play greeting	(*idle*, *absent*)	(*idle*, *absent*)	(*recording*, *record*)	(*idle*, *absent*)	(*play greeting*, *absent*)
recording	(*idle*, *absent*)	(*idle*, *recorded*)	(*idle*, *absent*)	(*idle*, *recorded*)	(*recording*, *absent*)

Figure 3.3: Update table for the telephone answering machine specifies next state and current output symbol as a function of current state and current input symbol.

with state transitions. That is, when a transition is taken, an output symbol is produced. Alternatively, we could have associated output symbols with states, resulting in a model known as **Moore machines**, after F. Moore, who studied them in 1956. In a Moore machine, an output symbol is produced while the machine is in a particular state. Mealy machines turn out to be more useful when they are composed synchronously, as we will do in the next chapter. This is the reason that we choose this variant of the model.

It is important to realize that state machine models, like most models, are not unique. A great deal of engineering judgment goes into a picture like figure 3.1, and two engineers might come up with very different pictures for what they believe to be the same system. Often, the differences are in the amount of detail shown. One picture may show the operation of a system in more detail than another. The less detailed picture is called an **abstraction** of the more detailed picture. Also likely are differences in the names chosen for states, input symbols and output symbols, and even in the meaning of the input and output symbols. There may be differences in how the machine responds to exceptional circumstances (input sequences that are not expected). For example, what should the answering machine do if it gets the input sequence $(\textit{ring}, \textit{end greeting}, \textit{end message})$? This probably reflects a malfunction in the system. In figure 3.1, the reaction to this sequence is easy to see: the machine ends up in the *idle* state.

Given these likely differences, it becomes important to be able to talk about **abstraction relations** and **equivalence relations** between state machine models. This turns out to be a fairly sophisticated topic, one that we touch upon below in section 3.3.

The meaning of state

We have three equivalent ways of describing a state machine: sets and functions, the state transition diagram, and the update table. These descriptions have complementary uses. The table makes obvious the sparsity of output symbols in the answering machine example. The table and the diagrams are both useful for a human studying the system to follow its behavior in different circumstances. The sets and functions and the table are useful for building the state machine in hardware or software. The sets and functions description is also useful for mathematical analysis.

Of course, the tables and the transition diagram can be used only if there are finitely many states and finitely many input and output symbols, i.e. if the sets *States*, *Inputs*, *andOutputs* are finite. The sets and functions description is often equally comfortable with finite and infinite state spaces. We will discuss infinite-state systems in chapter 5.

Like any state machine, the telephone answering machine is a **state-determined** system. Once we know its current state, we can tell what its future behavior is for any future input symbols. We do not need to know what input symbols in the past led to the current state in order to predict how the system will behave in the future. In this sense we can say the current state of the system summarizes the past history of the system. This is, in fact, the key intuitive notion of state.

The number of states equals the number of patterns we need to summarize the past history. If this is intrinsically finite, then a finite-state model exists for the system. If it is intrinsically infinite, then no finite-state model exists. We can often determine which of these two situations applies using simple intuition. We can also show that a system has a finite-state model by finding one. Showing that a system does not have a finite-state model is a bit more challenging.

Example 3.5: Consider the example of a system called *CodeRecognizer* whose input and output signals are sequences of 0 and 1 (with arbitrarily inserted stuttering symbols, which have no effect). The system outputs *recognize* at the end of every subsequence 1100 in the input, and otherwise it outputs *absent*. If the input x is given by a sequence

$$(x(0), x(1), \cdots),$$

and the output y is given by the sequence

$$(y(0), y(1), \cdots),$$

then, if none of the input symbols is *absent*,

$$y(n) = \begin{cases} \mathit{recognize} & \text{if } (x(n-3), x(n-2), x(n-1), x(n)) = (1,1,0,0) \\ \mathit{absent} & \text{otherwise} \end{cases} \tag{3.7}$$

Intuitively, in order to determine $y(n)$, it is enough to know whether the previous pattern of (non-*absent*) inputs is 0, 1, 11, or 110. If this intuition is correct, we can implement *CodeRecognizer* by a state machine with four states that remember the patterns 0, 1, 11, 110. The machine of figure 3.4 does the job. The fact that we have a finite-state machine model of this system shows that this is a finite-state system.

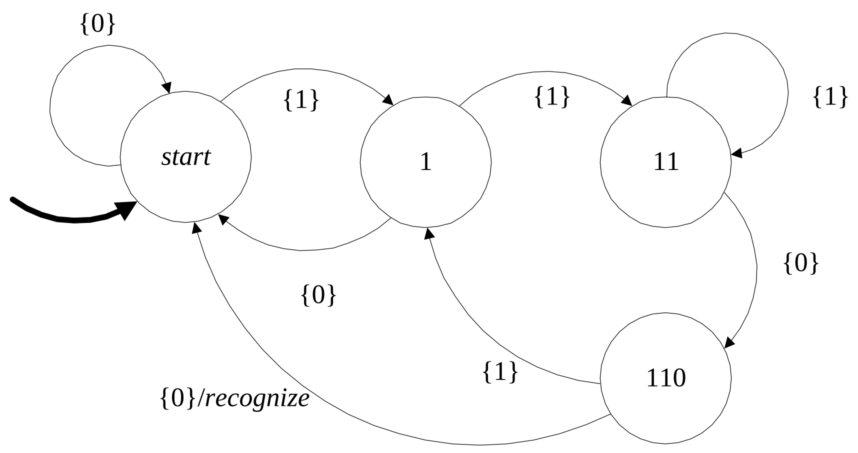


Figure 3.4: A machine that implements *CodeRecognizer*. It outputs *recognize* at the end of every input subsequence 1100, otherwise it outputs *absent*.

The relationship in this example between the number of states and the number of input patterns that need to be stored suggests how to construct functions mapping input sequences to output sequences that cannot be realized by finite state machines. Here is a particularly simple example of such a function called *Equal*.

Example 3.6: An input signal of *Equal* is a sequence of 0 and 1 (again with stuttering symbols arbitrarily inserted). At each step, *Equal* outputs *equal* if the previous inputs contain an equal number of 0's and 1's; otherwise *Equal* outputs *notEqual*. In other words, if the input sequence x is the sequence $(x(0), x(1), \cdots)$, with no stuttering symbols, then the output sequence $y = F(x)$ is given by

$$\forall\, n \in Naturals_0, \quad y(n) = \begin{cases} equal & \text{if number of 1's same as 0's in } x(0), \cdots, x(n) \\ notEqual & \text{otherwise} \end{cases} \tag{3.8}$$

Intuitively, in order to realize *Equal*, the machine must remember the difference between the number of 1's and 0's that have occurred in the past. Since these numbers can be arbitrarily large, the machine must have infinite memory, and so *Equal* cannot be realized by a finite-state machine.

We give a mathematical argument to show that *Equal* cannot be realized by any finite-state machine. The argument uses contradiction.

Suppose that a machine with N states realizes *Equal*. Consider an input sequence that begins with N 1's, $(1, \cdots, 1, x(N), \cdots)$. Let the state response be

$$(s(0), s(1), \cdots, s(N), \cdots).$$

Since there are only N distinct states, and the state response is of length at least $N+1$, the state response must visit at least one state twice. Call that state α. Suppose $s(m) =$

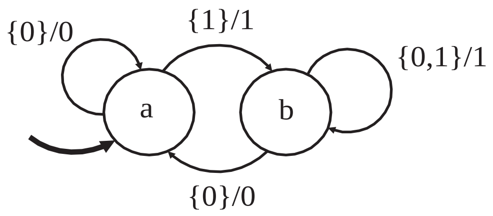


Figure 3.5: A simple nondeterministic state machine.

$s(n) = \alpha$, with $m < n \leq N$. Then the two sequences 1^m0^m and 1^n0^m must lead to the same state, hence yield the same last output symbol on entering state α.[4] But the last output symbol for 1^m0^m should be *equal*, and the last output symbol for 1^n0^m should be *notEqual*, which is a contradiction. So our hypothesis that a finite-state machine realizes *Equal* must be wrong! Exercise 6 asks you to construct an infinite state machine that realizes *Equal*.

3.3 Nondeterministic state machines

There are situations in which it is sufficient to give an incomplete model of a system. Such models are more compact than complete models because they hide inessential details. This compactness will often make them easier to understand.

A useful form of incomplete model is a **nondeterministic state machine**. A nondeterministic state machine often has fewer states and transitions than would be required by a complete model. The state machines we have studied so far are **deterministic**.

3.3.1 State transition diagram

The state transition diagram for a state machine has one bubble for each state and one arc for each state transition. Nondeterministic machines are no different. Each arc is labeled with by "*guard*/*output*," where

$$guard \subset Inputs \text{ and } output \in Outputs.$$

In a deterministic machine, the guards on arcs emerging from any given state are mutually exclusive. That is, they have no common symbols. This is precisely what makes the machine deterministic. For nondeterministic machines, we relax this constraint. Guards can overlap. Thus, a given input symbol may appear in the guard of more than one transition, which means that more than one transition can be taken when that input symbol arrives. This is precisely what makes the machine nondeterministic.

Example 3.7: Consider the state machine shown in figure 3.5. It begins in state *a* and transitions to state *b* the first time it encounters a 1 on the input. It then stays in state *b*

[4]Recall that 1^m means a sequence of m consecutive 1's, similarly for 0^m.

arbitrarily long. If it receives a 1 at the input, it must stay in state b. If it receives a 0, then it can either stay in b or transition to a. Given the input sequence

$$(0,1,0,1,0,1,\cdots)$$

then the following are all possible state responses and output sequences:

$$(a,a,b,a,b,a,b,\cdots)$$
$$(0,1,0,1,0,1,\cdots)$$

$$(a,a,b,b,b,a,b,\cdots)$$
$$(0,1,1,1,0,1,\cdots)$$

$$(a,a,b,b,b,b,b,\cdots)$$
$$(0,1,1,1,1,1,\cdots)$$

$$(a,a,b,a,b,b,b,\cdots)$$
$$(0,1,0,1,1,1,\cdots)$$

Nondeterminism can be used to construct an **abstraction** of a complicated machine, which is a simpler machine that has all the behaviors of the more complicated machine.

Example 3.8: Consider again the 60-minute parking meter. Its input alphabet is

$$\mathit{Inputs} = \{\mathit{coin5}, \mathit{coin25}, \mathit{tick}, \mathit{absent}\}.$$

Upon arrival of *coin5*, the parking meter increments its count by five, up to a maximum of 60 minutes. Upon arrival of *coin25*, it increments its count by 25, again up to a maximum of 60. Upon arrival of *tick*, it decrements its count by one, down to a minimum of zero.

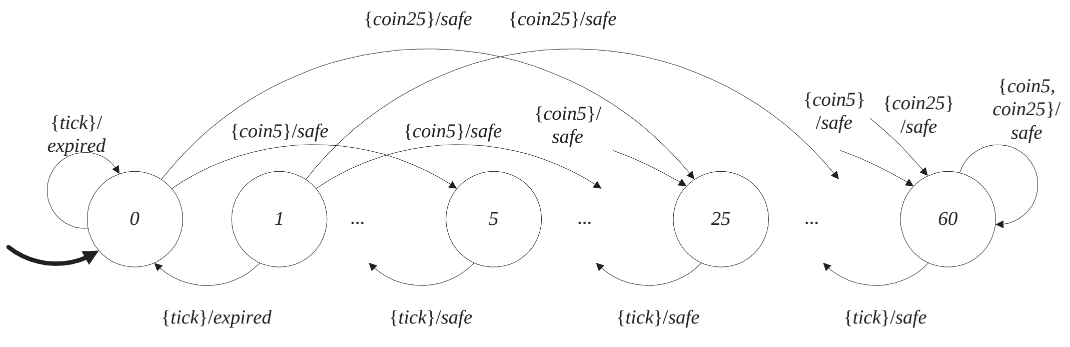
A deterministic state machine model is illustrated schematically in figure 3.6(a). The state space is

$$\mathit{States} = \{0,1,\cdots,60\},$$

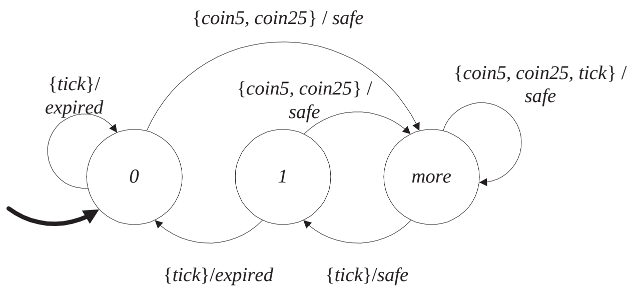
which is too many states to draw conveniently. Thus, patterns in the state space are suggested with ellipsis "$\cdots$".

Suppose that we are interested in modeling the interaction between this parking meter and a police officer. The police officer does not care what state the parking meter is in, except to determine whether the meter has expired or not. Thus, we need only two nonstuttering output symbols, so

$$\mathit{Outputs} = \{\mathit{safe}, \mathit{expired}, \mathit{absent}\}.$$



(a)



(b)

Figure 3.6: Deterministic and nondeterministic models for a 60 minute parking meter.

The symbol *expired* is produced whenever the machine enters state 0.

The model has enough states that a full state transition diagram is tedious and complex enough that it might not be useful for generating insight about the design. Moreover, the detail that is modeled may not add insight about the interaction with a police officer.

Figure 3.6(b) is a nondeterministic model of the same parking meter. It has three states,

$$States = \{0, 1, more\}.$$

The input symbols *coin5* and *coin25* in state 0 or 1 cause a transition to state *more*. The input symbol *tick* in state *more* nondeterministically moves the state to 1 or leaves it in *more*.

The top state machine has more detail than the bottom machine. Shortly, we will give a precise meaning to the phrase 'has more detail' using the concept of *simulation*. For the moment, note that the bottom machine can generate any output sequence that the top machine generates, for the same input sequence. But the bottom machine can also generate output sequences that the top machine cannot. For example, the sequence

$$(expired, safe, safe, expired, \cdots),$$

in which there are two *safe* output symbols between two *expired* output symbols is not a possible output sequence of the top machine, but it is a possible output sequence of the bottom machine. In the top machine, successive *expired* output symbols must be separated by 0 or at least five *safe* output symbols. This detail is not captured by the bottom machine. But in modeling the interaction with a police officer, this detail may not be important, so omitting it may be entirely appropriate.

The machines that we design and build, including parking meters, are usually deterministic. However, the state space of these machines is often very large, much larger than in this example, and it can be difficult to understand their behavior. We use simpler nondeterministic machine models that hide inessential details of the deterministic machine. The analysis of the simpler model reveals some properties, but not all properties, of the more complex machine. The art, of course, is in choosing the model that reveals the properties of interest.

3.3.2 Sets and functions model

The state machines we have been studying, with definitions of the form (3.1), are deterministic. If we know the initial state and the input sequence, then the entire state trajectory and output sequence can be determined. This is because any current state $s(n)$ and current input symbol $x(n)$ uniquely determine the next state and output symbol $(s(n+1), y(n)) = update(s(n), x(n))$.

In a nondeterministic state machine, the next state is not completely determined by the current state and input symbol. For a given current state $s(n)$ and input symbol $x(n)$, there may be more than one next state. So we cannot characterize the machine by the function $update(s(n), x(n))$ because there is no single next state. Instead, we define a function *possibleUpdates* so that $possibleUpdates(s(n), x(n))$

is the *set* of possible next states and output symbols. Whereas a deterministic machine has update function

$$update\colon States \times Inputs \to States \times Outputs,$$

a nondeterministic machine has a (nondeterministic) state transition function

$$possibleUpdates\colon States \times Inputs \to P(States \times Outputs), \tag{3.9}$$

where $P(State \times Outputs)$ is the **power set** of $States \times Outputs$. Recall that the power set is the set of all subsets. That is, any subset of $States \times Outputs$ is an element of $P(States \times Outputs)$.

In order for a nondeterministic machine to be **receptive**, it is necessary that

$$\forall\, s(n) \in States, x(n) \in Inputs \quad possibleUpdates(s(n), x(n)) \neq \emptyset.$$

Recall that a receptive machine accepts any input symbol in any state, makes a state transition (possibly back to the same state), and produces an output symbol. That is, there is no situation where the reaction to an input symbol is not defined.

Operationally, a nondeterministic machine arbitrarily selects the next state and current output symbol from *possibleUpdates* given the current state and current input symbol. The model says nothing about how the selection is made.

Similar to deterministic machines, we can collect the specification of a nondeterministic state machine into a 5-tuple

$$StateMachine = (States, Inputs, Outputs, possibleUpdates, initialState). \tag{3.10}$$

The *possibleUpdates* function is different from the *update* function of a deterministic machine.

Deterministic state machines of the form (3.1) are a special case of nondeterministic machines in which $possibleUpdates(s(n), x(n))$ consists of a single element, namely $update(s(n), x(n))$. In other words,

$$possibleUpdates(s(n), x(n)) = \{update(s(n), x(n))\}.$$

Thus, any deterministic machine, as well as any nondeterministic machine, can be given by (3.10).

In the nondeterministic machine of (3.10), a single input sequence may give rise to many state responses and output sequences. If $(x(0), x(1), x(2), \cdots)$ is an input sequence, then $(s(0), s(1), s(2), \cdots)$ is a (possible) state trajectory and $(y(0), y(1), y(2), \cdots)$ is a (possible) output sequence provided that

$$\begin{array}{rrcl} & s(0) & = & initialState \\ \forall n \geq 0, & (s(n+1), y(n)) & \in & possibleUpdates(s(n), x(n)). \end{array}$$

A deterministic machine defines a function from an input sequence to an output sequence,

$$F\colon InputSignals \to OutputSignals,$$

where

$$InputSignals = [Naturals_0 \to Inputs],$$

and

$$OutputSignals = [Naturals_0 \to Outputs].$$

We define a **behavior** of the machine to be a pair (x, y) such that $y = F(x)$, i.e., a behavior is a possible input, output pair. A deterministic machine is such that for each $x \in InputSignals$, there is exactly one $y \in OutputSignals$ such that (x, y) is a behavior.

We define the set

$$Behaviors \subset InputSignals \times OutputSignals, \tag{3.11}$$

where

$$Behaviors =$$
$$\{(x,y) \in InputSignals \times OutputSignals \mid y \text{ is a possible output sequence for input sequence } x\}.$$

For a deterministic state machine, the set *Behaviors* is the graph of the function F.

For a nondeterministic machine, for a given $x \in InputSignals$, there may be more than one $y \in OutputSignals$ such that (x, y) is a behavior. The set *Behaviors*, therefore, is no longer the graph of a function. Instead, it defines a **relation**—a generalization of a function where there can be two or more distinct elements in the range corresponding to the same element in the domain. The interpretation is still straightforward, however: if $(x, y) \in Behaviors$, then input sequence x may produce output sequence y.

3.4 Simulation and bisimulation

Two different state machines with the same input and output alphabets may be equivalent in the sense that for the same input sequence, they produce the same output sequence. We explore this concept of equivalence in this section.

> **Example 3.9:** The three state machines in figure 3.7, have the same input and output alphabets:
>
> $$Inputs = \{1, absent\} \text{ and } Outputs = \{0, 1, absent\}.$$
>
> Machine (a) has the most states. However, its behavior is identical to that of (b). Both machines produce an alternating sequence of two 1's and one 0 as they receive a sequence of 1's at the input. The machine in (c) is non-deterministic. It has more behaviors than (a) or (b): it can produce any alternating sequence of at least two 1's and one zero. Thus (c) is more general, or more abstract than the machines (a) or (b).

To study the relationships between the machines in figure 3.7 we introduce the concepts of **simulation** and **bisimulation**. The machine in (c) is said to **simulate** (b) and (a). The machine in (b) is said to **bisimulate** (a), or to **be bisimilar to** (a). Bisimulation can be viewed as a form of equivalence between state machines. Simulation can be viewed as a form of abstraction of state machines.

