

## Chapter 8 *Computation models*

### 8.1 Introduction

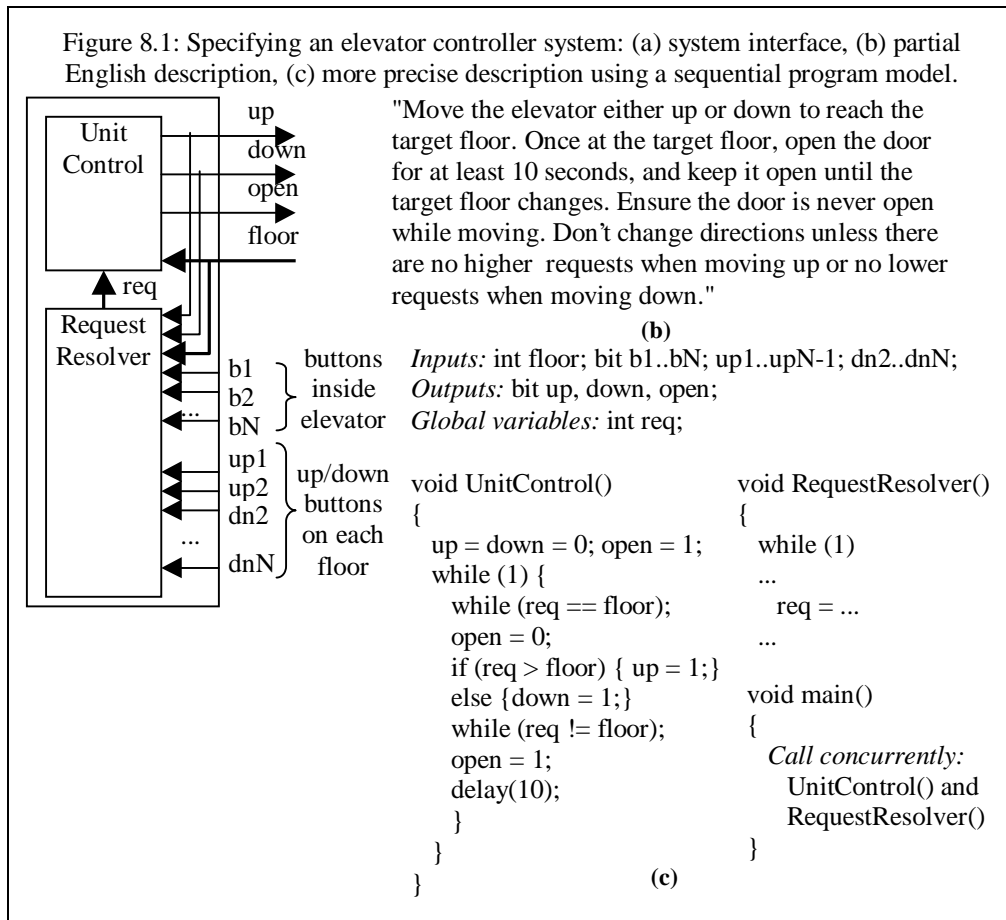
We implement a system's processing behavior with processors. But to accomplish this, we must have first described that processing behavior. One method we've discussed for describing processing behavior uses assembly language. Another, more powerful method uses a high-level programming language like C. Both these methods use what is known as a sequential program computation model, in which a set of instructions executes sequentially. A high-level programming language provides more advanced constructs for sequencing among the instructions than does an assembly language, and the instructions are more complex, but nevertheless, the sequential execution model (one statement at a time) is the same.

However, embedded system processing behavior is becoming very complex, requiring more advanced computation models to describe that behavior. The increasing complexity results from increasing IC capacity: the more we can put on an IC, the more functionality we want to put into our embedded system. Thus, while embedded systems previously encompassed applications like washing machines and small games requiring perhaps hundreds of lines of code, today they also extend to fairly sophisticated applications like television set-top boxes and digital cameras requiring perhaps hundreds of thousands of lines.

Trying to describe the behavior of such systems can be extremely difficult. The desired behavior is often not even fully understood initially. Therefore, designers must spend much time and effort simply understanding and describing the desired behavior of a system, and some studies have found that most system bugs come from mistakes made describing the desired behavior rather than from mistakes in implementing that behavior. The common method today of using an English (or some other natural language) description of desired behavior provides a reasonable first step, but is not nearly sufficient, because English is not precise. Trying to describe a system precisely in English can be an arduous and often futile endeavor -- just look at any legal document for any example of attempting to be precise in a natural language.