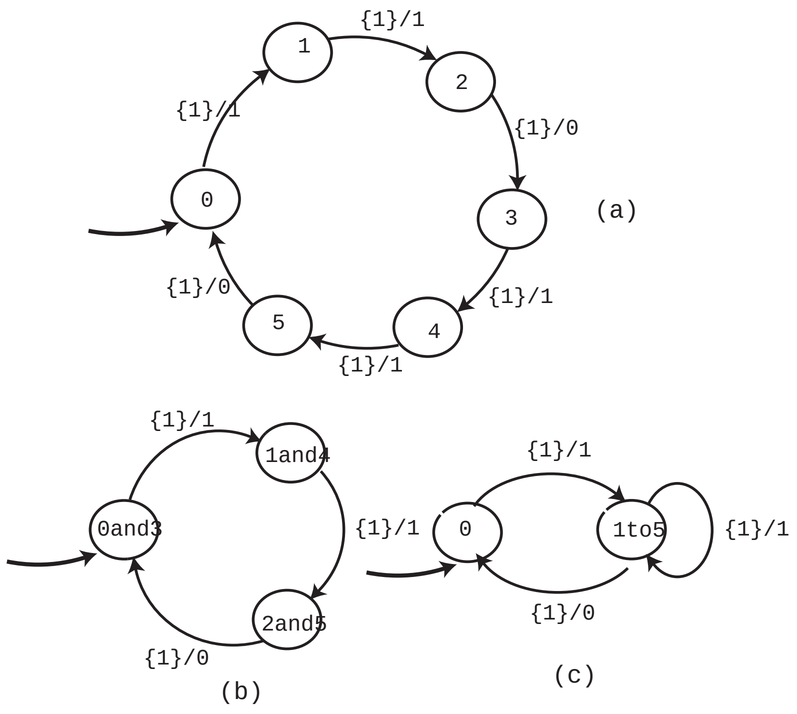


Figure 3.7: Three state machines where (a) and (b) simulate one another and (c) simulates (a) and (b).

Example 3.10: In figure 3.6, the bottom machine can generate any output sequence that the top machine generates, for the same input sequence. The reverse is not true; there are output sequences that the bottom machine can generate that the top machine cannot. The bottom machine is an abstraction of the top one.

We will see that the bottom machine simulates the top machine (but not vice versa).

To understand simulation, it is easiest to consider a **matching game** between one machine and the other, more abstract machine that simulates the first. The game starts with both machines in their initial states. The first machine is allowed to react to an input symbol. If this machine is nondeterministic, it may have more than one possible reaction; it is permitted to choose any one of these reactions. The second, more abstract machine must react to the same input symbol such that it produces the same output symbol. If it is non-deterministic, it is free to pick from among the possible reactions any one that matches the output symbol of the first machine and that will permit it to continue to match the output symbols of the first machine in future reactions. The second machine "wins" this matching game if it can always match the output symbol of the first machine. We then say that the second machine simulates the first one. If the first machine can produce an output symbol that the second one cannot match, then the second machine does not simulate the first machine.

Example 3.11: We wish to determine whether (c) simulates (b) in figure 3.7. The game starts with the two machines in their initial states, which we jointly denote by the pair

$$s_0 = (0and3, 0) \in States_b \times States_c.$$

Machine (b) (the one being simulated) moves first. Given an input symbol, it reacts. If it is nondeterministic, then it is free to react in any way possible, although in this case, (b) is deterministic, so it will have only one possible reaction. Machine (c) then has to **match** the move taken by (b); given the same input symbol, it must react such that it produces the same output symbol.

There are two possible input symbols to machine (b), 1 and *absent*. If the input symbol is *absent*, the machine reacts by stuttering. Machine (c) can match this by stuttering as well. For this example, it will always do to match stuttering moves, so we will not consider them further.

Excluding the stuttering input symbol, there is only one possible input symbol to machine (b), 1. The machine reacts by producing the output symbol 1 and changing to state *1and4*. Machine (c) can match this by taking the only possible transition out of its current state, and also produce output symbol 1. The resulting states of the two machines are denoted

$$s_1 = (1and4, 1to5) \in States_b \times States_c.$$

From here, again there is only one non-stuttering input symbol possible, so (b) reacts by moving to *2and5* and producing the output symbol 1. Now (c) has two choices, but in order to match (b), it chooses the (self-loop) transition which produces 1. The resulting states are

$$s_2 = (2and5, 1to5) \in States_b \times States_c.$$

From here, (b) reacts to the non-stuttering input symbol by moving to *0and3* and producing output symbol 0. To match this move, (c) selects the transition that moves the state to 0, producing 0. The resulting states are s_0, back to where we started. So we know that (c) can always match (b).

The "winning" strategy of the second machine can be summarized by the set

$$S_{b,c} = \{s_0, s_1, s_2\} \subset States_b \times States_c.$$

The set $S_{b,c}$ in this example is called a **simulation relation**; it shows how (c) simulates (b). A simulation relation associates states of the two machines. Suppose we have two state machines, X and Y, which may be deterministic or nondeterministic. Let

$$X = (States_X, Inputs, Outputs, possibleUpdates_X, initialState_X),$$

and

$$Y = (States_Y, Inputs, Outputs, possibleUpdates_Y, initialState_Y).$$

The two machines have the same input and output alphabets. If either machine is deterministic, then its *possibleUpdates* function always returns a set with only one element in it.

If Y simulates X, the simulation relation is given as a subset of $States_X \times States_Y$. Note the ordering here; the machine that moves first in the game, X, the one being simulated, is first in $States_X \times States_Y$.

To consider the reverse scenario, if X simulates Y, then the relation is given as a subset of $States_Y \times States_X$. In this version of the game Y must move first.

If we can find simulation relations in both directions, then the machines are bisimilar.

We can state the "winning" strategy mathematically. We say that Y **simulates** X if there is a subset $S \subset States_X \times States_Y$ such that

1. $(initialState_X, initialState_Y) \in S$, and
2. If $(s_X(n), s_Y(n)) \in S$, then $\forall\, x(n) \in Inputs$, and $\forall\, (s_X(n+1), y_X(n)) \in possibleUpdates_X(s_X(n), x(n))$, there is a $(s_Y(n+1), y_Y(n)) \in possibleUpdates_Y(s_Y(n), x(n))$ such that:
   (a) $(s_X(n+1), s_Y(n+1)) \in S$, and
   (b) $y_X(n) = y_Y(n)$.

This set S, if it exists, is called the simulation relation. It establishes a correspondence between states in the two machines.

Example 3.12: Consider again the state machines in figure 3.7. The machine in (b) simulates the one in (a). The simulation relation is a subset

$$S_{a,b} \subset \{0,1,2,3,4,5\} \times \{0and3, 1and4, 2and5\}.$$

The names of the states in (b) (which are arbitrary) are suggestive of the appropriate simulation relation. Specifically,

$$S_{a,b} = \{(0, \mathit{0and3}), (1, \mathit{1and4}), (2, \mathit{2and5}), (3, \mathit{0and3}), (4, \mathit{1and4}), (5, \mathit{2and5})\}.$$

The first condition of a simulation relation, that the initial states match, is satisfied because $(0, \mathit{0and3}) \in S_{a,b}$. The second condition can be tested by playing the game, starting in each pair of states in $S_{a,b}$.

Start with the two machines in one pair of states in $S_{a,b}$, such as the initial states $(0, \mathit{0and3})$. Then consider the moves that machine (a) can make in a reaction. Ignoring stuttering, if we start with $(0, \mathit{0and3})$, (a) must move to state 1 (given input 1). Given the same input symbol, can (b) match the move? To match the move, it must react to the same input symbol, produce the same output symbol, and move to a state so that the new state of (a) paired with the new state of (b) is in $S_{a,b}$. Indeed, given input symbol 1, (b) produces output symbol 1, and moves to state *1and4* which is matched to state 1 of (a).

It is easy (albeit somewhat tedious) to check that this matching can be done from any starting point in $S_{a,b}$.

This example shows how to use the game to check that a particular subset of $\mathit{States}_X \times \mathit{States}_Y$ is a simulation relation. Thus, the game can be used either to construct a simulation relation or to check whether a particular set is a simulation relation.

For the machines in figure 3.7, we have shown that (c) simulates (b) and that (b) simulates (a). Simulation is **transitive**, meaning that we can immediately conclude that (c) simulates (a). In particular, if we are given simulation relations $S_{a,b} \subset \mathit{States}_a \times \mathit{States}_b$ ((b) simulates (a)) and $S_{b,c} \subset \mathit{States}_b \times \mathit{States}_c$ ((c) simulates (b)), then

$$S_{a,c} = \{(s_a, s_c) \in \mathit{States}_a \times \mathit{States}_c \mid \text{ there exists } s_b \in S_b \text{ where } (s_a, s_b) \in S_{a,b} \text{ and } (s_b, s_c) \in S_{b,c}\} \tag{3.12}$$

is the simulation relation showing that (c) simulates (a).

Example 3.13: For the examples in figure 3.7, we have already determined

$$S_{a,b} = \{(0, \mathit{0and3}), (1, \mathit{1and4}), (2, \mathit{2and5}), (3, \mathit{0and3}), (4, \mathit{1and4}), (5, \mathit{2and5})\}.$$

and

$$S_{b,c} = \{(\mathit{0and3}, 0), (\mathit{1and4}, \mathit{1to5}), (\mathit{2and5}, \mathit{1to5})\}.$$

From (3.12) we can conclude that

$$S_{a,c} = \{(0,0), (1, \mathit{1to5}), (2, \mathit{1to5}), (3, 0), (4, \mathit{1to5}), (5, \mathit{1to5})\},$$

which further supports the suggestive choices of state names.

Simulation relations are not (necessarily) symmetric.

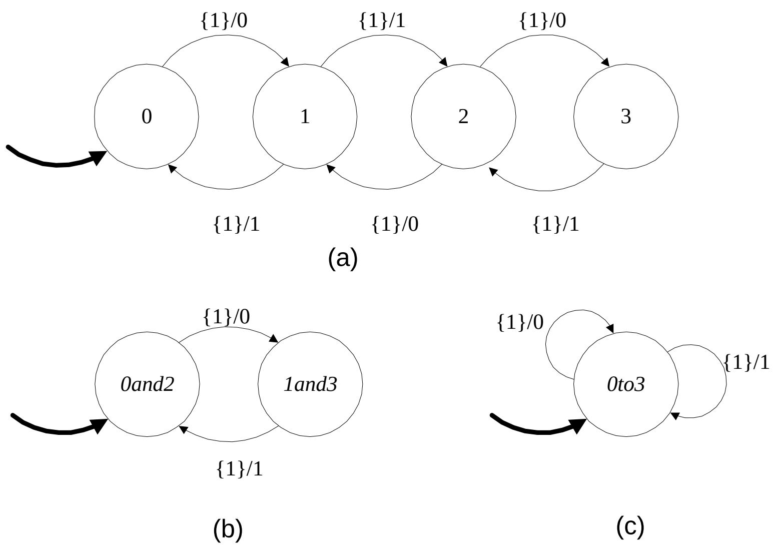


Figure 3.8: Three state machines where (a) and (b) are bisimilar and (c) simulates (a) and (b)

Example 3.14: For the examples in figure 3.7, (b) does not simulate (c). To see this, we can attempt to construct a simulation relation by playing the game. Starting in the initial states,

$$s_0 = (\mathit{0and3}, 0),$$

we allow (c) to move first. Presented with a nonstuttering input symbol, 1, it produces 1 and moves to *1to5*. Machine (b) can match this by producing 1 and moving to *1and4*. But from state *1to5*, (c) can now produce 0 with input symbol 1, which (b) cannot match. Thus, the game gets stuck, and we fail to construct a simulation relation.

Consider another example, one that illustrates that there may be more than one similarity relation between two machines.

Example 3.15: In figure 3.8, it is easy to check that (c) simulates (a) and (b). We now verify that (b) simulates (a) and also (a) simulates (b) by determining that not only can (b) match any move (a) makes, but (a) can also match any move (b) makes. In fact, since (a) is nondeterministic, in two of its states it has two distinct ways of matching the moves of (b). Since it moves second, it can arbitrarily choose from among these possibilities.

If from state 1 it always chooses to return to state 0, then the simulation relation is

$$S_{b,a} = \{(\mathit{0and2}, 0), (\mathit{1and3}, 1)\}.$$

Otherwise, if from state 2 it always chooses to return to state 1, then the simulation relation is

$$S_{b,a} = \{(\mathit{0and2}, 0), (\mathit{1and3}, 1), (\mathit{0and2}, 2)\}.$$

Otherwise, the simulation relation is

$$S_{b,a} = \{(\mathit{0and2}, 0), (\mathit{1and3}, 1), (\mathit{0and2}, 2), (\mathit{1and3}, 3)\}.$$

Thus, the simulation relation is not unique.

A common use of simulation is to establish a relationship between a more abstract model and a more detailed model. In the example above, (c) is a more abstract model of either (b) or (a). It is more abstract in the sense that it loses detail. For example, it has lost the property that 0's and 1's alternate in the output sequence. We now give a more compelling example of such abstraction, where the abstraction dramatically reduces the number of states while still preserving some properties of interest.

Example 3.16: In the case of the parking meter, the bottom machine in figure 3.6 simulates the top machine. Let A denote the top machine, and let B denote the bottom machine. We will now identify the simulation relation.

The simulation relation is a subset $S \subset \{0, 1, \cdots, 60\} \times \{0, 1, \mathit{more}\}$. It is intuitively clear that 0 and 1 of the bottom machine correspond to 0 and 1, respectively, of the top machine. Thus, $(0,0) \in S$ and $(1,1) \in S$. It is also intuitive that *more* corresponds to all of the remaining states $2, \cdots 60$ of the top machine. So we propose to define the simulation relation as

$$S = \{(0,0), (1,1)\} \cup \{(s_A, \mathit{more}) \mid 2 \leq s_A \leq 60\} \tag{3.13}$$

We now check that S is indeed a simulation relation, as defined above.

The first condition of a simulation relation, that the initial states match, is satisfied because $(0,0) \in S$. The second condition is more tedious to verify. It says that for each pair of states in S, and for each input symbol, the two machines can transition to a pair of new states that is also in S, and that these two transitions produce the same output symbol. Since machine A is deterministic, there is no choice about which transition it takes and which output symbol it produces. In machine B, there are choices, but all we require is that one of the choices match.

The only state of machine B that actually offers choices is *more*. Upon receiving *tick*, the machine can transition back to *more* or down to 1. In either case, the output symbol is *safe*. It is easy to see that these two choices are sufficient for state *more* to match states $2, 3, \ldots 60$ of machine A.

Thus the bottom machine indeed simulates the top machine with the simulation relation (3.13).

3.4.1 Relating behaviors

A simulation relation establishes a correspondence between two state machines, one of which is typically much simpler than the other. The relation lends confidence that analyzing the simpler machine indeed reveals properties of the more complicated machine.

This confidence rests on a theorem and corollary that we will develop in this section. These results relate the input/output behaviors of state machines that are related by simulation.

Given an input sequence $x = (x(0), x(1), x(2), \cdots)$, if a state machine can produce the output sequence $y = (y(0), y(1), y(2), \cdots)$, then (x, y) is said to be a behavior of the state machine. The set of all behaviors of a state machine obviously satisfies

$$\mathit{Behaviors} \subset \mathit{InputSignals} \times \mathit{OutputSignals}.$$

Theorem Let B simulate A. Then

$$\mathit{Behaviors}_A \subset \mathit{Behaviors}_B.$$

This theorem is easy to prove. Consider a behavior $(x, y) \in \mathit{Behaviors}_A$. We need to show that $(x, y) \in \mathit{Behaviors}_B$.

Let the simulation relation be S. Find all possible state responses for A

$$s_A = (s_A(0), s_A(1), \cdots)$$

that result in behavior (x, y). (If A is deterministic, then there will be only one.) The simulation relation assures us that we can find a state response for B

$$s_B = (s_B(0), s_B(1), \cdots)$$

where $(s_A(i), s_B(i)) \in S$, such that given input symbol x, B produces y. Thus, $(x, y) \in \mathit{Behaviors}_B$.

Intuitively, the theorem simply states that B can match every move of A and produce the same output sequence. It also implies that if B cannot produce a particular output sequence, then neither can A. This is stated formally in the following corollary.

Corollary Let B simulate A. Then if

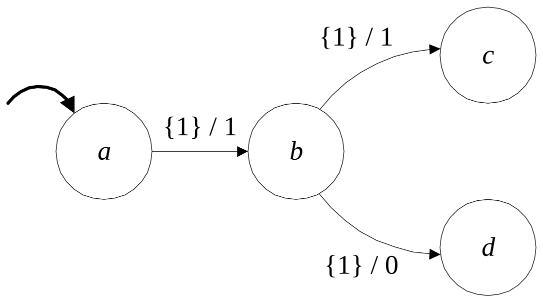
$$(x, y) \notin \mathit{Behaviors}_B$$

then

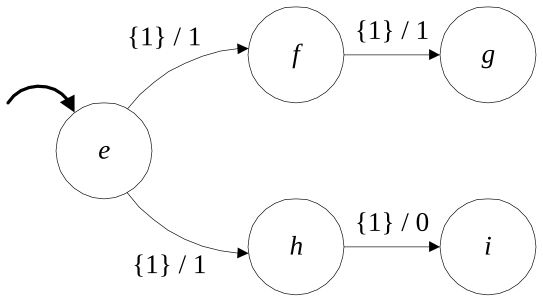
$$(x, y) \notin \mathit{Behaviors}_A.$$

The theorem and corollary are useful for analysis. The general approach is as follows. We have a state machine A. We wish to show that its input-output function satisfies some property. That is, every behavior satisfies some condition. We construct a simpler machine B whose input-output relation satisfies the same property, and where B simulates A. Then the theorem guarantees that A will satisfy this property, too. That is, since all behaviors of B satisfy the property, all behaviors of A must also. This technique is useful since it is often easier to understand a simple state machine than a complex state machine with many states.

Conversely, if there is some property that we must assure that no behavior of A has, it is sufficient to find a simpler machine B which simulates A and does not have this property. This scenario is typical of a **safety** problem, where we must show that dangerous outputs from our system are not possible.



(a)



(b)

Figure 3.9: Two state machines with the same behaviors that are not bisimilar.

Example 3.17: For the parking meter of figure 3.6, for example, we can use the nondeterministic machine to show that if a coin is inserted at step n, the output symbol at steps n and $n+1$ is *safe*. By the corollary, it is sufficient to show that the nondeterministic machine cannot do any differently.

It is important to understand what the theorem says, and what it does not say. It does not say, for example, that if $Behaviors_A \subset Behaviors_B$ then B simulates A. In fact, this statement is not true. Consider the two machines in figure 3.9, where

$$Inputs = \{1, absent\},$$

$$Outputs = \{0, 1, absent\}.$$

These two machines have the same behaviors. The non-stuttering output symbols are $(1,0)$ or $(1,1)$, selected nondeterministically, assuming the input sequence has at least two non-stuttering symbols. However, they are not bisimilar. In particular, (b) does not simulate (a). To see this, we play the matching game. Machine (a) is allowed to move first. Ignoring stuttering, it has no choice but to move from a to b and produce output symbol 1. Machine (b) can match this two ways; it has no basis upon which to prefer one way to match it over another, so it picks one, say moving to state f. Now it is the turn of machine (a), which has two choices. If it choses to move to d, then machine (b) cannot match its move. (A similar argument works if (b) picks state h.) Thus, machine (b) does not simulate machine (a), despite the fact that $Behaviors_A \subset Behaviors_B$.[5]

3.5 Summary

State machines are models of systems whose input and output signal spaces consist of sequences of symbols. There are three ways of defining state machines: sets and functions, state transition diagram, and the update table. The state machine model gives a step-by-step procedure for evaluating the output signal. This is a state-determined system: once we know the current state, we can tell the future behavior for any future input symbols.

A state machine can be non-deterministic: given the current state and current input symbol, it may have more than one possible next state and current output symbol. Non-determistic machines typically arise through abstraction of deterministic machines. Two state machines, with the same input and output alphabets, may be related through simulation and bisimulation. Simulation is used to understand properties of the behavior of one machine in terms of the behaviors of another (presumably simpler) machine. Bisimulation is used to establish a form of equivalence between state machines.