A computation model assists the designer to understand and describe the behavior by providing a means to compose the behavior from simpler objects. A *computation model* provides a set of objects, rules for composing those objects, and execution semantics of the composed objects. For example, the sequential program model provides a set of statements, rules for putting statements one after another, and semantics stating how the statements are executed one at a time. Unfortunately, this model is often not enough. Several other models are therefore also used to describe embedded system behavior. These include the communicating process model, which supports description of multiple sequential programs running concurrently. Another model is the state machine model, used commonly for control-dominated systems. A *control-dominated* system is one whose behavior consists mostly of monitoring control inputs and reacting by setting control outputs. Yet another model is the dataflow model, used for data-dominated systems. A *data-dominated* system's behavior consists mostly of transforming streams of input data into streams of output data, such as a system for filtering noise out of an audio signal as part of a cell phone. An extremely complex system may be best described using an object-oriented model, which provides an elegant means for breaking the complex system into simpler, well-defined objects.

A model is an abstract notion, and therefore we use *languages* to capture the model in a concrete form. For example, the sequential program model can be captured in a variety of languages, such as C, C++, Pascal, Java, Basic, Ada, VHDL, and Verilog. Furthermore, a single language can capture a variety of models. Languages typically are textual, but may also be graphical. For example, graphical languages have been proposed for sequential programming (though they have not been widely adopted).



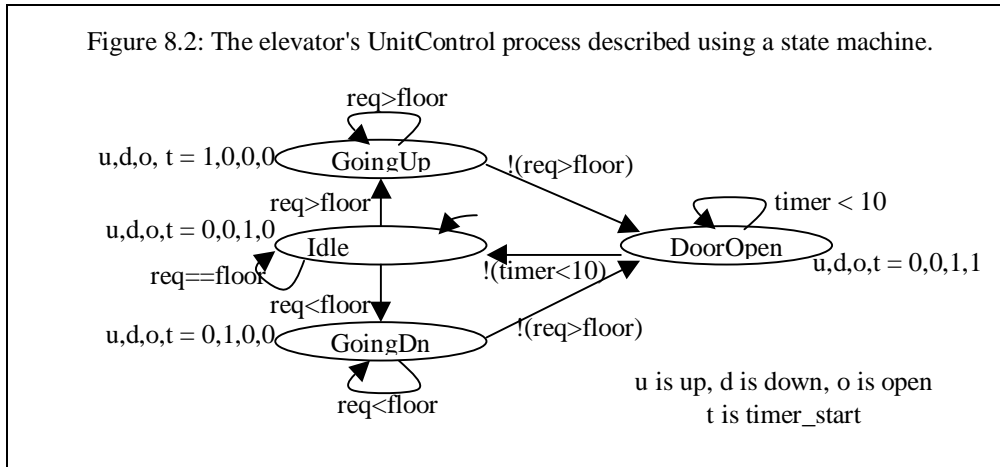
An earlier chapter focused on the sequential program model. This chapter will focus on the state machine and concurrent process models, both of which are commonly used in embedded systems.

## 8.2 Sequential program model

We described the sequential program model in Chapter 2. Here, we introduce an example system that we'll use in the chapter, and we'll use the sequential program model to describe part of the system. Consider the simple elevator controller system in Figure 8.1(a). It has several control inputs corresponding to the floor buttons inside the elevator and corresponding to the up and down buttons on each of the  $N$  floors at which the elevator stops. It also has a data input representing the current floor of the elevator. It has three control outputs that make the elevator move up or down, and open the elevator door. A partial English description of the system's desired behavior is shown in Figure 8.1(b).

We decide that this system is best described as two blocks. `RequestResolver` resolves the various floor requests into a single requested floor. `UnitControl` actually moves the elevator unit to this requested floor, as shown in Figure 8.1. Figure 8.1(c) shows a sequential program description for the `UnitControl` process. Note that this process is more precise than the English description. It firsts opens the elevator door, and then enters an infinite loop. In this loop, it first waits until the requested and current floors differ. It then closes the door and moves the elevator up or down. It then waits until the current floor equals the requested floor, stops moving the elevator, and opens the door for 10 seconds (assuming there's a routine called `delay`). It then goes back to the beginning of the infinite loop. The `RequestResolver` would be written similarly.

Figure 8.2: The elevator's UnitControl process described using a state machine.



### 8.3 State machine model

In a *state machine* model, we describe system behavior as a set of possible states; the system can only be in one of these states at a given time. We also describe the possible transitions from one state to another depending on input values. Finally, we describe the actions that occur when in a state or when transitioning between states.

For example, Figure 8.2 shows a state machine description of the `UnitControl` part of our elevator example. The initial state, `Idle`, sets up and down to 0 and open to 1. The state machine stays in state `Idle` until the requested floor differs from the current floor. If the requested floor is greater, then the machine transitions to state `GoingUp`, which sets up to 1, whereas if the requested floor is less, then the machine transitions to state `GoingDn`, which sets down to 1. The machine stays in either state until the current floor equals the requested floor, after which the machine transitions to state `DoorOpen`, which sets open to 1. We assume the system includes a timer, so we start the timer while transitioning to `DoorOpen`. We stay in this state until the timer says 10 seconds have passed, after which we transition back to the `Idle` state.

#### 8.3.1 Finite-state machines: FSM

We have described state machines somewhat informally, but now provide a more formal definition. We start by defining the well-known *finite-state machine* computation model, or *FSM*, and then we'll define extensions to that model to obtain a more useful model for embedded system design. An FSM is a 6-tuple,  $\langle S, I, O, F, H, s_0 \rangle$ , where:

- S is a set of states  $\{s_0, s_1, \dots, s_l\}$ ,
- I is a set of inputs  $\{i_0, i_1, \dots, i_m\}$ ,
- O is a set of outputs  $\{o_0, o_1, \dots, o_n\}$ ,
- F is a next-state function (i.e., transitions), mapping states and inputs to states ( $S \times I \rightarrow S$ ),
- H is an output function, mapping current states to outputs ( $S \rightarrow O$ ), and
- $s_0$  is an initial state.

The above is a *Moore*-type FSM above, which associates outputs with states. A second type of FSM is a *Mealy*-type FSM, which associates outputs with transitions, i.e., H maps  $S \times I \rightarrow O$ . You might remember that Moore outputs are associated with states by noting that the name *Moore* has two *o*'s in it, which look like states in a state diagram. Many tools that support FSM's support combinations of the two types, meaning we can associate outputs with states, transitions, or both.

We can use some shorthand notations to simplify FSM descriptions. First, there may be many system outputs, so rather than explicitly assigning every output in every state, we can say that any outputs not assigned in a state are implicitly assigned 0. Second, we often use an FSM to describe a single-purpose processor (i.e., hardware). Most hardware is synchronous, meaning that register updates are synchronized to clock pulses, e.g., registers are only updated on the rising (or falling) edge of a clock. Such an FSM would have every transition condition AND'ed with the clock edge (e.g., clock'rising and x and y). To avoid having to add this clock edge to every transition condition, we can simply say that the FSM is synchronous, meaning that every transition condition is implicitly AND'ed with the clock edge.

### 8.3.2 Finite-state machines with datapaths: FSMD

When using an FSM for embedded system design, the inputs and outputs represent boolean data types, and the functions therefore represent boolean functions with boolean operations. This model is sufficient for purely control systems that do not input or output data. However, when we must deal with data, two new features would be helpful: more complex data types (such as integers or floating point numbers), and variables to store data. Gajski refers to an FSM model extended to support more complex data types and variables as an FSM with datapath, or *FSMD*. Most other authors refer to this model as an extended FSM, but there are many kinds of extensions and therefore we prefer the more precise name of FSMD. One possible FSMD model definition is as follows:

$\langle S, I, O, V, F, H, s_0 \rangle$  where:

$S$  is a set of states  $\{s_0, s_1, \dots, s_l\}$ ,

$I$  is a set of inputs  $\{i_0, i_1, \dots, i_m\}$ ,

$O$  is a set of outputs  $\{o_0, o_1, \dots, o_n\}$ ,

$V$  is a set of variables  $\{v_0, v_1, \dots, v_n\}$ ,

$F$  is a next-state function, mapping states and inputs and variables to states ( $S \times I \times V \rightarrow S$ ),

$H$  is an action function, mapping current states to outputs and variables ( $S \rightarrow O \cup V$ ), and

$s_0$  is an initial state.

In an FSMD, the inputs, outputs and variables may represent various data types (perhaps as complex as the data types allowed in a typical programming language), and the functions  $F$  and  $H$  therefore may include arithmetic operations, such as addition, rather than just boolean operations as in an FSM. We now call  $H$  an action function rather than an output function, since it describes not just outputs, but also variable updates. Note that the above definition is for a Moore-type FSMD, and it could easily be modified for a Mealy type or a combination of the two types. During execution of the model, the complete system state consists not only of the current state  $s_i$ , but also the values of all variables. Our earlier state machine description of `UnitControl` was an FSMD, since it had inputs whose data types were integers, and had arithmetic operations (comparisons) in its transition conditions.

### 8.3.3 Describing a system as a state machine

Describing a system's behavior as a state machine (in particular, as an FSMD) consists of several steps:

1. List all possible states, giving each a descriptive name.
2. Declare all variables.
3. For each state, list the possible transitions, with associated conditions, to other states.
4. For each state and/or transition, list the associated actions

5. For each state, ensure that exiting transition conditions are exclusive (no two conditions could be true simultaneously) and complete (one of the conditions is true at any time).