Exercises

In some of the following exercises you are asked to design state machines that carry out a given task. The design is simple and elegant if the state space is properly chosen. Although the state space is not unique, there often is a natural choice. As usual, each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **E** A state machine with
$$Inputs = \{a, b, c, d, absent\},$$
has a state s with two emerging arcs with guards
$$guard1 = \{a\}$$
and
$$guard2 = \{a, b, d\}.$$
   (a) Is this state machine deterministic?
   (b) Define the set *else* for state s and specify the source and destination state for the *else* arc.

[5]Recall that in our notation $\subset$ allows the two sets to be equal.

2. **E** For the answering machine example of figure 3.1, assume the input sequence is

$$(\mathit{offhook}, \mathit{offhook}, \mathit{ring}, \mathit{offhook}, \mathit{ring}, \mathit{ring}, \mathit{ring}, \mathit{offhook}, \cdots).$$

   This corresponds to a user of the answering machine making two phone calls, answering a third after the first ring, and answering a second after the third ring.

   (a) Give the state response of the answering machine.
   (b) Give the trace of the answering machine.
   (c) Give the output sequence.

3. **E** Consider the alphabets

$$\mathit{Inputs} = \mathit{Outputs} = \mathit{Binary} = \{0, 1\}.$$

   Note that there is no stuttering input or output symbols here. This simplifies the notation in the problem somewhat.

   (a) Construct a state machine that uses these alphabets such that if $(x(0), x(1), \cdots)$ is any input sequence without stuttering symbols, the output sequence is given by

$$\forall\, n \in \mathit{Naturals}_0, \quad y(n) = \begin{cases} 1 & \text{if } n \geq 2 \wedge (x(n-2), x(n-1), x(n)) = (1,1,1) \\ 0 & \text{otherwise} \end{cases}$$

   In words, the machine outputs 1 if the current input symbol and the two previous input symbols are all 1's, otherwise it outputs 0. (Had we included a stuttering symbol, the above equation would be a bit more complicated.)

   (b) For the same input and output alphabet, construct a state machine that outputs 1 if the current input symbol and two previous input symbols are either $(1,1,1)$ or $(1,0,1)$, and otherwise it outputs 0.

4. **E** A modulo N counter is a device that can output any integer between 0 and $N-1$. The device has three input symbols, *increment*, *decrement*, and *reset*, plus, as always, a stuttering symbol *absent*; *increment* increases the output integer by 1; *decrement* decreases this integer by 1; and *reset* sets the output symbol to 0. Here *increment* and *decrement* are modulo N operations.

   Note: Modulo N numbers work as follows. For any integer m, $m \bmod N = k$ where $0 \leq k \leq N-1$ is the unique integer such that N divides $(m-k)$. Thus there are only N distinct modulo-N numbers, namely, $0, \cdots, N-1$.

   (a) Give the state transition diagram of this counter for $N = 4$.
   (b) Give the update table of this counter for $N = 4$.
   (c) Give a description of the state machine by specifying the five entities that appear in (3.1); again assume $N = 4$.
   (d) Take $N = 3$. Calculate the state response for the input sequence

$$(\mathit{increment}^4, \mathit{decrement}^3, \cdots)$$

   starting with initial state 1, where s^n means s repeated n times.

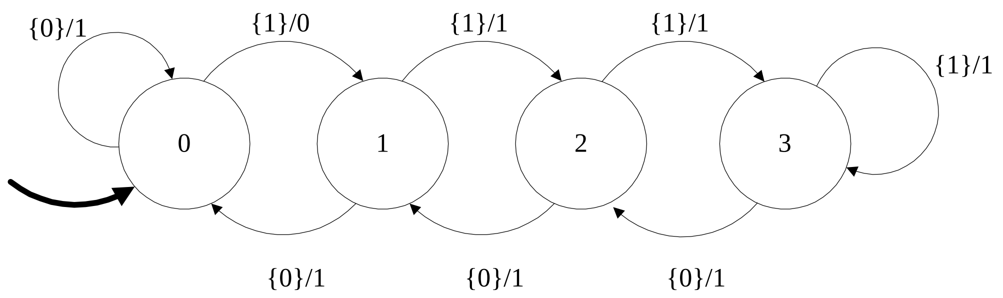


Figure 3.10: Machine that outputs at least one 1 between any two 0's.

5. **T** The state machine *UnitDelay* is defined to behave as follows. On the first non-stuttering reaction (when the first non-stuttering input symbol arrives), the output symbol *a* is produced. On subsequent reactions (when subsequent input symbols arrive), the input symbol that arrived at the previous non-stuttering reaction is produced as an output symbol.

   (a) Assume the input and output alphabets are

   $$Inputs = Outputs = \{a, b, c, absent\}.$$

   Give a finite state machine that implements *UnitDelay* for this input set. Give both a state transition diagram and a definition of each of the components in (3.1).

   (b) Assume the input and output sets are

   $$Inputs = Outputs = Naturals_0 \cup \{absent\},$$

   and that on the first non-stuttering reaction, the machine produces 0 instead of *a*. Give an (informal) argument that no finite state machine can implement *UnitDelay* for this input set. Give an infinite state machine by defining each of the components in (3.1).

6. **T** Construct an infinite state machine that realizes *Equal*.

7. **C** An elevator connects two floors, 1 and 2. It can go up (if it is on floor 1), down (if it is on floor 2) and stop on either floor. Passengers at any floor may press a button requesting service. Design a controller for the elevator so that (1) every request is served, and (2) if there is no pending request, the elevator is stopped. For simplicity, do not be concerned about responding to requests from passengers inside the elevator.

8. **T** The state machine in figure 3.10 has the property that it outputs at least one 1 between any two 0's. Construct a two-state nondeterministic state machine that simulates this one and preserves that property.

9. **T** For the nondeterministic state machine in figure 3.11 the input and output alphabets are

   $$Inputs = Outputs = \{0, 1, absent\}.$$

   (a) Define the *possibleUpdates* function (3.9) for this state machine.

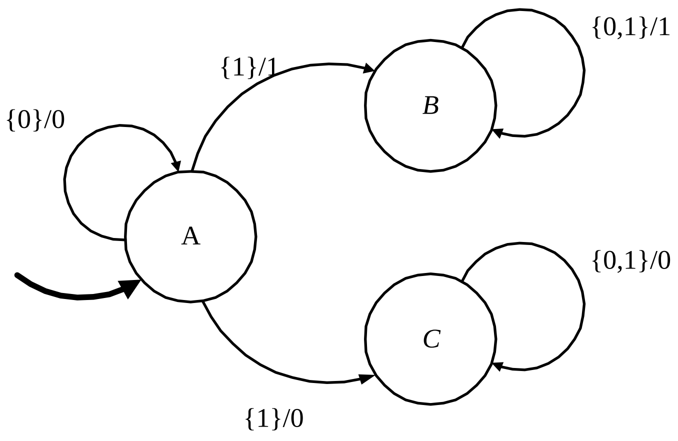


Figure 3.11: Nondeterministic state machine for exercise 9.

(b) Define the relation *Behaviors* in (3.11) for this state machine. Part of the challenge here is to find a way to describe this relation compactly. For simplicity, ignore stuttering; i.e. assume the input symbol is never *absent*.

10. **E** The state machine in figure 3.12 implements *CodeRecognizer*, but has more states than the one in figure 3.4. Show that it is equivalent by giving a bisimulation relation with the machine in figure 3.4.

11. **E** The state machine in figure 3.13 has input and output alphabets

$$Inputs = \{1, a\},$$

$$Outputs = \{0, 1, a\},$$

where a (short for *absent*) is the stuttering symbol. State whether each of the following is in the set *Behaviors* for this machine. In each of the following, the ellipsis "$\cdots$" means that the last symbol is repeated forever. Also, in each case, the input and output signals are given as sequences.

(a) $((1,1,1,1,1,\cdots),(0,1,1,0,0,\cdots))$
(b) $((1,1,1,1,1,\cdots),(0,1,1,0,a,\cdots))$
(c) $((a,1,a,1,a,\cdots),(a,1,a,0,a,\cdots))$
(d) $((1,1,1,1,1,\cdots),(0,0,a,a,a,\cdots))$
(e) $((1,1,1,1,1,\cdots),(0,a,0,a,a,\cdots))$

12. **E** The state machine in figure 3.14 has

$$Inputs = \{1, absent\},$$

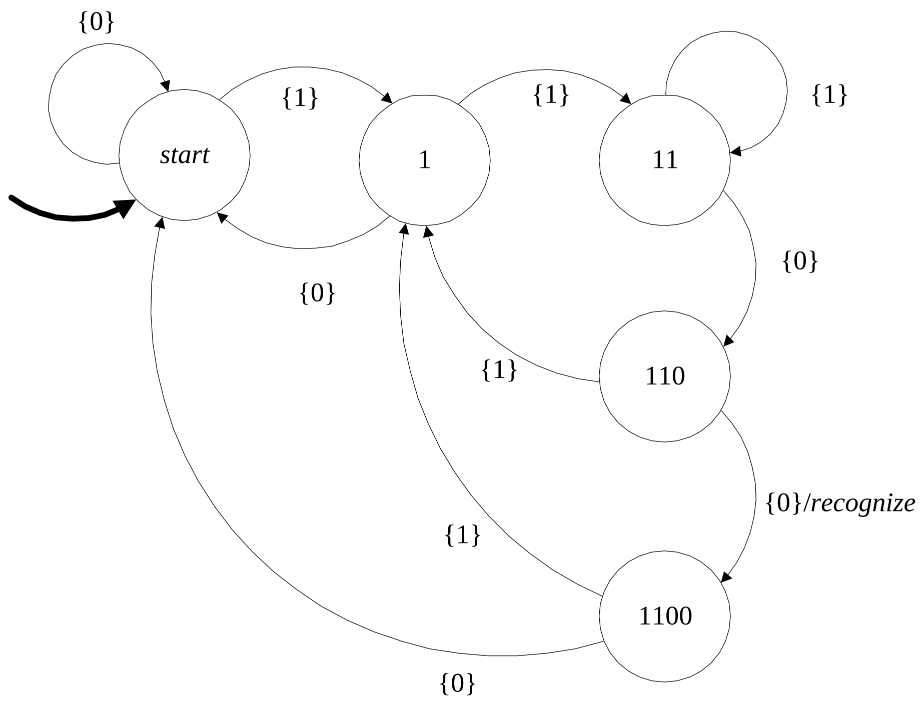


Figure 3.12: A machine that implements *CodeRecognizer*, but has more states than the one in figure 3.4.

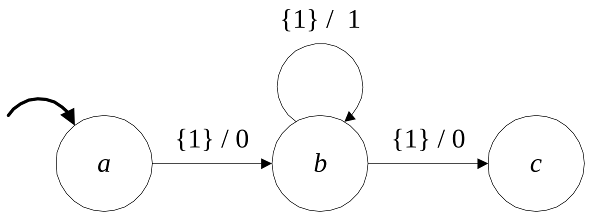


Figure 3.13: State machine for problem 11.

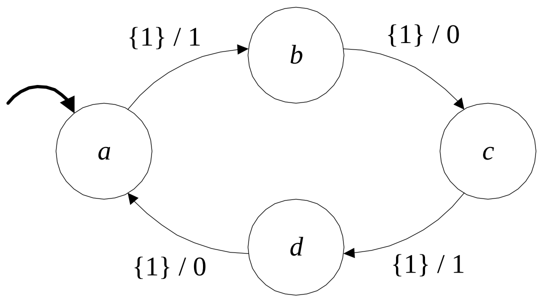


Figure 3.14: A machine that has more states than it needs.

$$Outputs = \{0, 1, absent\}.$$

Find a bisimilar state machine with only two states, and give the bisimulation relation.

13. **E** You are told that state machine A has

$$Inputs = \{1, 2, absent\},$$

$$Outputs = \{1, 2, absent\},$$

$$States = \{a, b, c, d\}.$$

but you are told nothing further. Do you have enough information to construct a state machine B that simulates A? If so, give such a state machine, and the simulation relation.

14. **E** Construct a state machine with $Inputs = \{0, 1, absent\}$, $Outputs = \{r, absent\}$, that outputs r whenever the input signal (without stuttering symbols) contains the sequence $(0, 0, 0)$, otherwise it outputs *absent*. More precisely, if $x = (x(0), x(1), \cdots)$ is the input sequence then $y = (y(0), y(1), \cdots)$ is the output sequence, where

$$y(n) = \begin{cases} r, & \text{if } (x(n-2), x(n-1), x(n)) = (0,0,0) \\ absent & \text{otherwise} \end{cases}$$

15. **T** Consider a state machine where

$$Inputs = \{1, absent\},$$

$$Outputs = \{0, 1, absent\},$$

$$States = \{a, b, c, d, e, f\},$$

$$initialState = a,$$

and the *update* function is given by the following table (ignoring stuttering):

$(currentState, inputSymbol)$	$(nextState, outputSymbol)$
$(a, 1)$	$(b, 1)$
$(b, 1)$	$(c, 0)$
$(c, 1)$	$(d, 0)$
$(d, 1)$	$(e, 1)$
$(e, 1)$	$(f, 0)$
$(f, 1)$	$(a, 0)$

(a) Draw the state transition diagram for this machine.

(b) Ignoring stuttering, give the *Behaviors* relation for this machine.

(c) Find a state machine with three states that is bisimilar to this one. Draw that state machine, and give the bisimulation relation.

Chapter 4

Composing State Machines

We design interesting systems by composing simpler components. Since systems are functions, their composition is function composition, as discussed in section 2.1.5. State machines, however, are not given directly as functions that map input sequences into output sequences. Instead, they are given procedurally, where the *update* function defines how to progress from one state to the next. This chapter explains how to define a new state machine that describes a composition of multiple state machines.

In section 2.3.4 we used a block diagram syntax to define compositions of systems. We will use the same syntax here, and we will similarly build up an understanding of composition by first considering easy cases. The hardest cases are those where there is feedback, because the input of one state machine may depend on its own output. It is challenging in this case to come up with a procedure for updating the state of the composite machine. For some compositions, in fact, it isn't even possible. Such compositions are said to be ill-formed.

4.1 Synchrony

We consider a set of interconnected components, where each component is a state machine. By "interconnected" we mean that the outputs of one component may be inputs of another. We wish to construct a state machine model for the composition of components. Composition has two aspects. The first aspect is straightforward: it specifies which outputs of one component are the inputs of another component. These input-output connections are specified using block diagrams.

The second aspect of composition concerns the timing relationships between inputs and outputs. We choose a particular style of composition called **synchrony**. This style dictates that each state machine in the composition reacts *simultaneously* and *instantaneously*. So a reaction of the composite machine consists of a set of simultaneous reactions of each of the component machines.

A reaction of the composite machine is triggered by inputs from the environment. Thus, *when* a reaction occurs is externally determined. This is the same as for a single machine. As with a single state machine, a composite machine may stutter. This simply means that each component machine stutters.

A system that reacts only in response to external stimulus is said to be **reactive**. Because our compositions are synchronous, they are often called **synchronous/reactive** systems.

The reactions of the component machines and of the composite machine are viewed as being instantaneous. That is, a reaction does not take time. In particular, the output symbol from a state machine is viewed as being *simultaneous* with the input symbol, without delay. This creates some interesting subtleties, especially for feedback composition when the input of a state machine is connected to its own output. We will discuss the ramifications of the synchronous/reactive interpretation below.

Synchrony is a very useful model of the behavior of physical systems. Digital circuits, for example, are almost always designed using this model. Circuit elements are viewed as taking no time to calculate their outputs given their inputs, and time overall is viewed as progressing in a sequence of discrete time steps according to ticks of a clock. Of course, the time that it takes for a circuit element to produce an output cannot ultimately be ignored, but the model is useful because for most circuit elements in a complex design, this time *can* be ignored. Only the time delay of the circuit elements along a **critical path** affects the overall performance of the circuit.

More recently than for circuits, synchrony has come to be used in software as well. Concurrent software modules interact according to the synchronous model. Languages built on this principle are called **synchronous languages**. They are used primarily in real-time embedded system[1] design.

4.2 Side-by-side composition

A simple form of composition of two state machines is shown in figure 4.1. We call this **side-by-side composition**. Side-by-side composition in itself is not useful, but it is useful in combination with other types of composition. The two state machines in figure 4.1 do not interact with one another. Nonetheless we wish to define a single state machine representing the synchronous operation of the two component state machines.

The state space of the composite state machine is simply

$$States = States_A \times States_B.$$

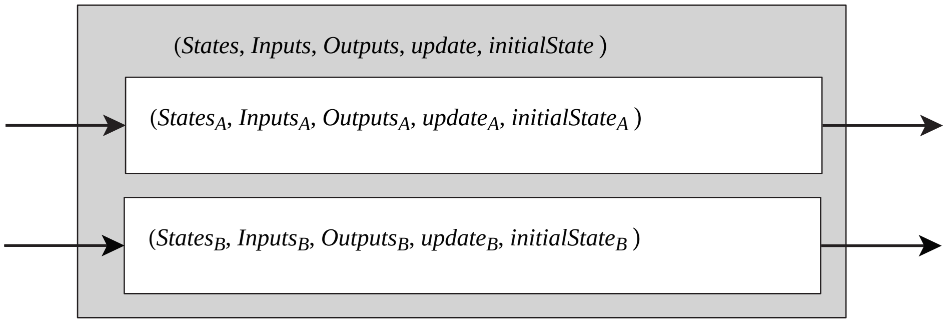
We could take the cross product in the opposite order, resulting in a *different* but *bisimilar* composite state machine. The initial state is

$$initialState = (initialState_A, initialState_B).$$

The input and output alphabets are

$$Inputs = Inputs_A \times Inputs_B, \tag{4.1}$$

[1]An **embedded system** is a computing system (a computer and its software) that is embedded in a larger system that is not first and foremost a computer. A digital cellular telephone, for example, contains computers that realize the radio modem and the speech codec. Recent cars contain computers for ignition control, anti-lock brakes, and traction control. Aircraft contain computers for navigation and flight control. In fact, most modern electronic controllers of physical systems are realized as embedded systems.


(States, Inputs, Outputs, update, initialState)
($States_A$, $Inputs_A$, $Outputs_A$, $update_A$, $initialState_A$)
($States_B$, $Inputs_B$, $Outputs_B$, $update_B$, $initialState_B$)

Definition of the side-by-side composite machine:
$States = States_A \times States_B$
$Inputs = Inputs_A \times Inputs_B$
$Outputs = Outputs_A \times Outputs_B$
$initialState = (initialState_A, initialState_B)$
$((s_A(n+1), s_B(n+1)), (y_A(n), y_B(n)))$
$= update((s_A(n), s_B(n)), (x_A(n), x_B(n))),$

where

$(s_A(n+1), y_A(n)) = update_A(s_A(n), x_A(n))$ and
$(s_B(n+1), y_B(n)) = update_B(s_B(n), x_B(n))$

Figure 4.1: Summary of side-by-side composition of state machines.

$$Outputs = Outputs_A \times Outputs_B. \tag{4.2}$$

The update function of the composite machine, *update*, consists of the update functions of the component machines, side-by-side:

$$((s_A(n+1), s_B(n+1)), (y_A(n), y_B(n))) = update((s_A(n), s_B(n)), (x_A(n), x_B(n))),$$

where

$$(s_A(n+1), y_A(n)) = update_A(s_A(n), x_A(n)),$$

and

$$(s_B(n+1), y_B(n)) = update_B(s_B(n), x_B(n)).$$

Recall that $Inputs_A$ and $Inputs_B$ include a stuttering element. This is convenient because it allows a reaction of the composite when we really want only one of the machines to react. Suppose the stuttering elements are $absent_A$ and $absent_B$. Then if the second component of the input symbol is $absent_B$, the reaction of the composite consists only of the reaction of the first machine. The stuttering element of the composite is the pair of stuttering elements of the component machines, $(absent_A, absent_B)$.