If the transitions leaving a state are not exclusive, then we have a *non-deterministic* state machine. When the machine executes and reaches a state with more than one transition that could be taken, then one of those transitions is taken, but we don't know which one that would be. The non-determinism prevents having to over-specify behavior in some cases, and may result in don't-cares that may reduce hardware size, but we won't focus on non-deterministic state machines in this book.

If the transitions leaving a state are not complete, then that usually means that we stay in that state until one of the conditions becomes true. This way of reducing the number of explicit transitions should probably be avoided when first learning to use state machines.

### 8.3.4 Comparing the state machine and sequential program models

Many would agree that the state machine model excels over the sequential program model for describing a control-based system like the elevator controller. The state machine model is designed such that it encourages a designer to think of all possible states of the system, and to think of all possible transitions among states based on possible input conditions. The sequential program model, in contrast, is designed to transform data through a series of instructions that may be iterated and conditionally executed. Each encourages a different way of thinking of a system's behavior.

A common point of confusion is the distinction between state machine and sequential program models versus the distinction between graphical and textual languages. In particular, a state machine description excels in many cases, not because of its graphical representation, but rather because it provides a more natural means of computing for those cases; it can be captured textually and still provide the same advantage. For example, while in Figure 8.2 we described the elevator's `UnitControl` as a state machine captured in a graphical state-machine language, called a state diagram, we could have instead captured the state machine in a textual state-machine language. One textual language would be a state table, in which we list each state as an entry in a table. Each state's row would list the state's actions. Each row would also list all possible input conditions, and the next state for each such condition. Conversely, while in Figure 8.1 we described the elevator's `UnitControl` as a sequential program captured using a textual sequential programming language (in this case C), we could have instead captured the sequential program using a graphical sequential programming language, such as a flowchart.

Figure 8.3: Capturing the elevator's UnitControl state machine in a sequential programming language (in this case C).

```

#define IDLE      0
#define GOINGUP  1
#define GOINGDN  2
#define DOOROPEN 3
void UnitControl()
{
  int state = IDLE;
  while (1) {
    switch (state) {
      IDLE:    up=0; down=0; open=1; timer_start=0;
              if (req==floor) {state = IDLE;}
              if (req > floor) {state = GOINGUP;}
              if (req < floor) {state = GOINGDN;}
              break;
      GOINGUP: up=1; down=0; open=0; timer_start=0;
              if (req > floor) {state = GOINGUP;}
              if (!(req>floor)) {state = DOOROPEN;}
              break;
      GOINGDN: up=1; down=0; open=0; timer_start=0;
              if (req > floor) {state = GOINGDN;}
              if (!(req>floor)) {state = DOOROPEN;}
              break;
      DOOROPEN: up=0; down=0; open=1; timer_start=1;
                if (timer < 10) {state = DOOROPEN;}
                if (!(timer<10)){state = IDLE;}
                break;
    }
  }
}

```

### 8.3.5 Capturing a state machine model in a sequential programming language

As elegant as the state machine model is for capturing control-dominated systems, the fact remains that the most popular embedded system development tools use sequential programming languages like C, C++, Java, Ada, VHDL or Verilog. These tools are typically complex and expensive, supporting tasks like compilation, synthesis, simulation, interactive debugging, and/or in-circuit emulation. Unfortunately, sequential programming languages do not directly support the capture of state machines, i.e., they don't possess specific constructs corresponding to states or transitions. Fortunately, we can still describe our system using a state machine model while capturing the model in a sequential program language, by using one of two approaches.

In a *front-end tool* approach, we install an additional tool that supports a state machine language. These tools typically define graphical and perhaps textual state machine languages, and include nice graphic interfaces for drawing and displaying states as circles and transitions as directed arcs. They may support graphical simulation of the state machine, highlighting the current state and active transition. Such tools automatically generate code in a sequential program language (e.g., C code) with the same functionality as the state machine. This sequential program code can then be input