Example 4.1: The side-by-side composition in the top of figure 4.2 has the composite machine with state space

$$States = States_A \times States_B = \{(1,1), (2,1)\},$$

and alphabets

$$Inputs = \{(0,0), (1,0), (absent_A, 0), (0, absent_B), (1, absent_B), (absent_A, absent_B)\},$$

$$Outputs = \{(a,c), (b,c), (absent_A, c), (a, absent_B), (b, absent_B), (absent_A, absent_B)\}.$$

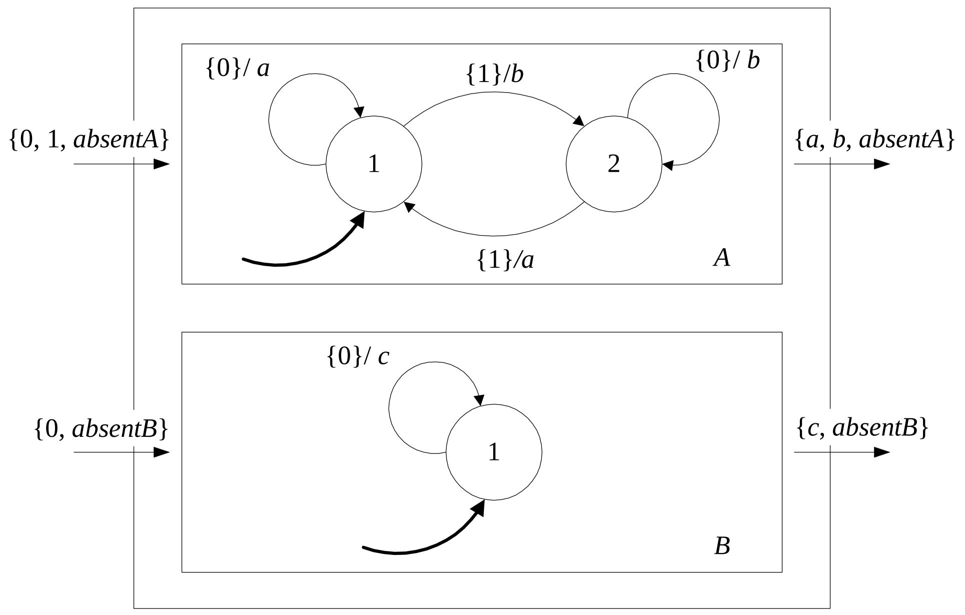
The initial state is

$$initialState = (1,1).$$

The *update* function can be given as a table, only a part of which is displayed below. The state transition diagram in the lower part of figure 4.2 gives the same part of the update function.

current	(*next state*, *output*) for input			
state	(0,0)	(1,0)	($absent_A$,0)	$\cdots$
(1,1)	$((1,1),(a,c))$	$((2,1),(b,c))$	((1,1)(($absent_A$, c))	$\cdots$
(2,1)	$((2,1),(b,c))$	$((1,1),(a,c))$	((2,1), (($absent_A$, c))	$\cdots$

Notice that if the second component of the input sequence is always $absent_B$, then the side-by-side composition behaves essentially as machine A, and if the first component is always $absent_A$, then it behaves as machine B. The stuttering element of the composite is of course the pair $(absent_A, absent_B)$.



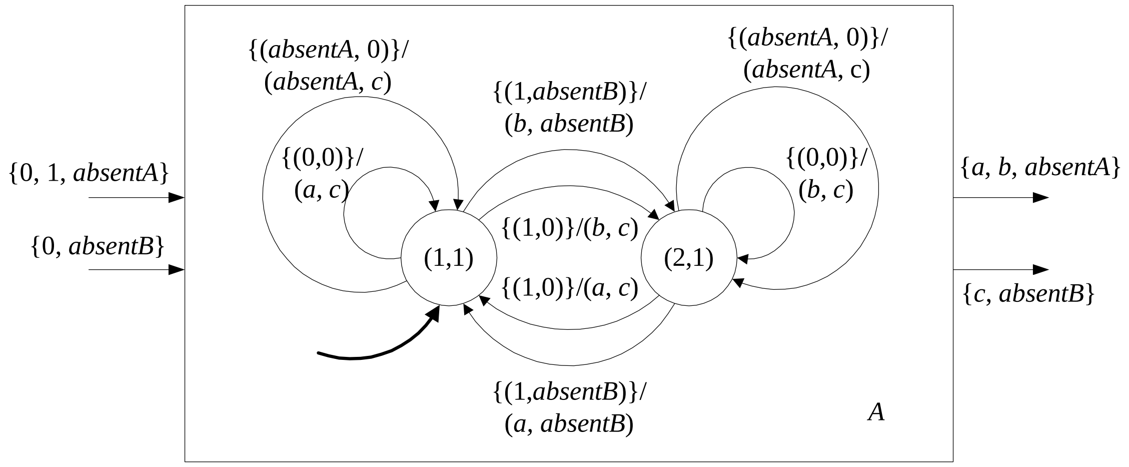
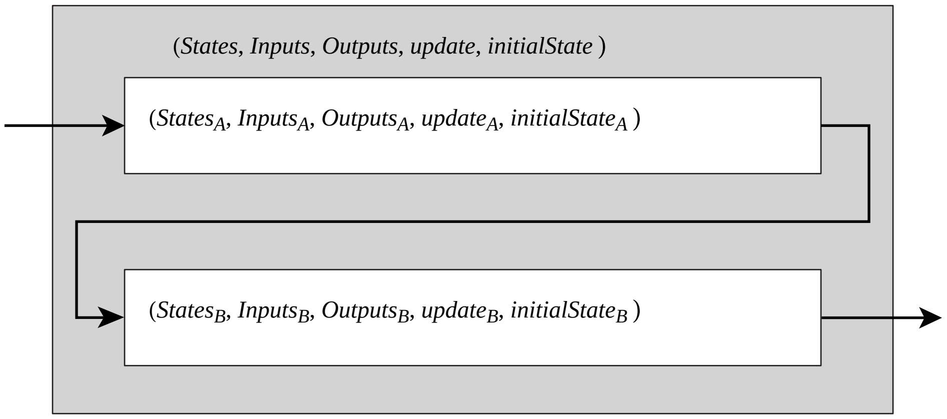


Figure 4.2: Example of a side-by-side composition.


(States, Inputs, Outputs, update, initialState)
(States_A, Inputs_A, Outputs_A, update_A, initialState_A)
(States_B, Inputs_B, Outputs_B, update_B, initialState_B)

Assumptions about the component machines:
$Outputs_A \subset Inputs_B$

Definition of the cascade composite machine:
$States = States_A \times States_B$
$Inputs = Inputs_A$
$Outputs = Outputs_B$
$initialState = (initialState_A, initialState_B)$
$((s_A(n+1), s_B(n+1)), y_B(n)) = update((s_A(n), s_B(n)), x(n)),$

where

$(s_A(n+1), y_A(n)) = update_A(s_A(n), x(n))$ and
$(s_B(n+1), y_B(n)) = update_B(s_B(n), y_A(n)).$

Figure 4.3: Summary of cascade composition of state machines.

4.3 Cascade composition

We now interconnect two state machines as shown in figure 4.3, where the output of one is the input of the other. This is called a **cascade composition** or a **series connection**. We define the composition so that the component machines react together, synchronously, as one state machine.

Suppose the two state machines are given by

$$StateMachine_A = (States_A, Inputs_A, Outputs_A, update_A, initialState_A)$$

and

$$StateMachine_B = (States_B, Inputs_B, Outputs_B, update_B, initialState_B).$$

Let the composition be given by

$$StateMachine = (States, Inputs, Outputs, update, initialState).$$

Clearly, for a composition like that in figure 4.3 to be possible we must have

$$Outputs_A \subset Inputs_B.$$

Then any output sequence produced by machine A can be an input sequence for machine B. As a result,

$$OutputSignals_A \subset InputSignals_B.$$

This is analogous to a **type constraint** in programming languages, where in order for two pieces of code to interact, they must use compatible data types. We encountered a similar constraint in discussing function composition, section 2.1.5.

We are ready to construct a state machine model for this series connection. As noted in the figure, the input alphabet of the composite is

$$Inputs = Inputs_A.$$

The stuttering element of *Inputs*, of course, is just the stuttering element of $Inputs_A$. The output alphabet of the composite is

$$Outputs = Outputs_B.$$

The state space of the composite state machine is the product set

$$States = States_A \times States_B. \tag{4.3}$$

This asserts that the composite state machine is in state $(s_A(n), s_B(n))$ when $StateMachine_A$ is in state $s_A(n)$ and $StateMachine_B$ is in state $s_B(n)$. The initial state is

$$initialState = (initialState_A, initialState_B).$$

We could equally well have defined the states of the composite state machine in the opposite order,

$$States = States_B \times States_A.$$

This would result in a different but bisimilar state machine description (either one simulates the other). Intuitively, it does not matter which of these two choices we make, and we choose (4.3).

To complete the description of the composite machine, we need to define its *update* function in terms of the component machines. Here, a slight subtlety arises. Since we are using synchronous composition, the output symbol of machine A is *simultaneous* with its input symbol. Thus, in a reaction, the output symbol of machine A in that reaction must be available to machine B in the same reaction. This seems intuitive, but it has some counterintuitive consequences. Although the reactions of machine A and B are simultaneous, we must determine the reaction of A before we can determine the reaction of B. This apparent paradox is an intrinsic feature of synchronous composition. We will have to deal with it carefully in feedback composition, where it is not immediately evident which reactions need to be determined first.

In the cascade composition, it is intuitively clear what we need to do to define the update function. We first determine the reaction of machine A. Suppose that at the n-th reaction the input symbol is $x(n)$ and the state is $s(n) = (s_A(n), s_B(n))$, where $s_A(n)$ is the state of machine A and $s_B(n)$ is the state of machine B. Machine A reacts by updating its state to $s_A(n+1)$ and producing output symbol $y_A(n)$,

$$(s_A(n+1), y_A(n)) = update_A(s_A(n), x(n)). \tag{4.4}$$

Its output symbol $y_A(n)$ becomes the input symbol to machine B. Machine B reacts by updating its state to $s_B(n+1)$ and producing output symbol $y_B(n)$,

$$(s_B(n+1), y_B(n)) = update_B(s_B(n), y_A(n)). \tag{4.5}$$

The output of the composite machine, of course, is just the output of machine B, and the next state of the composite machine is just $(s_A(n+1), s_B(n+1))$, so the composite machine's *update* is

$$((s_A(n+1), s_B(n+1)), y_B(n)) = update((s_A(n), s_B(n)), x(n)),$$

where $s_A(n+1)$, $s_B(n+1)$, and $y_B(n)$ are given by (4.4) and (4.5). The definition of the composite machine is summarized in figure 4.3.

Example 4.2: The cascade composition in figure 4.4 has the composite machine with state space

$$States = States_A \times States_B = \{(0,0), (0,1), (1,0), (1,1))\}$$

and alphabets

$$Inputs = Outputs = \{0, 1, absent\}.$$

The initial state is

$$initialState = (0,0).$$

The *update* function is given by the table:

current	(*next state*, *output*) for input		
state	0	1	*absent*
(0,0)	((0,0),0)	((1,1),1)	((0,0), *absent*)
(0,1)	((0,0),1)	((1,1),0)	((0,1), *absent*)
(1,0)	((1,1),1)	((0,0),0)	((1,0), *absent*)
(1,1)	((1,1),0)	((0,0),1)	((1,1), *absent*)

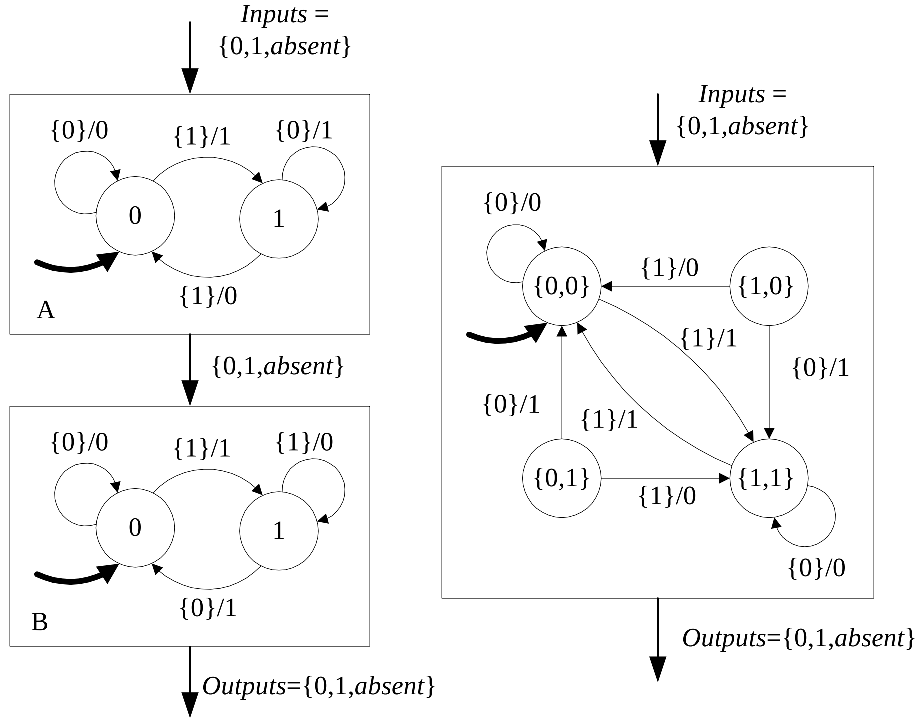


Figure 4.4: Example of a cascade composition. The composed state machine is on the right.

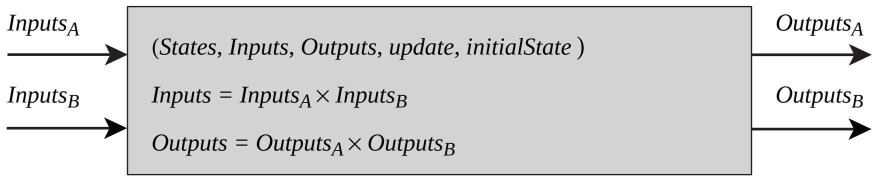

$Inputs_A$
$Inputs_B$
$Outputs_A$
$Outputs_B$
(States, Inputs, Outputs, update, initialState)
$Inputs = Inputs_A \times Inputs_B$
$Outputs = Outputs_A \times Outputs_B$

Figure 4.5: State machine with product-form inputs and outputs.

The update function is also presented in the state transition diagram of figure 4.4. The self-loops corresponding to the stuttering input symbol *absent* are not shown in the diagram.

Observe from the table or the diagram that states $(0,1)$ and $(1,0)$ are not reachable from the initial state. A state s is said to be **reachable** if some sequence of input symbols can take the state machine from the initial state to s. This suggests that a simpler machine with fewer states would exhibit the same input/output behaviors. In fact, notice from the table that the input is always equal to the output! A trivial one-state machine can exhibit the same input/output behaviors. (Exercise 8 gives a procedure for calculating the reachable states of an arbitrary state machine.)

The simple behavior of the composite machine is not immediately apparent from figure 4.4. We have to systematically construct the composite machine to derive this simple behavior. In fact, this composite machine can be viewed as an encoder and decoder, because the input bit sequence is encoded by a distinctly different bit sequence (the intermediate signal in figure 4.4), and then the second machine, given the intermediate signal, reconstructs the original.

This particular encoder is known as a **differential precoder**. It is “differential” in that when the input symbol is 0, the intermediate signal sample is unchanged from the previous sample (whether it was 0 or 1), and when the input symbol is 1, the sample is changed. Thus, the input symbol indicates *change* in the input with a 1, and *no change* with a 0. Differential precoders are used when it is important that the average number of 1’s and 0’s is the same, regardless of the input sequence that is encoded.

4.4 Product-form inputs and outputs

In the state machine model of (3.1), at each step the environment selects one input symbol to which the machine reacts and produces one output symbol. Sometimes we wish to model the fact that some input values are selected by one part of the environment, while other input values are simultaneously selected by another part. Also, some output values are sent to one part of the environment, while other output values are simultaneously sent to another part. The product-form composition permits these models.

The machine in figure 4.5 is shown as a block with two distinct input and output arrows. The figure suggests that the machine receives inputs from two sources and sends outputs to two destinations.

In the answering machine example of chapter 3, for instance, the *end greeting* input value might originate in a physically different piece of hardware in the machine than the *offhook* value.

The distinct arrows into and out of a block are called *ports*. Each port has a set of values called the **port alphabet** associated with it, as shown in figure 4.5. Each port alphabet must include a stuttering element. The set *Inputs* of input values to the state machine is the product of the input sets associated with the ports. Of course, the product can be constructed in any order; each ordering results in a distinct (but bisimilar) state machine model.

In figure 4.5 there are two input ports and two output ports. The upper input port can present to the state machine any value in the alphabet $Inputs_A$, which includes *absent*, its stuttering element. The lower port can present any value in the set $Inputs_B$, which also includes *absent*. The input value actually presented to the state machine in a reaction is taken from the set

$$Inputs = Inputs_A \times Inputs_B.$$

The stuttering element for this alphabet is the pair $(absent, absent)$. The output value produced by a reaction is taken from the set

$$Outputs = Outputs_A \times Outputs_B.$$

If the output of the n-th reaction is $(y_A(n), y_B(n))$, then the upper port shows $y_A(n)$ and the lower port shows $y_B(n)$. These can now be separately presented as inputs to downstream state machines. Again, the stuttering element is $(absent, absent)$.

Example 4.3: The answering machine of figure 3.1 has input alphabet

$$Inputs = \{ring, offhook, end\ greeting, end\ message\}.$$

In a typical realization of an answering machine, *ring* and *offhook* come from a subsystem (often an **ASIC**, or **application-specific integrated circuit**) that interfaces to the telephone line. The value *end greeting* comes from another subsystem, such as a magnetic tape machine or digital audio storage device, that plays the answer message. The value *end message* comes from a similar, but distinct, subsystem that records incoming messages. So a more convenient model will show three separate factors for the inputs, as in figure 4.6. That figure also shows the outputs in product form, anticipating that the distinct output values will need to be sent to distinct subsystems.

Several features distinguish the diagram in figure 4.6 from that of figure 3.1. Each state except the *idle* state has acquired a self-loop labeled *stutter*, which is a name for the guard

$$stutter = \{(absent, absent, absent)\}.$$

This self loop prevents the state machine from returning to the idle state (via the *else* transition) when nothing interesting is happening on the inputs. Usually, there will not be a reaction if nothing interesting is happening on the inputs, but because of synchrony, this machine may be composed with others, and all machines have to react at the same time. So if anything interesting is happening elsewhere in the system, then this machine

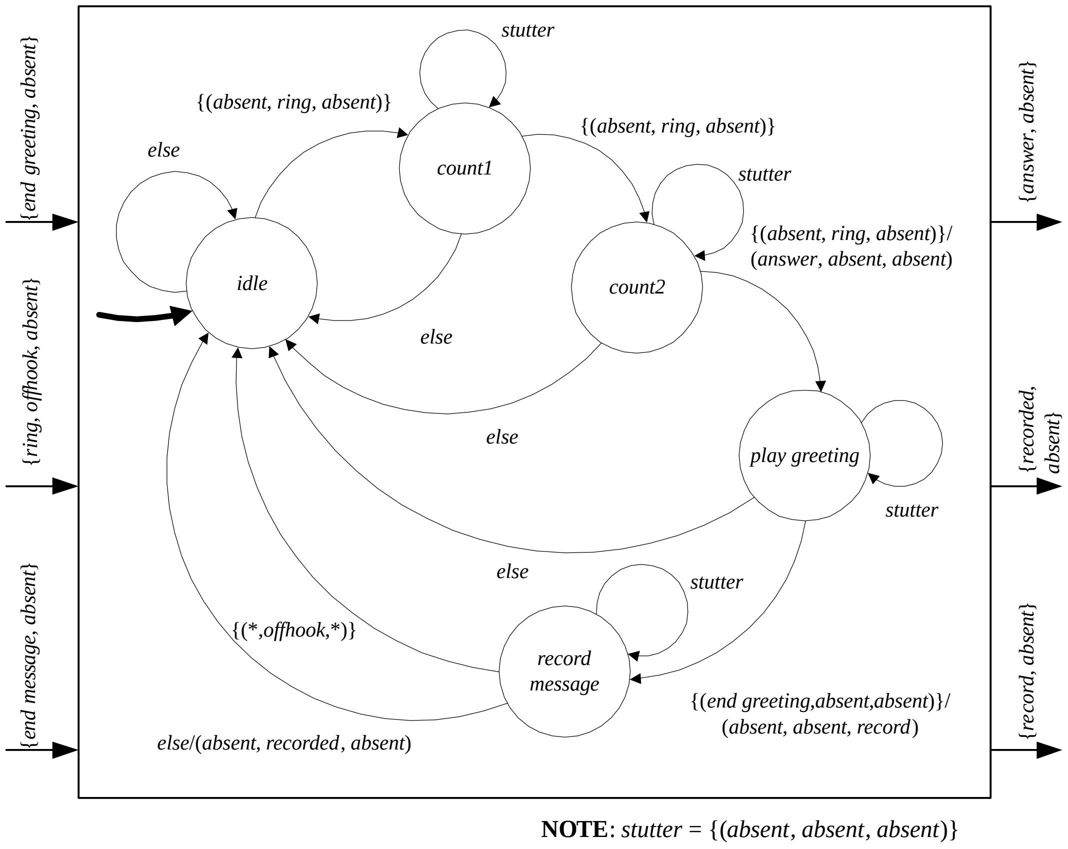


Figure 4.6: Answering machine with product-form inputs and outputs has three input ports and three output ports.