Figure 8.4: General template for capturing a state machine in a sequential programming language.

```

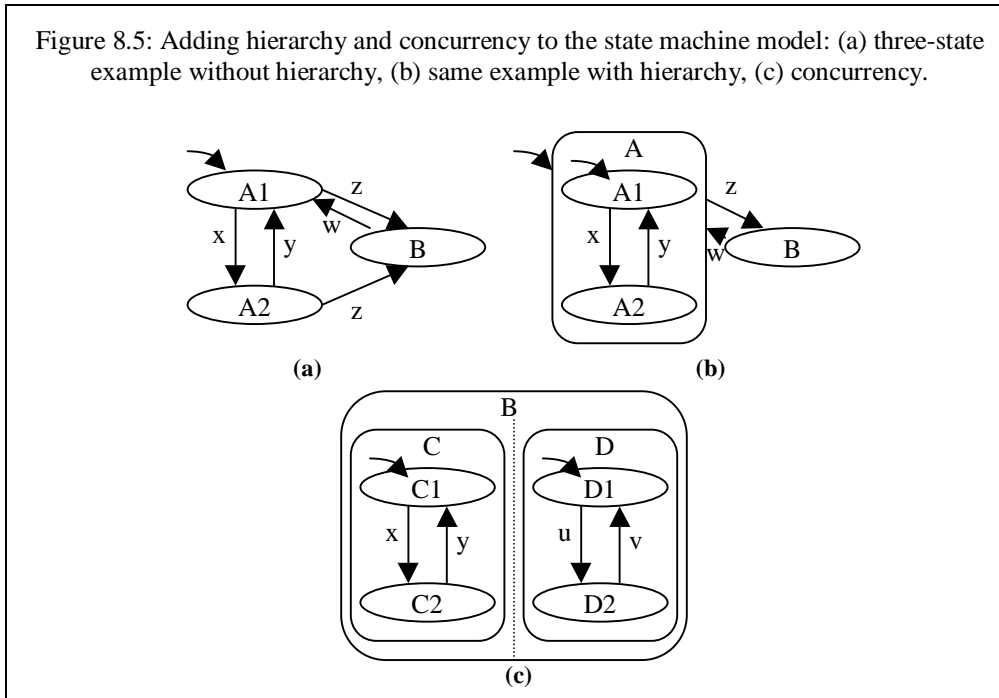
#define S0          0
#define S1          1
...
#define SN          N
void StateMachine()
{
    int state = S0; // or whatever is the initial state.
    while (1) {
        switch (state) {
            S0:
                // Insert S0's actions here
                // Insert transitions Ti leaving S0:
                if (T0's condition is true ) {state = T0's next state; // insert T0's actions here. }
                if (T1's condition is true ) {state = T1's next state; // insert T1's actions here. }
                ...
                if (Tm's condition is true ) {state = Tm's next state; // insert Tm's actions here. }
                break;
            S1:
                // Insert S1's actions here
                // Insert transitions Ti leaving S1
                break;
            ...
            SN:
                // Insert SN's actions here
                // Insert transitions Ti leaving SN
                break;
        }
    }
}

```

to our main development tool. In many cases, the front-end tool is designed to interface directly with our main development tool, so that we can control and observe simulations occurring in the development tool directly from the front-end tool. The drawback of this approach is that we must support yet another tool, which includes additional licensing costs, version upgrades, training, integration problems with our development environment, and so on.

In contrast, we can use a *language subset* approach. In this approach, we directly capture our state machine model in a sequential program language, by following a strict set of rules for capturing each state machine construct in an equivalent set of sequential program constructs. This approach is by far the most common approach for capturing state machines, both in software languages like C as well as hardware languages like VHDL and Verilog. We now describe how to capture a state machine model in a sequential program language.

We start by capturing our `UnitControl` state machine in the sequential programming language C, illustrated in Figure 8.3. We enumerate all states, in this case using the `#define` C construct. We capture the state machine as a subroutine, in which we declare a state variable initialized to the initial state. We then create an infinite loop, containing a single switch statement that branches to the case corresponding to the value of the state variable. Each state's case starts with the actions in that state, and then the transitions from that state. Each transition is captured as an if statement that checks if the



transition's condition is true and then sets the next state. Figure 8.4 shows a general template for capturing a state machine in C.

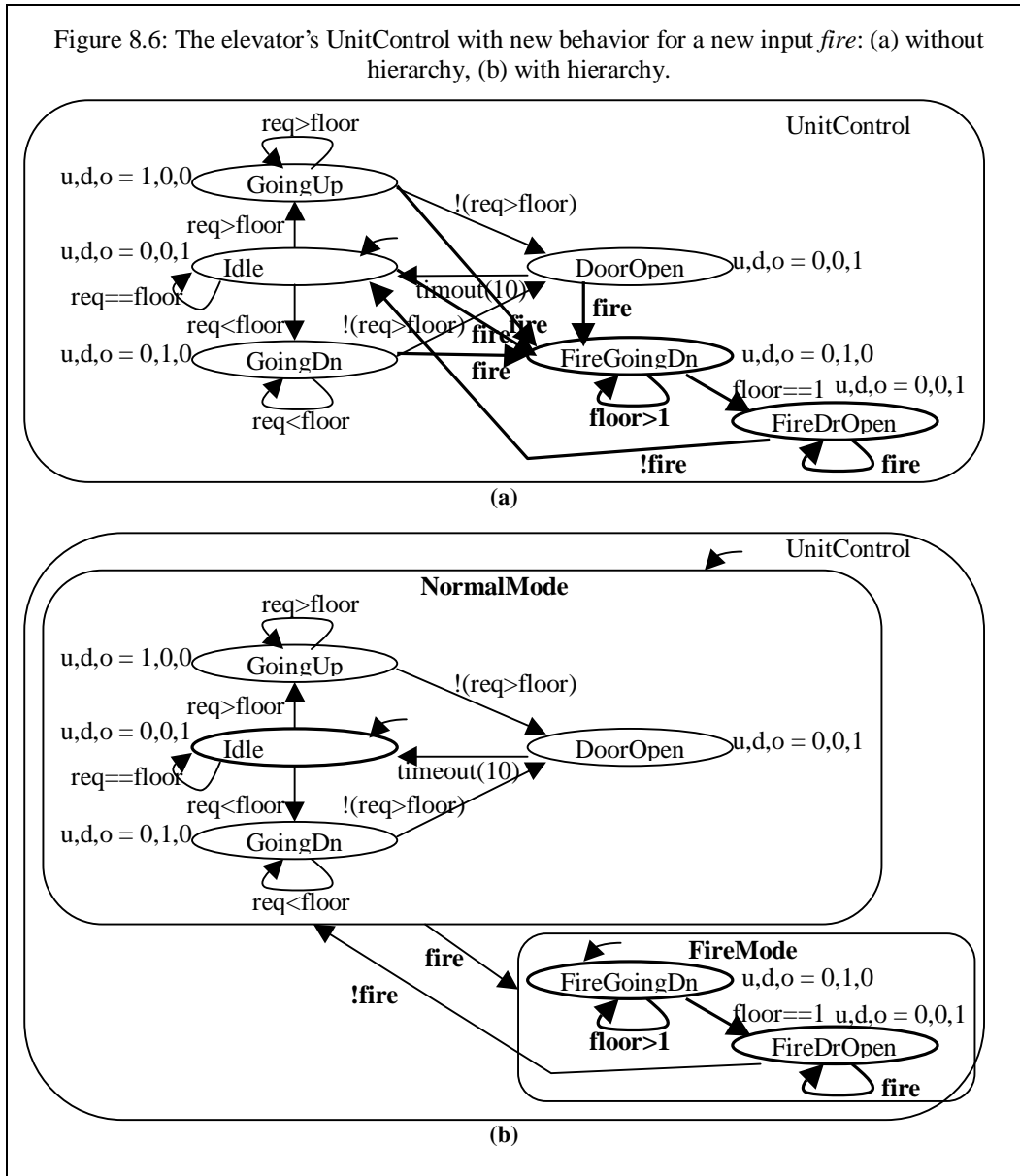
### 8.3.6 Hierarchical/Concurrent state machines (HCFSM) and Statecharts

Harel proposed extensions to the state machine model to support hierarchy and concurrency, and developed Statecharts, a graphical state machine language designed to capture that model. We refer to the model as a hierarchical/concurrent FSM, or HCFSM.

The *hierarchy* extension allows us to decompose a state into another state machine, or conversely stated, to group several states into a new hierarchical state. For example, consider the state machine in Figure 8.5(a), having three states A1 (the initial state), A2, and B. Whenever we are in either A1 or A2 and event *z* occurs, we transition to state B. We can simplify this state machine by grouping A1 and A2 into a hierarchical state A, as shown in Figure 8.5(b). State A is the initial state, which in turn has an initial state A1. We draw the transition to B on event *z* as originating from state A, not A1 or A2. The meaning is that regardless of whether we are in A1 or A2, event *z* causes a transition to state B.

As another hierarchy example, consider our earlier elevator example, and suppose that we want to add a control input *fire*, along with new behavior that immediately moves the elevator down to the first floor and opens the door when *fire* is true. As shown in Figure 8.6(a), we can capture this behavior by adding a transition from every state originally in *UnitControl* to a new state called *FireGoingDn*, which moves the elevator to the first floor, followed by a state *FireDrOpen*, which holds the door open on the first floor. When *fire* becomes false, we go to the *Idle* state. While this new state machine captures the desired behavior, it is becoming more complex due to many more transitions, and harder to comprehend due to more states. We can use hierarchy to reduce the number of transitions and enhance understandability. As shown in Figure 8.6(b), we can group the original state machine into a hierarchical state called *NormalMode*, and group the fire-related states into a state called *FireMode*. This grouping reduces the number of transitions, since instead of four transitions from each original state to the fire-related states, we need only one transition, from *NormalMode*

Figure 8.6: The elevator's UnitControl with new behavior for a new input *fire*: (a) without hierarchy, (b) with hierarchy.