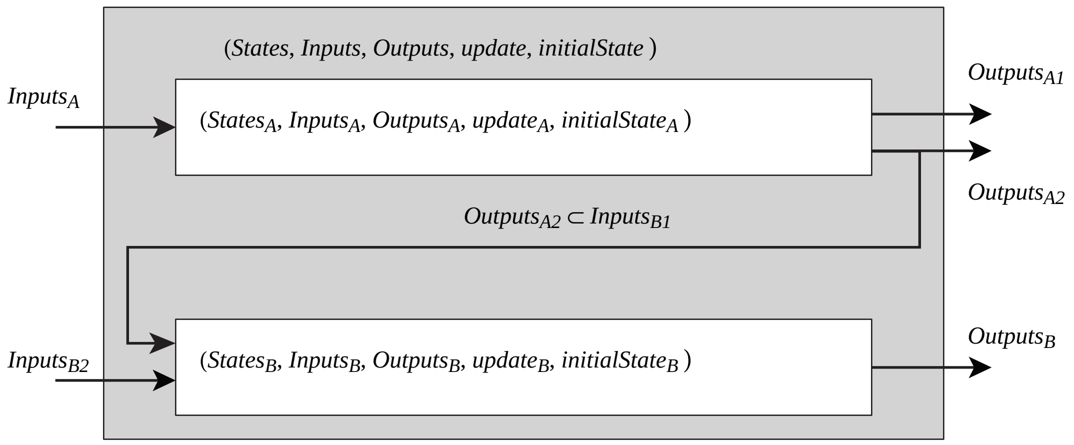

(States, Inputs, Outputs, update, initialState)
Inputs_A
(States_A, Inputs_A, Outputs_A, update_A, initialState_A)
Outputs_A1
Outputs_A2
Outputs_A2 ⊂ Inputs_B1
Inputs_B2
(States_B, Inputs_B, Outputs_B, update_B, initialState_B)
Outputs_B

Figure 4.7: More complex composition.

has to react even though nothing interesting is happening here. Recall that such a reaction is called a stutter. The state does not change, and the output symbol produced is the stuttering element of the output alphabet.

Each guard now consists of a set of triples, since the product-form input has three components. The shorthand "(*, *offhook*, *)" on the arc from the *record message* state to the *idle* state represents a set,

$$(*, \mathit{offhook}, *) = \{(\mathit{absent}, \mathit{offhook}, \mathit{absent}), (\mathit{end\ greeting}, \mathit{offhook}, \mathit{absent}),$$
$$(\mathit{absent}, \mathit{offhook}, \mathit{end\ message}), (\mathit{end\ greeting}, \mathit{offhook}, \mathit{end\ message})\}.$$

The "*" is a **don't care** or **wildcard** notation. Anything in its position will trigger the guard.

Because there are three output ports, the output symbols are also triples, but most of them are implicitly $(\mathit{absent}, \mathit{absent}, \mathit{absent})$.

4.5 General feedforward composition

Given that state machines can have product-form inputs and outputs, it is easy to construct a composition of state machines that combines features of both the cascade composition of figure 4.3 and the side-by-side composition of figure 4.1. An example is shown in figure 4.7. In that figure,

$$\begin{aligned} \mathit{Outputs}_A &= \mathit{Outputs}_{A1} \times \mathit{Outputs}_{A2} \\ \mathit{Inputs}_B &= \mathit{Inputs}_{B1} \times \mathit{Inputs}_{B2}. \end{aligned}$$

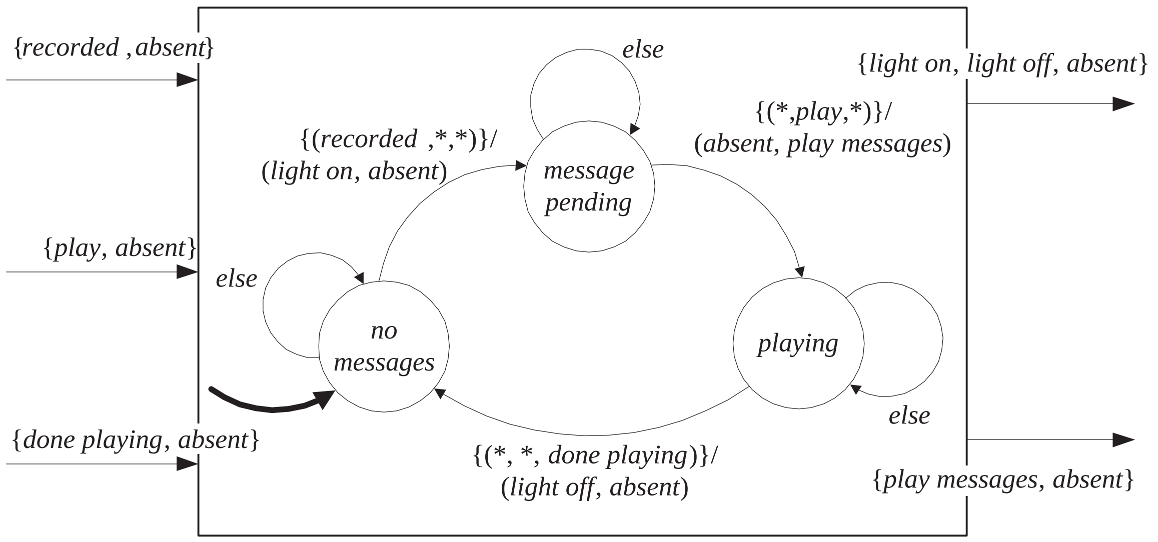


Figure 4.8: Playback system for composing with the answering machine.

Notice that the bottom port of machine *A* goes both to the output of the composite machine and to the top port of machine *B*. Sending a value to multiple destinations like this is called **forking**. In exercise 1 at the end of this chapter you are asked to define the composite machine for this example.

Example 4.4:

We compose the answering machine of figure 4.6 with a playback system, shown in figure 4.8, which plays messages that have been recorded by the answering machine. The playback system receives the *recorded* input symbol from the answering machine whenever the answering machine is done recording a message. Its task is to light an indicator that a message is pending, and to wait for a user to press a play button on the answering machine to request that pending messages be played back. When that button is pressed, all pending messages are played back. When they are done being played back, then the indicator light is turned off.

The composition is shown in figure 4.9. The figure shows a number of other components, not modeled as state machines, to help understand how everything works in practice. These other components are shown as three-dimensional objects, to emphasize their physicality. We have simplified the figure by omitting the *absent* elements of all the sets. They are implicit.

A telephone line interface provides *ring* and *offhook* when these are detected. Detection of one of these can trigger a reaction of the composite machine. In fact, any output symbol from any of the physical components can trigger a reaction of the composite machine. When *AnsweringMachine* generates the *answer* output symbol, then the “greeting playback device” plays back the greeting. From the perspective of the state machine model, all that happens is that time passes (during which some reactions may occur), and then an *end greeting* input symbol is received. The recording device

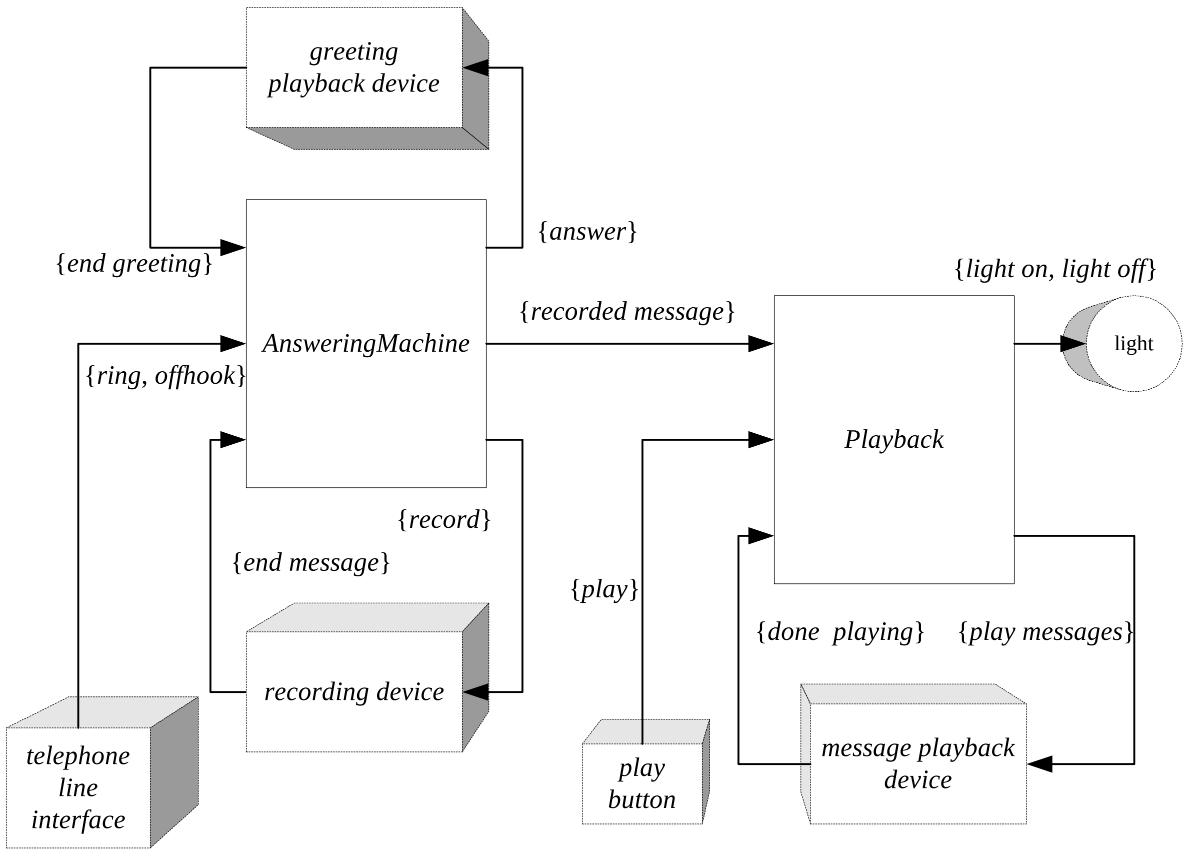


Figure 4.9: Composition of an answering machine with a message playback machine. The three-dimensional boxes are physical components that are not modeled as state machines. They are the sources of some inputs and the destinations of some outputs.

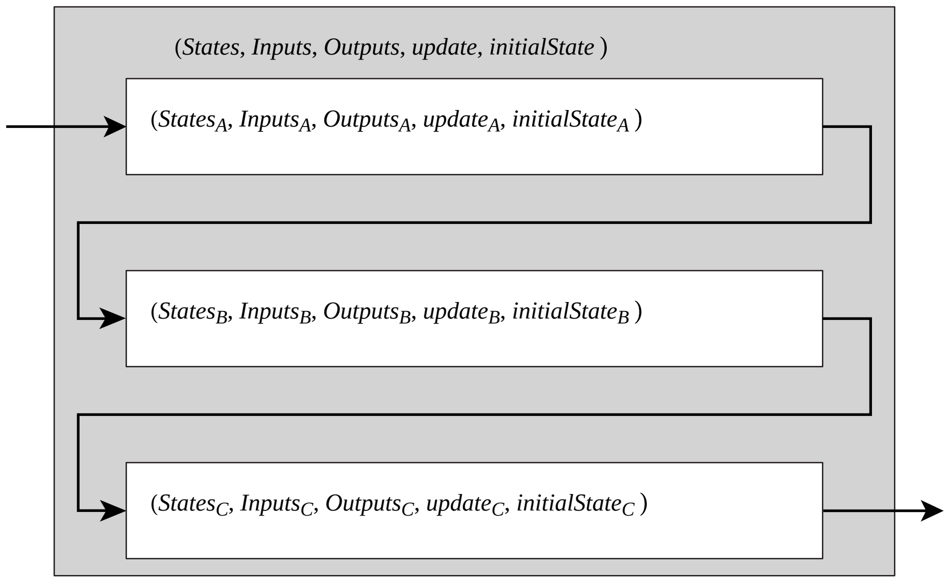

(States, Inputs, Outputs, update, initialState)
($States_A$, $Inputs_A$, $Outputs_A$, $update_A$, $initialState_A$)
($States_B$, $Inputs_B$, $Outputs_B$, $update_B$, $initialState_B$)
($States_C$, $Inputs_C$, $Outputs_C$, $update_C$, $initialState_C$)

Figure 4.10: Cascade composition of three state machines. They can be composed in different ways into different, but bisimilar, state machines.

works similarly. When *AnsweringMachine* generates a *recorded* output symbol, then the *Playback* machine will respond by lighting the indicator light. When a user presses the play button the input symbol *play* is generated, the composite machine reacts, and the *Playback* machine issues a *play messages* output symbol to the "message playback device." This device also allows time to pass, then generates a *done playing* input symbol to the composite state machine.

If we wish to model the playback or recording subsystem in more detail using finite state machines, then we need to be able to handle feedback compositions. These are considered below.

4.6 Hierarchical composition

By using the compositions discussed above, we can now handle any interconnection of state machines that does not have feedback. Consider for example the cascade of three state machines shown in figure 4.10. The composition techniques we have discussed so far involved only two state machines. It is easy to generalize the composition in figure 4.3 to handle three state machines (see exercise 2), but a more systematic method might be to apply the composition of figure 4.3 to compose two of the state machines, and then apply it again to compose the third state machine with the result of the first composition. This is called **hierarchical composition**.

In general, given a collection of interconnected state machines, there are several ways to hierarchically compose them. For example, in figure 4.10, we could first compose machines *A* and *B* to get,

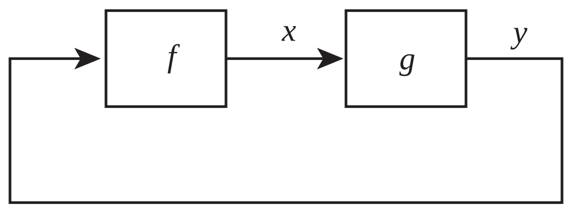


Figure 4.11: Illustration of a fixed point problem.

say, machine D, and then compose D with C. Alternatively, we could first compose B and C to get, say, machine E, and then compose E and A. These two procedures result in different but bisimilar state machine models (each simulates the other).

4.7 Feedback

In simple feedback systems, an output from a state machine is fed back as an input to the same state machine. In more complicated feedback systems, several state machines might be connected in a loop; the output of one eventually affects its own input through some intervening state machines.

Feedback is a subtle form of composition in the synchronous model. In synchronous composition, in a reaction, the output symbol of a state machine is simultaneous with the input symbol. So the output symbol of a machine in feedback composition depends on an input symbol that depends on its own output symbol!

We frequently encounter such situations in mathematics. A common problem is to find x such that

$$x = f(x) \tag{4.6}$$

for a given function f. A solution to this equation, if it exists, is called a **fixed point** in mathematics. It is analogous to feedback because the 'output' $f(x)$ of f is equal to its 'input' x, and vice versa. The top diagram in figure 4.12 illustrates a similar relationship: the state machine's output symbol is the same as its (simultaneous) input symbol.

A more complicated problem, involving two equations, is to find x and y so that

$$x = f(y), \text{ and } y = g(x).$$

The analogous feedback composition has two state machines in feedback, with the structure of figure 4.11.[2]

A fixed-point equation like (4.6) may have no fixed point, a unique fixed point, or multiple fixed points. Take for example the function $f\colon Reals \to Reals$ where $\forall x \in Reals,\ \ f(x) = 1 + x^2$. In this case, (4.6) becomes $x = 1 + x^2$, which has no fixed point in the reals. If $f(x) = 1 - x$, (4.6) becomes

[2]Figure 4.9 would be a feedback composition if any of the three recording or playback devices were modeled as state machines. In the figure, however, these devices are part of the environment.

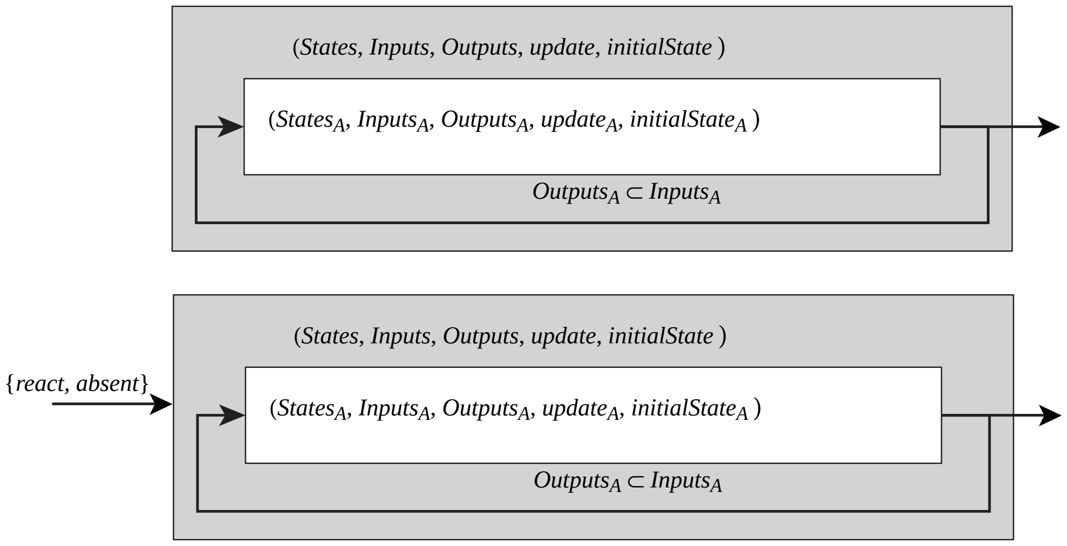

(States, Inputs, Outputs, update, initialState)
(States_A, Inputs_A, Outputs_A, update_A, initialState_A)
Outputs_A ⊂ Inputs_A
{react, absent}
(States, Inputs, Outputs, update, initialState)
(States_A, Inputs_A, Outputs_A, update_A, initialState_A)
Outputs_A ⊂ Inputs_A

Figure 4.12: Feedback composition with no inputs.

$x = 1 - x$, which has a unique fixed point, $x = 0.5$. Lastly, if $f(x) = x^2$, (4.6) becomes $x = x^2$, which has two fixed points, $x = 0$ and $x = 1$.

In the context of state machines, a feedback composition with no fixed point in some reachable state is a defective design; we call such a composition **ill-formed**. We can not evaluate an ill-formed composition. Usually, we also wish to exclude feedback compositions that have more than one non-stuttering fixed point in some reachable state. So these too are ill-formed. A feedback composition with a unique non-stuttering fixed point in all reachable states is **well-formed**. Fortunately, it is easy to construct well-formed feedback compositions, and they prove surprisingly useful. We explore this further, beginning with a somewhat artificial case of feedback composition with no inputs.