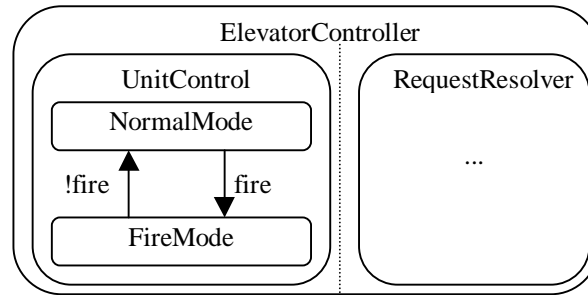
to FireMode. This grouping also enhances understandability, since it clearly represents two main operating modes, one normal and one in case of fire.

The *concurrency* extension allows us to use hierarchy to decompose a state into two concurrent states, or conversely stated, to group two concurrent states into a new hierarchical state. For example, Figure 8.5 (c), shows a state B decomposed into two concurrent states C and D. C happens to be decomposed into another state machine, as does D. Figure 8.7 shows the entire ElevatorController behavior captured as a HCFSM with two concurrent states.

Therefore, we see that there are two methods for using hierarchy to decompose a state into substates. *OR*-decomposition decomposes a state into sequential states, in which only one state is active at a time (either the first state OR the second state OR the third state, etc.). *AND*-decomposition decomposes a state into concurrent states, all of which are active at a time (the first state AND the second state AND the third state, etc.).

The Statecharts language includes numerous additional constructs to improve state machine capture. A *timeout* is a transition with a time limit as its condition. The transition

Figure 8.7: Using concurrency in an HCFSM to describe both processes of the ElevatorController.



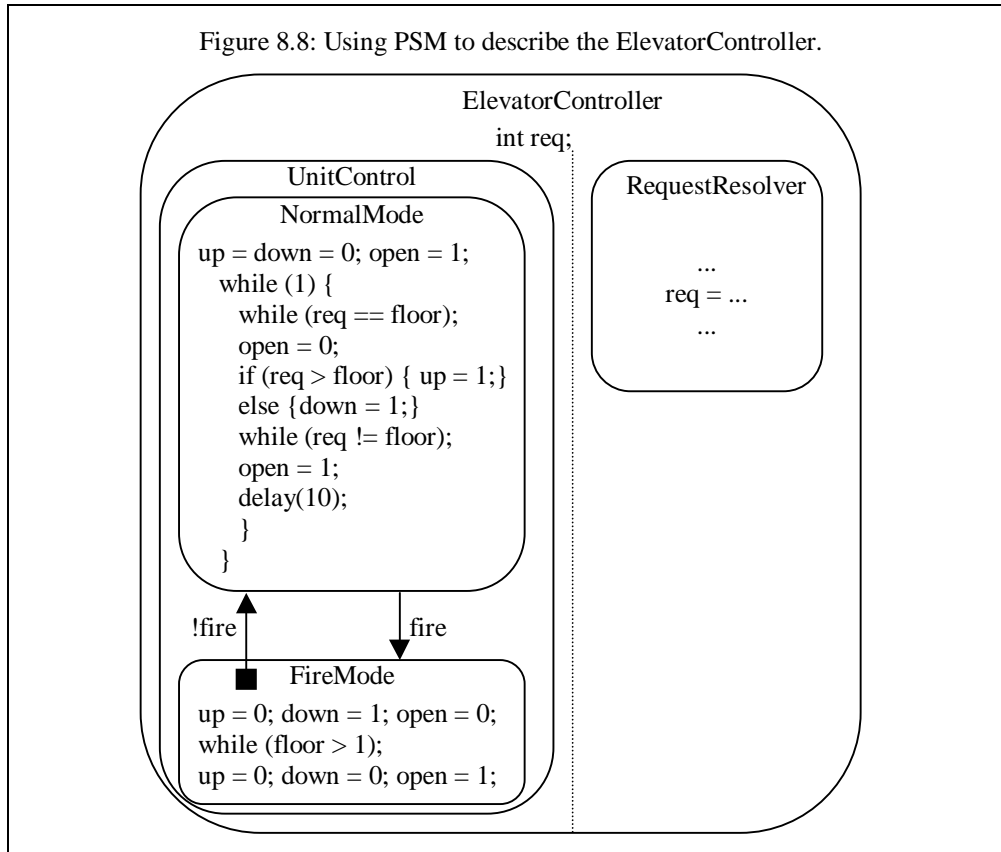
is automatically taken if the transition source state is active for an amount of time equal to the limit. Note that this would have simplified the `UnitControl` state machine; rather than starting and checking an external timer, we could simply have created a transition from `DoorOpen` to `Idle` with the condition `timeout(10)`. *History* is a mechanism for remembering the last substate that an OR-decomposed state A was in before transitioning to another state B. Upon re-entering state A, we can start with the remembered substate rather than A's initial state. Thus, the transition leaving A is treated much like an interrupt and B as an interrupt service routine.