4.7.1 Feedback composition with no inputs

The upper state machine in figure 4.12 has an output port that feeds back to its input port. We wish to construct a state machine model that hides the feedback, as suggested by the figure. The result will be a state machine with no input. This does not fit our model, which requires the environment to provide inputs to which the machine reacts. So we artificially provide an input alphabet

$$Inputs = \{react, absent\},$$

as suggested in the lower machine in figure 4.12. We interpret the input symbol *react* as a command for the internal machine to react, and the input symbol *absent* as a command for the internal machine to stutter. The output alphabet is

$$Outputs = Outputs_A.$$

This is an odd example of a synchronous/reactive system because of the need for this artificial input alphabet. Typically, however, such a system will be composed with others, as suggested in

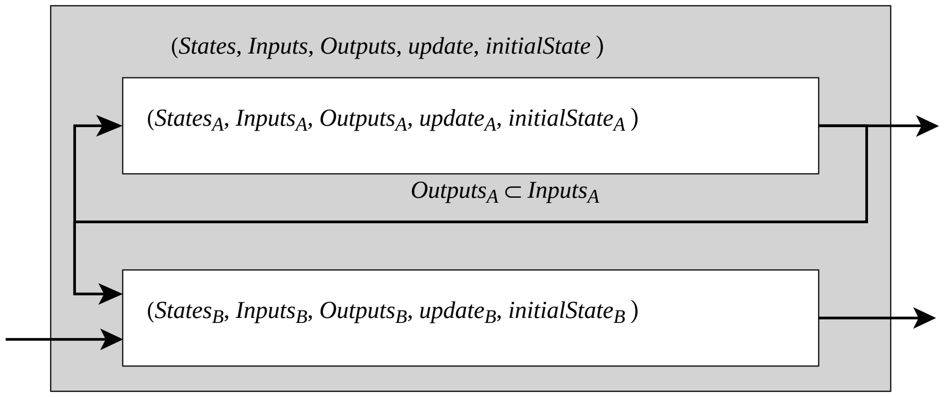

(States, Inputs, Outputs, update, initialState)
(StatesA, InputsA, OutputsA, updateA, initialStateA)
OutputsA ⊂ InputsA
(StatesB, InputsB, OutputsB, updateB, initialStateB)

Figure 4.13: Feedback composition composed with another state machine.

figure 4.13. That composition does have an external input. So the overall composition, including the component with no external input, reacts whenever an external input symbol is presented, and there is no need for the artificial inputs. Of course when a stuttering element is provided to the composite, all components stutter.

Although it is not typical, we first consider the example in figure 4.12 because the formulation of the composition is simplest. We will augment the model to allow inputs after this.

In figure 4.12, for the feedback connection to be possible, of course, we must have

$$Outputs_A \subset Inputs_A.$$

Suppose the current state at the n-th reaction is $s(n) \in States_A$. The problem is to find the output symbol $y(n) \in Outputs_A$. Since $y(n)$ is also the input symbol, it must satisfy

$$(s(n+1), y(n)) = update_A(s(n), y(n)),$$

where $s(n+1)$ is the next state. The difficulty here is that the "unknown" $y(n)$ appears on both sides. Once we find $y(n)$, $s(n+1)$ is immediately determined by the $update_A$ function. To simplify the discussion, we get rid of $s(n+1)$ by working with the function

$$output_A : States_A \times Inputs_A \to Outputs_A$$

This function gives the output symbol as a function of the current state and the current input symbol, as we saw in section 3.1.1. So our problem is: given the current state $s(n)$ and the known function $output_A$, find $y(n)$ such that

$$y(n) = output_A(s(n), y(n)). \tag{4.7}$$

Here $s(n)$ is a known constant, so the equation is of the form (4.6), and its solution, if it exists, is a fixed point.

One solution that is always available is to stutter, i.e.,

$$y(n) = absent, \text{ (and then } s(n+1) = s(n)),$$

since $absent = output_A(s(n), absent)$, assuming that *absent* is the stuttering input symbol for machine A. But this is not an interesting solution, since the state does not change. We want to find a non-stuttering solution for $y(n)$.

We say that the composition of figure 4.12 is **well-formed** if for every reachable $s(n) \in States_A$, there is a unique non-stuttering output symbol $y(n)$ that solves (4.7); otherwise, the composition is **ill-formed**. If the composition is well-formed, the composite machine definition is:

$$\begin{aligned}
&States = States_A \\
&Inputs = \{react, absent\} \\
&Outputs = Outputs_A \\
&initialState = initialState_A \\
&update(s(n), x(n)) = \begin{cases} update_A(s(n), y(n)), \\ \text{where } y(n) \neq absent \text{ uniquely satisfies (4.7)} & \text{if } x(n) = react \\ \\ (s(n), x(n)) & \text{if } x(n) = absent \end{cases}
\end{aligned}$$

Notice that the composite machine is defined only if the composition is well-formed, i.e., there is a unique $y(n)$ that satisfies (4.7). If there is no such $y(n)$, the composition is ill-formed and the composite machine is not defined.

The next example illustrates the difference between well-formed and ill-formed compositions. It will suggest a procedure to solve (4.7) in the important special case of systems with state-determined output.

Example 4.5: Consider the three feedback compositions in figure 4.14. In all cases, the input and output alphabets of the component machines are

$$Inputs_A = Outputs_A = \{true, false, absent\}.$$

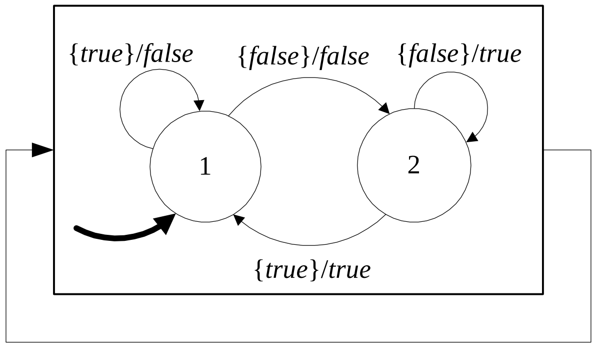
The input alphabet to the *composite* machine is $\{react, absent\}$, as in figure 4.12, but we do not show this (to reduce clutter in the figure). We want to find a non-stuttering solution $y(n)$ of (4.7). Since the output symbol is also the input symbol, we are looking for a non-stuttering input symbol.

Consider the top machine first. Suppose the current state is the initial state, $s(n) = 1$. There are two outgoing arcs, and for a non-stuttering input symbol, both produce $y(n) = false$, so we can conclude that the output symbol of the machine is *false*. Since the output symbol is *false*, then the input symbol is also *false*, and the non-stuttering fixed point of (4.7) is unique,

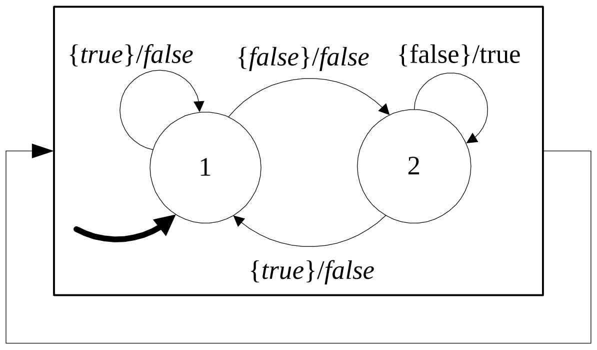
$$output_A(1, false) = false.$$

The state transition taken by the reaction goes from state 1 to state 2.

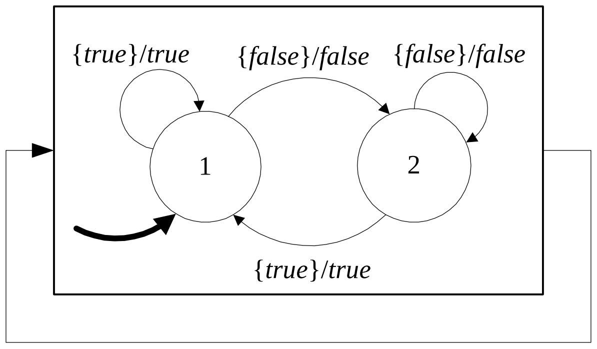
Suppose next that the current state is $s(n) = 2$. Again, there are two outgoing arcs. Both arcs produce output symbol *true* for a non-stuttering input symbol, so we can conclude



(a)



(b)



(c)

Figure 4.14: Three examples of feedback composition. Examples (b) and (c) are ill-formed. Composition (b) has no non-stuttering fixed point in state 2, while composition (c) has two non-stuttering fixed points in either state.

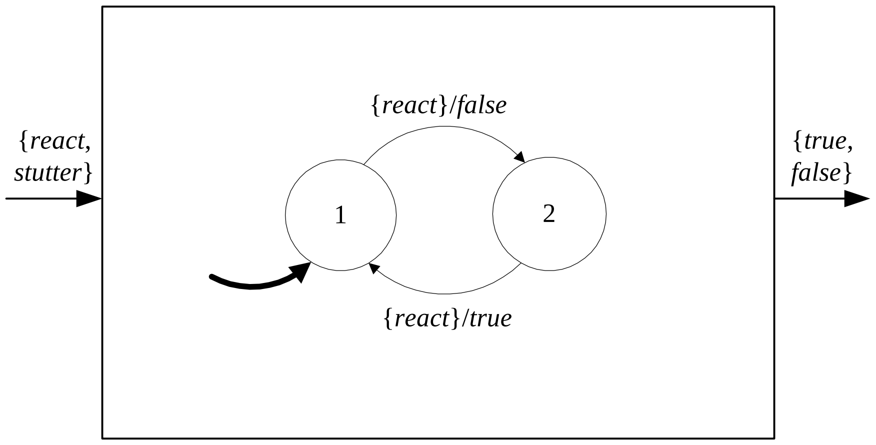


Figure 4.15: Composite machine for figure 4.14(a).

that the output symbol is *true*. Since the output symbol is *true*, then the input symbol is also *true*, there is a unique non-stuttering fixed point,

$$output_A(2, true) = true,$$

and the state transition taken goes from 2 to 1. Since there is a unique non-stuttering fixed point in every reachable state, the feedback composition is well-formed.

The composite machine alternates states on each reaction, and produces the output sequence

$$(false, true, false, true, false, true, \cdots)$$

for the input sequence

$$(react, react, react, \cdots).$$

The composite machine is shown in figure 4.15.

Now consider the second machine in figure 4.14. If the initial state is 1 the analysis is the same as above. There is a unique non-stuttering fixed point, the output and input symbols are both *false*, and the state transition goes from 1 to 2. But if the initial state is 2 and the unknown input symbol is *true*, the output symbol is *false*; and if the unknown input symbol is *false*, the output symbol is *true*. Thus there is no non-stuttering fixed point $y(n)$ that solves,

$$output_A(2, y(n)) = y(n).$$

The feedback composition is not well-formed.

Consider the third machine in figure 4.14. This feedback composition is also ill-formed but for a different reason. If the initial state is 1 and the unknown input symbol is *true*, the output symbol is also *true*, so *true* is a fixed point, and the output symbol can be *true*. However, the output symbol can also be *false*, since if it is, then a transition will be taken that produces the input symbol *false*. So *false* is also a fixed point. Thus, the problem here is that there is more than one non-stuttering solution, not that there are none!

Our conclusion is that with machines like the second and third, you cannot connect them in a feedback composition as shown. The second is rejected because it has no solution and the third because it has more than one. We only accept feedback compositions where there is exactly one non-stuttering solution in each reachable state.

4.7.2 State-determined output

In the first machine of figure 4.14, in each state, all outgoing arcs produce the *same* output symbol, independent of the input symbol. In other words, the output symbol $y(n)$ depends only on the state; in the example, $y(n) = \mathit{false}$ if $s(n) = 1$, and $y(n) = \mathit{true}$ if $s(n) = 2$. The unique fixed point of (4.7) is this output symbol, and we can immediately conclude that the feedback composition is well-formed.

We say that a machine A has **state-determined output** if in every reachable state $s(n) \in \mathit{States}_A$, there is a unique output symbol $y(n) = b$ (which depends on $s(n)$) independent of the non-stuttering input symbol; i.e. for every $x(n) \neq \mathit{absent}$,

$$\mathit{output}_A(s(n), x(n)) = b.$$

In this special case of state-determined output, the composite machine is:

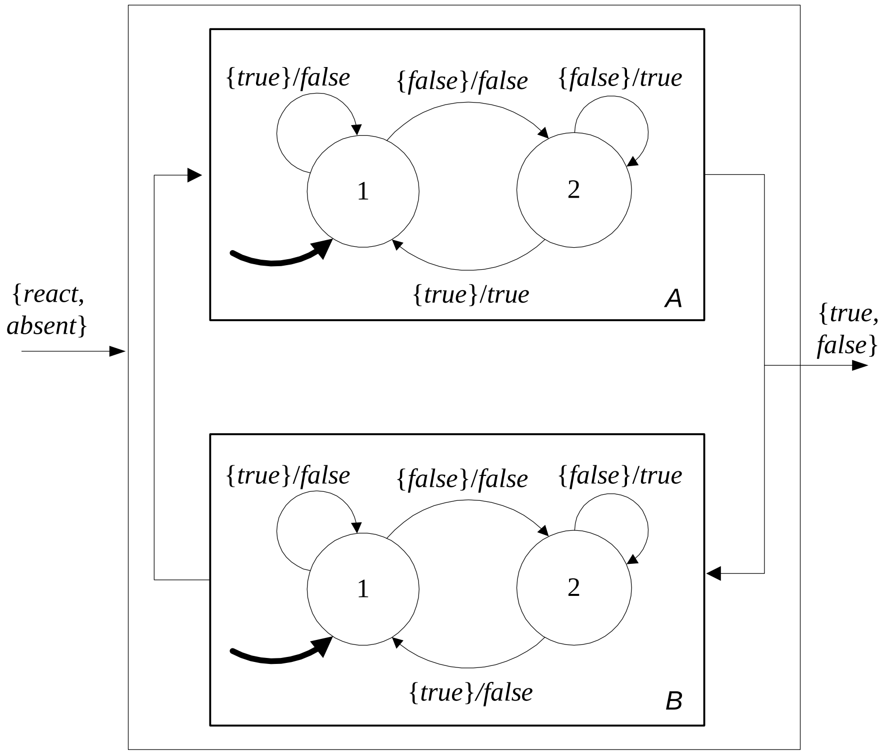
$$\begin{aligned}
&\mathit{States} = \mathit{States}_A\\
&\mathit{Inputs} = \{\mathit{react}, \mathit{absent}\}\\
&\mathit{Outputs} = \mathit{Outputs}_A\\
&\mathit{initialState} = \mathit{initialState}_A\\
&\mathit{update}(s(n), x(n)) = \begin{cases} \mathit{update}_A(s(n), b), & \\ \text{where } b \text{ is the unique output symbol in state } s(n) & \text{if } x(n) = \mathit{react} \\ & \\ (s(n), x(n)) & \text{if } x(n) = \mathit{absent} \end{cases}
\end{aligned}$$

When a machine with state-determined output is combined with any other state machines in a feedback composition, the resulting composition is also well-formed, as illustrated in the next example.

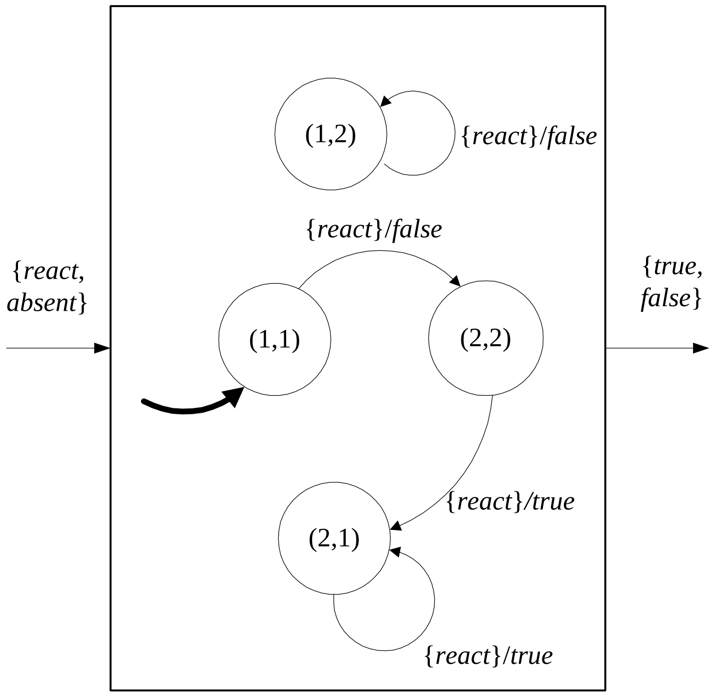
Example 4.6:

In figure 4.16 the machine A is combined with machine B in a feedback composition. A is the same as the first machine, and B is the same as the second machine in figure 4.14. (The output port of B is drawn on the left and the input port on the right so that the block diagram looks neater.) A has state-determined output, but B does not. The composition is well-formed.

To see this, suppose both machines are in their initial states 1. A produces output symbol *false*, independent of its input symbol. This output is the input of B which then



(a)



(b)

Figure 4.16: Machine A has state-determined output, but B does not. The feedback composition is well-formed, and the composite machine is shown on the bottom. Note that state $(1,2)$ is not reachable.

produces output symbol *false* and makes the transition to state 2. The output symbol *false* of B is the input to A which makes the transition to its state 2. (A and B make their state transitions together in our synchronous/reactive model.) We can determine the output symbol and transition in the same way for all other states. The state diagram of the composite machine is shown in the figure on the right. Note that state $(1,2)$ is not reachable from the initial state $(1,1)$, so we could have ignored it in determining whether the composition is well-formed.

The input alphabet of the composite machine is $\{react, absent\}$, taking *absent* as the stuttering input symbol. The output alphabet is the same as the output alphabet of A, $\{true, false, absent\}$. The state space is $States_A \times States_B$. The *update* function is given by the table:

current	(*next state*, *output*) for input	
state	*react*	*absent*
(1,1)	((2,2),*false*)	((1,1),*absent*)
(2,2)	((2,1),*true*)	((2,2), *absent*)
(1,2)	((1,2),*false*)	((1,2), *absent*)
(2,1)	((2,1),*true*)	((2,1), *absent*)

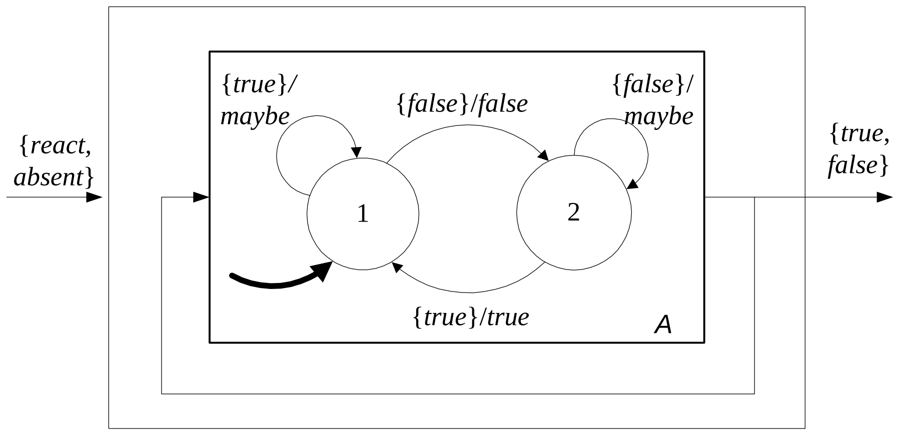
It is possible for a machine without state-determined outputs to be placed in a well-formed feedback composition as illustrated in the next example.

Example 4.7: Consider the example in figure 4.17. For the component machine, the output alphabet is $Outputs_A = \{true, false, maybe, absent\}$, and the input alphabet is $Inputs_A = \{true, false, absent\}$. The stuttering element is *absent*. The machine does not have state-determined output because, for instance, the outgoing arcs from state 1 can produce both *maybe* and *false*. Nevertheless, equation (4.7) has a unique non-stuttering fixed point in each state:

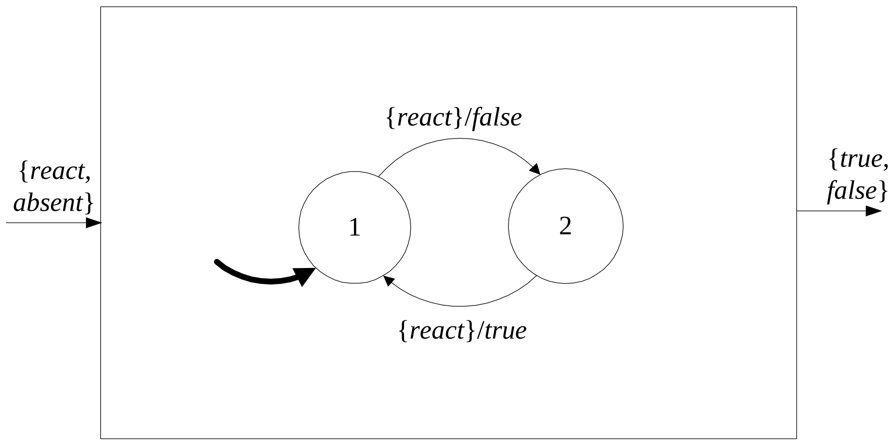
$$output_A(1, false) = false, \text{ and } output_A(2, true) = true.$$

So the feedback composition is well-formed. The composite machine is shown on the bottom.

It can be considerably harder to find the behavior of a feedback composition without state-determined outputs, even if the composition is well-formed. Below, in section 4.7.4, we give a constructive procedure that often works to quickly find a fixed point, and to determine whether it is unique. However, even that procedure does not always work (and in fact, will fail on example 4.7). If the input alphabet is finite, the only strategy that always works is to try all possible output values $y(n)$ is (4.7) for each reachable state $s(n)$. Before discussing this procedure, we generalize to more interesting feedback compositions.



(a)



(b)

Figure 4.17: Machine A does not have state-determined outputs, but the feedback composition is well-formed. The machine on the bottom is the composite machine.

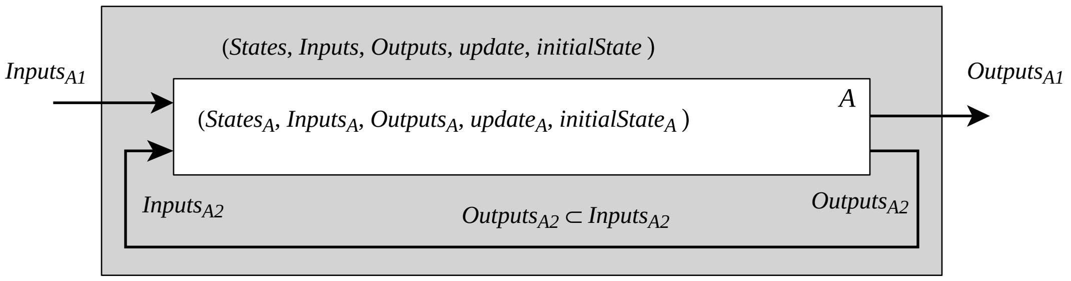

(States, Inputs, Outputs, update, initialState)
Inputs_A1
Outputs_A1
A
(States_A, Inputs_A, Outputs_A, update_A, initialState_A)
Inputs_A2
Outputs_A2 ⊂ Inputs_A2
Outputs_A2

Figure 4.18: Feedback composition of a state machine.

4.7.3 Feedback composition with inputs

Now consider the state machine in figure 4.18. It has two input and output ports. The second output port feeds back to the second input port. We wish to construct a state machine model that hides the feedback, as suggested by the figure, and becomes a simple input/output state machine. This is similar to the example in figure 4.12, but now there is an additional input and an additional output. The procedure for finding the composite machine is similar, but the notation is more cumbersome. Given the current state and the current external input symbol, we must determine the "unknown" output symbol.

The inputs and outputs of machine A are in product form:

$$\begin{aligned} Inputs_A &= Inputs_{A1} \times Inputs_{A2}, \\ Outputs_A &= Outputs_{A1} \times Outputs_{A2}. \end{aligned}$$

For the feedback composition to be possible we must have

$$Outputs_{A2} \subset Inputs_{A1}.$$

The output function of A is

$$output_A : States_A \times Inputs_A \to Outputs_A.$$

It is convenient to write it in product form as,

$$output_A = (output_{A1}, output_{A2}),$$

where

$$output_{A1} : States_A \times Inputs_A \to Outputs_{A1},$$

gives the output symbol at the first output port and

$$output_{A2} : States_A \times Inputs_A \to Outputs_{A2},$$

gives the output symbol at the second output port.

Suppose we are given that at the n-th reaction, the current state of A is $s(n)$ and the current external input symbol is $x_1(n) \in Inputs_{A1}$. Then the problem is to find the "unknown" output symbol $(y_1(n), y_2(n)) \in Outputs_A$ such that

$$output_A(s(n), (x_1(n), y_2(n))) = (y_1(n), y_2(n)). \tag{4.8}$$

The symbol $y_2(n)$ appears on both sides because the second input $x_2(n)$ to machine A is equal to $y_2(n)$. In terms of the product form, (4.8) is equivalent to two equations:

$$output_{A1}(s(n), (x_1(n), y_2(n))) \;=\; y_1(n), \tag{4.9}$$

$$output_{A2}(s(n), (x_1(n), y_2(n))) \;=\; y_2(n). \tag{4.10}$$

In these equations, $s(n)$ and $x_1(n)$ are known, while $y_1(n)$ and $y_2(n)$ are unknown. Observe that if (4.10) has a unique solution $y_2(n)$, then the input symbol to A is $(x_1(n), y_2(n))$ and the next state $s(n+1)$ and output symbol $y_1(n)$ are determined. So the fixed point equation (4.10) plays the same role as (4.7).

We say that the composition of figure 4.18 is well-formed if for every reachable state $s(n) \in States_A$ and for every external input symbol $x_1(n) \in Inputs_{A1}$, there is a unique non-stuttering output symbol $y_2(n) \in Outputs_{A2}$ that solves (4.10). If the composition is well-formed, the composite machine definition is:

$States = States_A$
$Inputs = Inputs_{A1}$
$Outputs = Outputs_{A1}$
$initialState = initialState_A$
$update(s(n), x(n)) = (nextState(s(n), x(n)), output(s(n), x(n)))$:
$nextState(s(n), x(n)) = nextState_A(s(n), (x(n), y_2(n)))$ and
$output(s(n), x(n)) = output_A(s(n), (x(n), y_2(n)))$, where $y_2(n)$ is the unique solution of (4.10).

(The *nextState* function is defined in section 3.1.1.)

In the following example, we illustrate the procedure for defining the composition machine given a sets and functions description (3.1) for the component machine A.

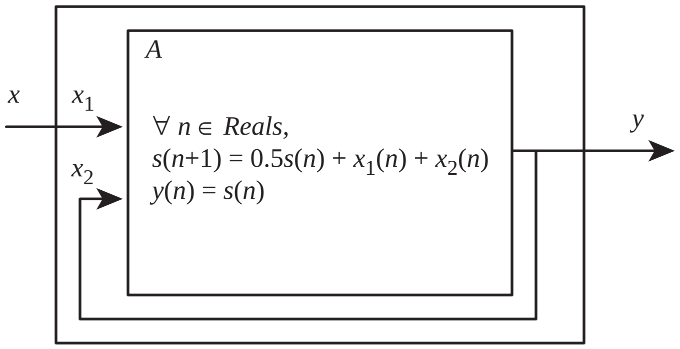
Example 4.8: Figure 4.19 shows a feedback composition, where component machine A has two input ports and one output port,

$$Inputs_A = Reals \times Reals, \;\; Outputs_A = Reals,$$

and states $States_A = Reals$. Thus A has infinite input and output alphabets and infinitely many states. At the n-th reaction, the pair of input values is denoted by $(x_1(n), x_2(n))$, the current state by $s(n)$, the next state by $s(n+1)$, and the output symbol by $y(n)$. In terms of these, the update function is given by

$$(s(n+1), y(n)) = update_A(s(n), (x_1(n), x_2(n))) = (0.5s(n) + x_1(n) + x_2(n), s(n)).$$

Equivalently,



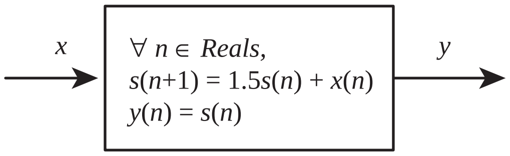


Figure 4.19: Machine A has two input ports and one output port. The output port is connected to the second input port. The composition is well-formed. The composite machine is shown at the bottom.

$$s(n+1) = nextState_A(s(n),(x_1(n),x_2(n))) = 0.5s(n)+x_1(n)+x_2(n)$$
$$y(n) = output_A(s(n),(x_1(n),x_2(n))) = s(n)$$

Thus, the component machine A has state-determined output. The feedback connects the output port to the second input port, so $x_2(n) = y(n)$. Given the current state $s(n)$ and the external input symbol $x(n)$ at the first input port, (4.10) becomes,

$$output_A(s(n),(x_1(n),x_2(n))) = x_2(n),$$

which gives

$$s(n) = x_2(n).$$

So the composite machine is defined by

$$Inputs = Reals, \;\; Outputs = Reals, \;\; States = Reals$$
$$update(s(n),x(n)) = (0.5s(n)+x(n)+s(n),s(n)) = (1.5s(n)+x(n),s(n)).$$

Note that the input to the composite machine is a scalar. The composite machine is shown in the lower part of the figure.

4.7.4 Constructive procedure for feedback composition

Our examples so far involve one or two state machines and a feedback loop. If any machine in the loop has state determined output, then finding the fixed point is easy. Most interesting designs are

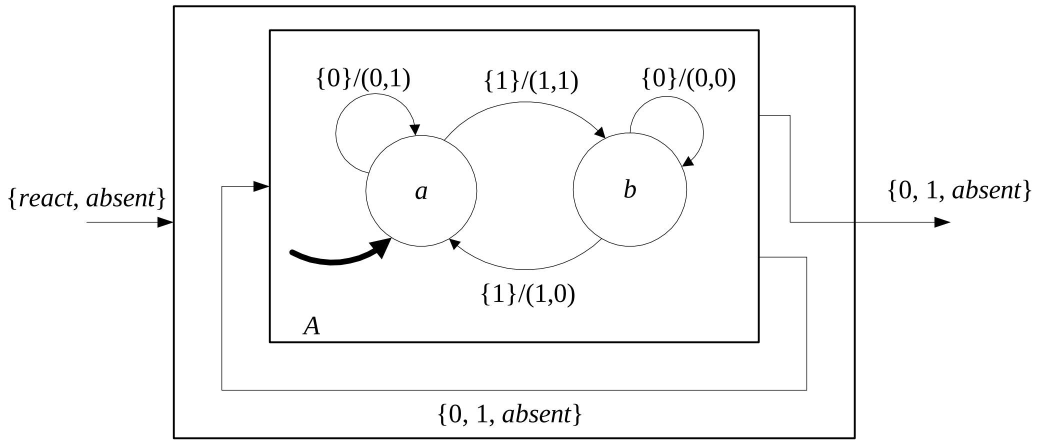


Figure 4.20: Feedback composition without state-determined output.

more complicated, involving several state machines and several feedback loops, and the loops do not necessarily include mahcines with state-determined output.

In this section, we describe a constructive procedure for finding the fixed point that often (but not always) works. It is "constructive" in the sense that it can it be applied mechanically, and will, in a finite number of steps, either identify a fixed point or give up. The approach is simple. At each reaction, begin with all unspecified signals having value *unknown*. Then with what is known about the input symbols, try each state machine to determine as much as possible about the output symbols. You can try the state machines in any order. Given what you learn about the output symbols, then update what you know about the feedback input symbols, and repeat the process, trying each state machine again. Repeat this process until all signal values are specified, or until you learn nothing more about the output symbols. We illustrate the procedure in an example involving only one machine, but keep in mind that the procedure works for any number of machines.

Example 4.9: Figure 4.20 shows a feedback composition without state-determined output. Nonetheless, our constructive procedure can be used to find a unique fixed point for each reaction. Suppose that the current state is *a*, and that the input to the composition is *react*. Begin by assuming that the symbol on the feedback connection is *unknown*. Try component machine *A* (this is the only component machine in this example, but if there were more, we could try them in any order). Examining machine *A*, we see that in its current state, *a*, the output symbol cannot be fully determined. Thus, this machine does not have state-determined output. However, more careful examination reveals that in state *a*, the second element of the output tuple *is* determined. That second element has value 1. Fortunately, this changes the value on the feedback connection from *unknown* to 1.

Now we repeat the procedure. We choose a state machine to try. Again, there is only one state machine in this example, so we try *A*. This time, we know that the input symbol is 1, so we know that the machine must take the transition from *a* to *b* and produce the output tuple $(1,1)$. This results in all symbols being known for the reaction, so we are done evaluating the reaction.

Now assume the current state is b. Again, the feedback symbol is initially *unknown*, but once again, trying A, we see that the second element of the output tuple must be 0. Thus, we change the feedback symbol from *unknown* to 0 and try the machine again. This time, its input is 0, so it must take the self loop back to b and produce the output tuple $(0,0)$.

Recall that the set *Behaviors* is the set of all (x,y) such that x is an input sequence and y is an output sequence. For this machine, ignoring stuttering, the only possible input sequence is $(react, react, react, \cdots)$. We have just determined that the resulting output sequence is $(1,0,0,0,\cdots)$. Thus, ignoring stuttering,

$$Behaviors = \{((react, react, react, \cdots), (1,0,0,0,\cdots))\}.$$

Of course, we should take into account stuttering, so this set needs to be augmented with all (x,y) pairs that look like the one above but have stuttering symbols inserted.

This procedure can be applied in general to any composition of state machines. If the procedure can be applied successfully (nothing remains *unknown*) for all reachable states of the composition, then the composition is well-formed. The following example applies the procedure to a more complicated example.

Example 4.10: We add more detail to the message recorder in figure 4.9. In particular, as shown in figure 4.21, we wish to model the fact that the message recorder stops recording when either it detects a dialtone or when a timeout period is reached. This is modeled by a two-state finite state machine, shown in figure 4.22. Note that this machine does not have state-determined output. For example, in state *idle*, the output could be $(absent, start\ recording)$ or it could be $(absent, absent)$ when the input is not the stuttering input.

The *MessageRecorder* and *AnsweringMachine* state machines form a feedback loop. Let us verify that composition is well-formed. First, note that in the *idle* state of the *MessageRecorder*, the upper output symbol is known to be *absent* (see figure 4.22). Thus, only in the *recording* state is there any possibility of a problem that would lead to the composition being ill-formed. In that state, the output symbol is not known unless the input symbols are known. However, notice that the *recording* state is entered only when a *record* input symbol is received. In figure 4.6, you can see that the *record* value is generated only when entering state *record message*. But in all arcs emerging from that state, the lower output symbol of *AnsweringMachine* will always be absent; the input symbol does not need to be known to know that. Continuing this reasoning by considering all possible state transitions from this point, we can convince ourselves that the feedback loop is well-formed.

The sort of reasoning in this more complicated example is difficult and error-prone for even moderate compositions of state machines. It is best automated. Compilers for synchronous languages

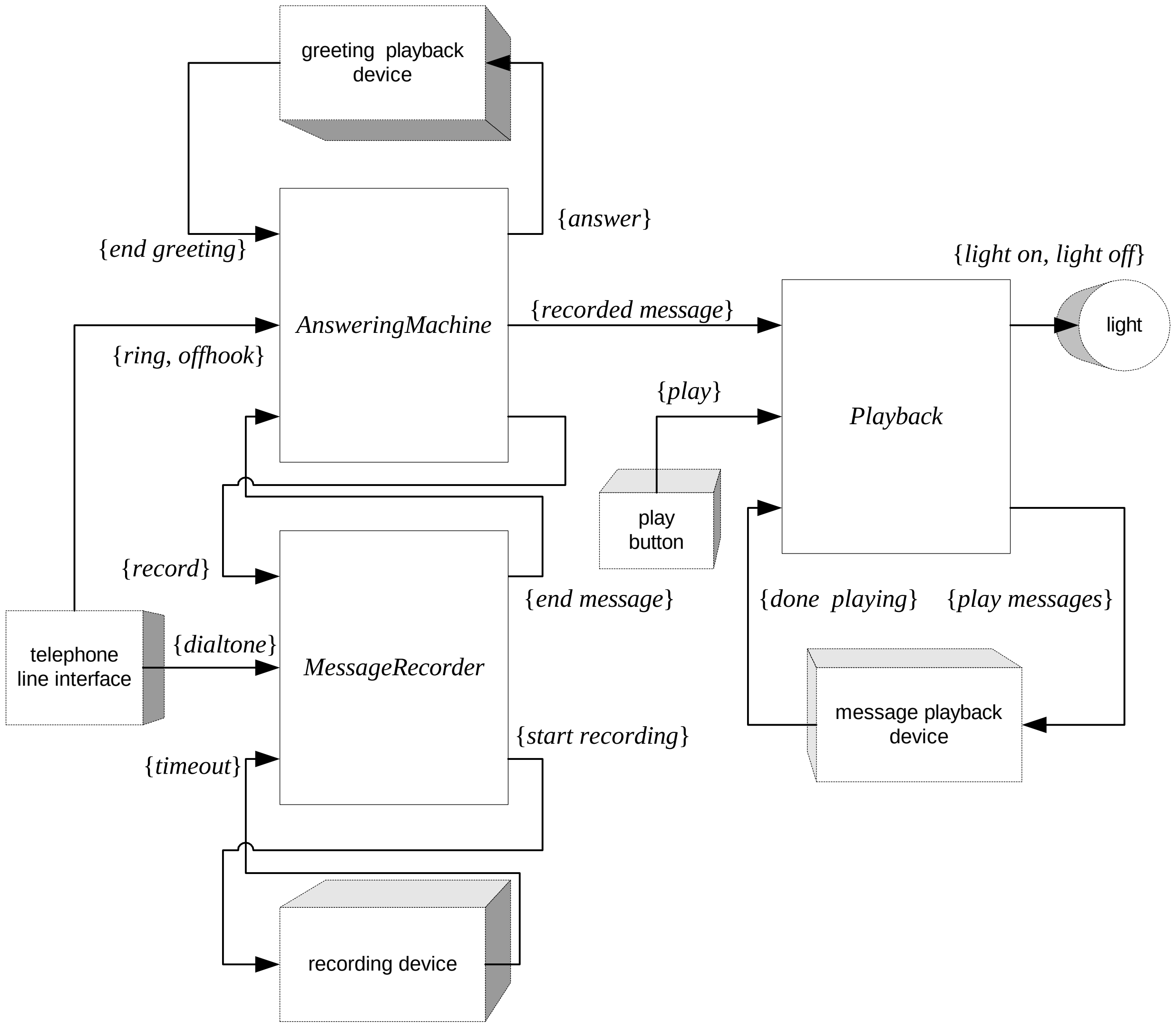


Figure 4.21: Answering machine composition with feedback. The *absent* elements are not shown (to reduce clutter).

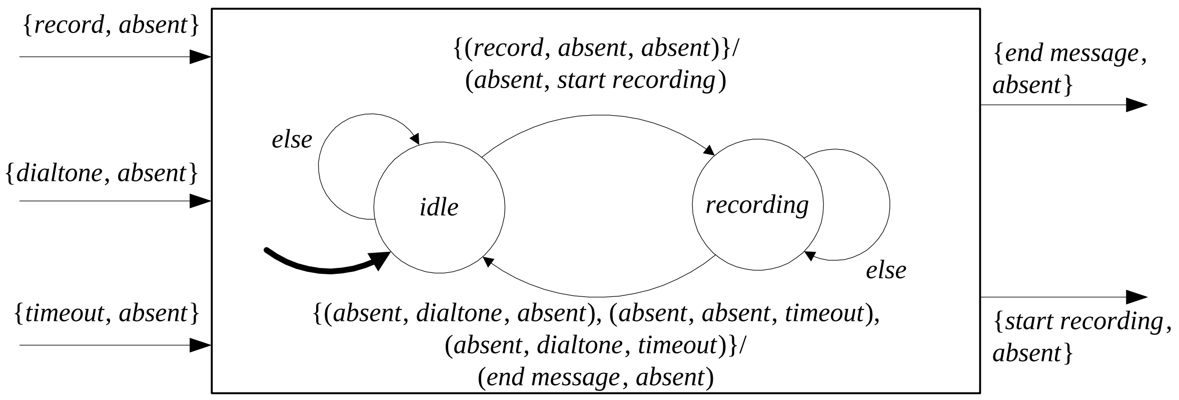

{record, absent}
{dialtone, absent}
{timeout, absent}
{(record, absent, absent)}/
(absent, start recording)
else
idle
recording
else
{(absent, dialtone, absent), (absent, absent, timeout),
(absent, dialtone, timeout)}/
(end message, absent)
{end message, absent}
{start recording, absent}

Figure 4.22: Message recorder subsystem of the answering system.

do exactly this. Successfully compiling a program involves proving that feedback loops are well-formed.

4.7.5 Exhaustive search

If a feedback composition has one or more machines with state-determined output, then finding a unique fixed point is easy. Without such state-determined output, we can apply the procedure in the previous section. Unfortunately, if the procedure fails, we cannot conclude that the composition is ill-formed. The procedure fails for example 4.7, shown in figure 4.17, despite the fact that this example is well-formed. For that example, we can determine the unique fixed point by exhaustive search. That is, for each reachable state of the composition, and for each possible input to the composition, we try all possible transitions out of the current states of the component machines. We reject those that lead to a contradiction. For example, in figure 4.17, assuming the current state is 1, the output of the component machine cannot be *maybe* because then the input would have to be *maybe*, which would result in the output being *absent*. If after rejecting all contradictions there remains exactly one possibility in each reachable state, then the composition is well-formed.

Exhaustive search works in figure 4.17 only because the number of reachable states is finite and the number of transitions out of each state is finite. If either of these conditions is violated, then exhaustive search will not work. Thus, there are state machine that when put in a feedback loop are well-formed, but where there is no constructive procedure for evaluating a reaction (see box on page 142). Even when exhaustive search is theoretically possible, in practice the number of possibilities that must be tried grows extremely fast.

4.7.6 Nondeterministic machines

Nondeterministic state machines can be composed just as deterministic state machines are composed. In fact, since deterministic state machines are a special case, the two types of state machines can be mixed in a composition. Compositions without feedback are straightforward, and operate

Probing further: Constructive Semantics

The term "**semantics**" means meaning. We have defined the meaning of compositions of state machines using the notion of synchrony, which makes feedback compositions particularly interesting. When we define "well-formed," we are, in effect, limiting the compositions that are valid. Compositions that are not well-formed fall outside our synchronous semantics. They have no meaning.

One way to define the semantics of a composition is to give a procedure for evaluating the composition (the resulting procedure is called an **operational semantics**). We have given three successively more difficult procedures for evaluating a reaction of a composition of state machines with feedback. If at least one machine in each directed loop has state-determined output, then it is easy to evaluate a reaction. If not, we can apply the constructive procedure of section 4.7.4. However, that procedure may result in some feedback connections remaining *unknown* even though the composition is well-formed. The ultimate procedure is exhaustive search, as described in section 4.7.5. However, exhaustive search is not always possible, and even when it is theoretically possible, the number of possibilities to explore may be so huge that it is not practical. There are state machines that when put in a feedback loop are well-formed, but where there is no constructive procedure for evaluating a reaction, and no constructive way to demonstrate that they are well-formed. Thus, there is no operational semantics for our feedback compositions.

This situation is not uncommon in computing and in mathematics. Kurt Gödel's famous incompleteness theorem (1931), for example, states (loosely) that in any formal logical system, there are statements that are true but not provable. This is analogous in that we can have feedback compositions that are well-formed, but we have no procedure that will always work to demonstrate that they are well-formed. Around the same time, Alan Turing and Alonzo Church demonstrated that there are functions that cannot be computed by any procedure.

To deal with this issue, Gerard Berry has proposed that synchronous composition have a **constructive semantics**, which means precisely that well-formed compositions are defined to be those for which the constructive procedure of section 4.7.4 works. When that procedure fails, we simply declare the composition to be unacceptable. This is pragmatic solution, and in many situations, it is adequate. However, particularly in the field of feedback control, it proves too restrictive. Most feedback control systems in practical use would be rejected by this semantics.

See G. Berry, *The Constructive Semantics of Pure Esterel*, Book Draft, http://www.esterel-technologies.com/corporate/berry.htm.

almost exactly as described above (see exercises 14 and 15). Compositions with feedback require a small modification to our evaluation process.

Recall that to evaluate the reaction of a feedback composition, we begin by setting to *unknown* any input symbols that are not initially known. We then proceed through a series of rounds where in each round, we attempt to determine the output symbols of the state machines in the composition given what we know about the input symbols. After some number of rounds, no more information is gained. At this point, if all of the input and output symbols are known, then the composition is well-formed. This procedure works for most (but not all) well-formed compositions.

This process needs to be modified slightly for nondeterministic machines because in each reaction, a machine may have several possible output symbols and several possible next states. For each machine, at each reaction, we define the sets $PossibleInputs \subset Inputs$, $PossibleNextStates \subset States$ and $PossibleNextOutputs \subset Outputs$. If the inputs to a particular machine in the composition are known completely, then *PossibleInputs* has exactly one element. If they are completely unknown, then *PossibleInputs* is empty.

The rounds proceed in a similar fashion to before. For each state machine in the composition, given what is known about the input symbols, i.e. given *PossibleInputs*, determine what you can about the next state and output symbols. This may result in elements being added to *PossibleNextStates* and *PossibleNextOutputs*. When a round results in no such added elements, the process has converged. If none of the *PossibleInputs* or *PossibleOutputs* sets is empty, then the composition is well-formed.

4.8 Summary

Many systems are designed as state machines. Usually the design is structured by composing component state machines. In this chapter, we considered synchronous composition. Feedback composition proves particularly subtle because the input symbol of a state machine in a reaction may depend on its own output symbol in the same reaction. We call a feedback composition well-formed if every signal has a unique non-stuttering symbol in each reaction. In lab C.4, you contruct a well-formed feedback composition of state machines.

Describing systems as compositions of state machines helps in many ways. It promotes understanding. The block diagram syntax that describes the structure often shows that individual components are responsible for distinct functions of the overall system. Some components may already be available and so we can reuse their designs. The design of the answering machine in figure 4.9 takes into account the availability of the telephone line interface, recording device, etc. Composition also simplifies description; once we specify the component state machines and the composition, the overall state machine is automatically defined by the rules of composition. Compilers for synchronous programming languages and tools for verification do this automatically.

We have three successively more difficult procedures for evaluating a reaction of a composition of state machines with feedback. If at least one machine in each directed loop has state-determined output, then it is easy to evaluate a reaction. If not, we can apply the constructive procedure of section 4.7.4. But that procedure may be inconclusive. The ultimate procedure is exhaustive search, as described in section 4.7.5. However, exhaustive search is not always possible, and even when it is theoretically possible, the number of possibilities to explore may be so huge that it is not practical.

Exercises

In some of the following exercises you are asked to design state machines that carry out a given task. The design is simplified and elegant if the state space is properly chosen. Although the state space is not unique, there often is a natural choice. Each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **E** Define the composite state machine in figure 4.7 in terms of the component machines, as done for the simpler compositions in figures 4.3 and 4.1. Be sure to state any required assumptions.

2. **E** Define the composite state machine in figure 4.10 in terms of the component machines, as done for the simpler compositions in figures 4.3 and 4.1. Be sure to state any required assumptions. Give the definition in two different ways:

   (a) Directly form a product of the three state spaces.

   (b) First compose the *A* and *B* state machines to get a new *D* state machine, and then compose *D* with *C*.

   (c) Comment on the relationship between the models in part (a) and (b).

3. **T** Consider the state machine *UnitDelay* studied in part (a) of exercise 5 at the end of the previous chapter.

   (a) Construct a state machine model for a cascade composition of two such machines. Give the sets and functions model (it is easier than the state transition diagram or table).

   (b) Are all of the states in the state space of your model in part (a) reachable? If not, give an example of an unreachable state.

   (c) Give the state space (only) for cascade compositions of three and four unit delays. How many elements are there in each of these state spaces?

   (d) Give an expression for the size of the state space as function of the number *N* of cascaded delays in the cascade composition.

4. **C** Consider the parking meter example of the previous chapter, example 3.1, and the modulo *N* counter of exercise 4 at the end of the previous chapter. Use these two machines to model a citizen that parks at the meter when the machines start, and inserts 25 cents every 30 minutes, and a police officer who checks the meter every 45 minutes, and issues a ticket if the meter is expired. For simplicity, assume that the police office issues a new ticket each time he finds the meter expired, and that the citizen remains parked forever.

   You may construct the model at the block diagram level, as in figure 4.9, but describe in words any changes you need to make to the designs of the previous chapter. Give state transition diagrams for any additional state machines you need. How long does it take for the citizen to get the first parking ticket?

Assume you have an eternal clock that emits an event *tick* every minute.

Note that the output alphabet of the modulo N counter does not match the input alphabet of the parking meter. Neither does its input alphabet match the output alphabet of the parking meter. Thus, one or more intermediate state machines will be needed to translate these alphabets. You should fully specify these state machines (i.e., don't just give them at the block diagram level). **Hint:** These state machines, which perform an operation called **renaming**, only need one state.

5. **C** Consider a machine with

$$\begin{aligned} States &= \{0,1,2,3\}, \\ Inputs &= \{increment, decrement, reset, absent\}, \\ Outputs &= \{zero, absent\}, \\ initialState &= 0, \end{aligned}$$

such that *increment* increases the state by 1 (modulo 4), *decrement* decreases the state by 1 (modulo 4), *reset* resets the state to 0, and the output symbol is *absent* unless the next state is 0, in which case the output symbol is *zero*. So, for example, if the current state is 3 and the input is *increment*, then the new state will be 0 and the output will be *zero*. If the current state is 0 and the input is *decrement*, then the new state will be 3 and the output will be *absent*.

(a) Give the *update* function for this machine, and sketch the state transition diagram.

(b) Design a cascade composition of two state machines, each with two states, such that the composition has the same behaviors as the one above. Give a diagram of the state machines and their composition, and carefully define all the input and output alphabets.

(c) Give a bisumulation relation between the single machine and the cascade composition.

6. **C** A road has a pedestrian crossing with a traffic light. The light is normally green for vehicles, and the pedestrian is told to wait. However, if a pedestrian presses a button, the light turns yellow for 30 seconds and then red for 30 seconds. When it is red, the pedestrian is told "cross now." After the 30 seconds of red, the light turns back to green. If a pedestrian presses the button again while the light is red, then the red is extended to a full minute.

Construct a composite model for this system that has at least two state machines, *TrafficLight* for the traffic light seen by the cars, and *WalkLight* for the walk light seen by the pedestrians. The state of machine should represent the state of the lights. For example, *TrafficLight* should have at least three states, one for green, one for yellow, and one for red. Each color may, however, have more than one state associated with it. For example, there may be more than one state in which the light is red. It is typical in modeling systems for the states of the model to represent states of the physical system.

Assume you have a timer available such that if you emit an output *start timer*, then 30 seconds later an input symbol *timeout* will appear. It is sufficient to give the state transition graphs for the machines. State any assumptions you need to make.

7. **E** Suppose you are given two state machines A and B, Suppose the sizes of the input alphabets are i_A, i_B, respectively, the sizes of the output alphabets are o_A, o_B respectively, and the numbers of states are s_A, s_B, repectively. Give the sizes of the input and output alphabets and the number of states for the following compositions:
   (a) side-by-side,
   (b) cascade,
   (c) and feedback, where the structure of the feedback follows the pattern in figure 4.16(a).

8. **T** Example 4.2 shows a state machine in which a state is not reachable from the initial state. Here is a recursive algorithm to calculate the reachable states for any nondeterministic machine,

$$StateMachine = (States, Inputs, Outputs, possibleUpdates, initialState).$$

   Recursively define subsets $ReachableStates(n)$, $n = 0, 1, \cdots$ of $States$ by:

$$\begin{aligned} ReachableStates(0) &= \{initialState\}, \text{ and for } n \geq 0 \\ ReachableStates(n+1) &= \{s(n+1) \mid \exists s(n) \in ReachableStates(n), \exists x(n) \in Inputs, \exists y(n) \in Outputs \\ &\quad (s(n+1), y(n)) \in possibleUpdates(s(n), x(n))\} \\ &\quad \cup ReachableStates(n). \end{aligned}$$

   In words: $ReachableStates(n+1)$ is the set of states that can be reached from $ReachableStates(n)$ in one step using any input symbol, together with $ReachableStates(n)$.

   (a) Show that for all n, $ReachableStates(n) \subset ReachableStates(n+1)$.
   (b) Show that $ReachableStates(n)$ is the set of states that can be reached in n or fewer steps, starting in $initialState$. Now show that if for some n,

$$ReachableStates(n) = ReachableStates(n+1), \tag{4.11}$$

   then $ReachableStates(n) = ReachableStates(n+k)$ for all $k \geq 0$.
   (c) Suppose (4.11) holds for $n = N$. Show that $ReachableStates(N)$ is the set of all reachable states, i.e. this set comprises all the states that can be reached using any input sequence starting in $initialState$.
   (d) Suppose there are N states. Show that (4.11) holds for $n = N$.
   (e) Compute $ReachableStates(n)$ for all n for the machine in figure 4.4.
   (f) Suppose $States$ is infinite. Show that the set of reachable states is given by

$$\cup_{n=0}^{\infty} ReachableStates(n).$$

9. **T** The algorithm in Exercise 8 has a fixed point interpretation. For a nondeterministic state machine,

$$StateMachine = (States, Inputs, Outputs, possibleUpdates, initialState),$$

define the function $nextStep : P(States) \to P(States)$ that maps subsets of *States* into subsets of *States* (recall that $P(A)$ is the power set of A) as follows: for any $S(n) \subset States$

$$\begin{aligned} nextStep(S(n)) &= \{s(n+1) \mid \exists s(n) \in S(n), \exists x(n) \in Inputs, \exists y(n) \in Outputs, \\ &\quad (s(n+1), y(n)) \in possibleUpdates(s(n), x(n))\} \cup S(n). \end{aligned}$$

By definition, a fixed point of *nextStep* is any subset $S \subset States$ such that $nextStep(S) = S$.

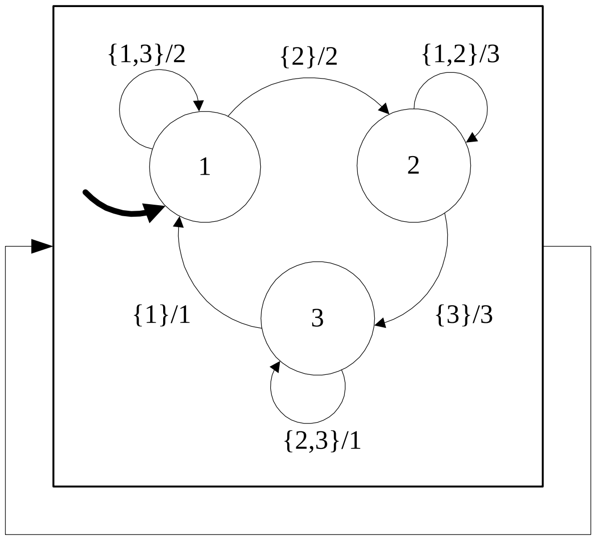
(a) Show that $\emptyset$ and *States* are both fixed points of *nextState*.

(b) Let *ReachableStates* be the set of all states that can be reached starting in *initialState*. Show that *ReachableStates* is also a fixed point.

(c) Show that *ReachableStates* is the least fixed point of *nextStep* containing *initialState*.

10. **C** Recall the playback machine of figure 4.8 and the *CodeRecognizer* machine of Figure 3.4. Enclose *CodeRecognizer* in a block and compose it with the playback machine so that someone can play back the recorded messages only if she correctly enters the code 1100. You will need to modify the playback machine appropriately.

11. **E** Consider the following state machine in a feedback composition, where the input and output alphabets for the state machine is $\{1, 2, 3, absent\}$:



Is it well-formed? If so, then find the output symbols for the first 10 reactions.

12. **E** In this problem, we will explore the fact that a carefully defined delay in a feedback composition always makes the composition well-formed.

(a) For an input and output alphabet

$$Inputs = Outputs = \{true, false, absent\}$$

design a state machine that outputs *false* on the first reaction, and then in subsequent reactions, outputs the value observed at the input in the previous reaction. This is similar to *UnitDelay* of problem 5 at the end of chapter 3, with the only difference being that it outputs an initial *false* instead of *absent*.

(b) Compose the machine in figure 4.14 (b) with the delay from part (a) of this problem in a feedback loop (as in figure 4.16). Give an argument that the composition is well-formed. Then do the same for figure 4.14 (c) instead of (b).

13. **C** Construct a feedback state machine with the structure of figure 4.12 that outputs the periodic sequence $a, b, c, a, b, c \cdots$ (with, as usual, any number of intervening stuttering outputs between the non-stuttering outputs).

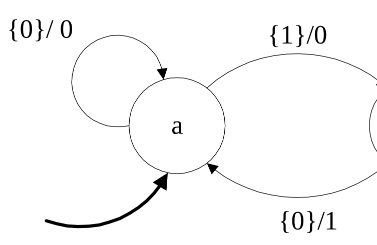
14. **E** Modify figure 4.1 as necessary so that the machines in the side-by-side composition are both nondeterministic.

15. **E** Modify figure 4.3 as necessary so that the machines in the cascade composition are both nondeterministic.

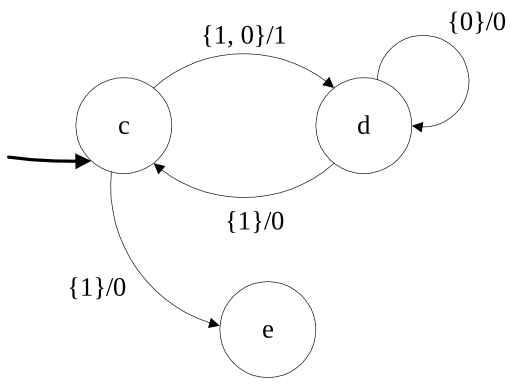
16. **C,T** Data packets are to be reliably exchanged between two computers over communication links that may lose packets. The following protocol has been suggested. Suppose computer *A* is sending and *B* is receiving. Then *A* sends a packet and starts a timer. If *B* receives the packet it sends back an acknowledgment. (The packet or the acknowledgment or both may be lost.) If *A* does not receive the acknowledgment before the timer expires, it retransmits the packet. If the acknowledgment arrives before the timer expires, *A* sends the next packet.

    (a) Construct two state machines, one for *A* and one for *B*, that implement the protocol.
    (b) Construct a two-state nondeterministic machine to model the link from *A* to *B*, and another copy to model the link from *B* to *A*. Remember that the link may correctly deliver a packet, or it may lose it.
    (c) Compose the four machines to model the entire system.
    (d) Suppose the link correctly delivers a packet, but after a delay that exceeds the timer setting. What will happen?

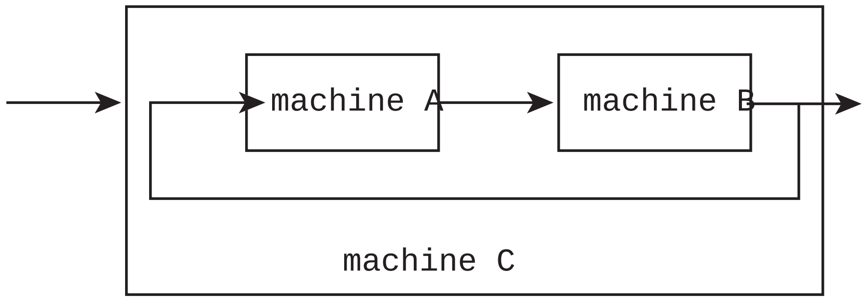
17. **T** Consider the following three state machines:



machine A



machine B



Machines A and B have input and output alphabets

$$Inputs = Outputs = \{0, 1, absent\}.$$

Machine C has the same output alphabet, but input alphabet $Inputs_C = \{react, absent\}$.

(a) Which of these machines is deterministic?

(b) Draw the state transition diagram for the composition (machine C), showing only states that are reachable from the initial state.

(c) Give the $Behaviors_C$ relation for the composition of machine C, ignoring stuttering.

18. **T** The feedback composition in figure 4.14(c) is ill-formed because it has two non-stuttering fixed points in each of the two states of the component machine. Instead of declaring it to be ill-formed, we could have interpreted the composition as representing a nondeterministic state machine. That is, in each state, we accept either of the two possible fixed points as possible reactions of the machine. Using this interpretation, give the nondeterministic machine for the feedback composition by giving its sets and functions model and a state transition diagram.