### 8.3.7 Program-state machines (PSM)

The program-state machine (PSM) model extends state machines to allow use of sequential program code to define a state's actions (including extensions for complex data types and variables), as well as including the hierarchy and concurrency extensions of HCFSM. Thus, PSM is a merger of the HCFSM and sequential program models, subsuming both models. A PSM having only one state (called a program-state in PSM terminology), where that state's actions are defined using a sequential program, is the same as a sequential program. A PSM having many states, whose actions are all just assignment statements, is the same as an HCFSM. Lying between these two extremes are various combinations of the two models.

For example, Figure 8.8 shows a PSM description of the `ElevatorController` behavior, which we AND-decompose into two concurrent program-states `UnitControl` and `RequestResolver`, as in the earlier HCFSM example. Furthermore, we OR-decompose `UnitControl` into two sequential program-states, `NormalMode` and `FireMode`, again as in the HCFSM example. However, unlike the HCFSM example, we describe `NormalMode` as a sequential program (identical to that of Figure 8.1(c)) rather than a state machine. Likewise, we describe `FireMode` as a sequential program. We didn't have to use sequential programs for those program-states, and could have used state machines for one or both -- the point is that PSM allows the designer to choose whichever model is most appropriate.

PSM enforces a stricter hierarchy than the HCFSM model used in Statecharts. In Statecharts, transitions may point to or from a substate within a state, such as the transition in Figure 8.6(b) pointing from the substate of the state to the `NormalMode` state. Having this transition start from `FireDrOpen` rather than `FireMode` causes the elevator to always go all the way down to the first floor when the `fire` input becomes true, even if the input is true just momentarily. PSM, on the other hand, only allows transitions between sibling states, i.e., between states with the same parent state. PSM's model of hierarchy is the same as in sequential program languages that use subroutines for hierarchy; namely, we always enter the subroutine from one point, and when we exit the subroutine we do not specify to where we are exiting.

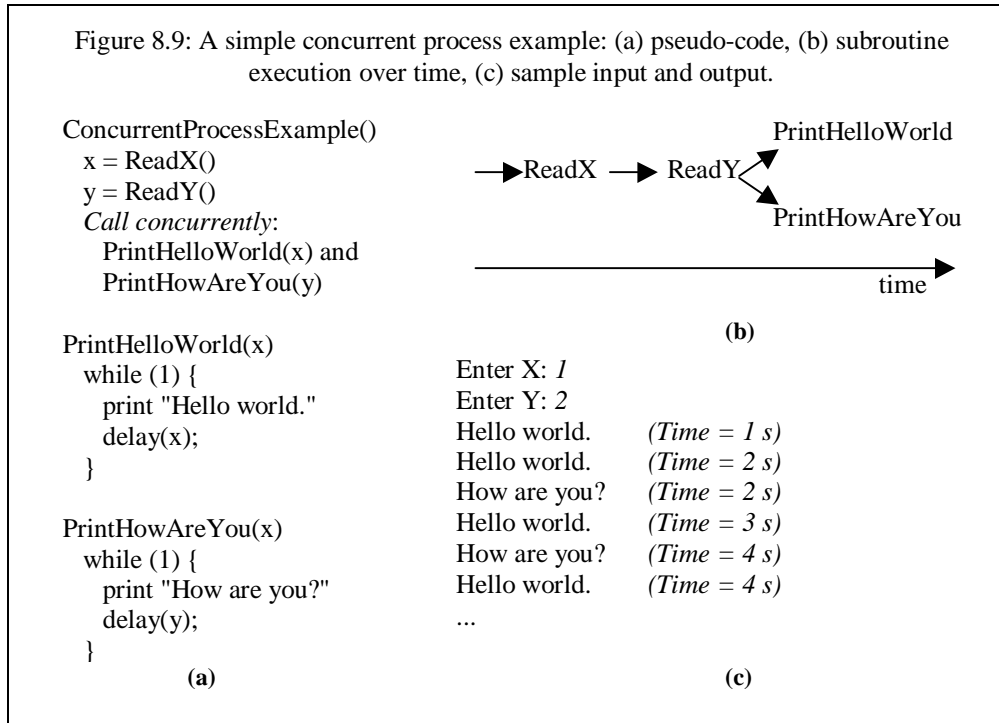


As in the sequential programming model, but unlike the HCFSM model, PSM includes the notion of a program-state *completing*. If the program-state is a sequential program, then reaching the end of the code means the program-state is complete. If the program-state is OR-decomposed into substates, then a special *complete* substate may be added. Transitions may occur from a substate to the complete substate (but no transitions may leave the complete substate), which when entered means that the program-state is complete. Consequently, PSM introduces two types of transitions. A *transition-immediately* (TI) transition is taken immediately if its condition becomes true, regardless of the status of the source program-state -- this is the same as the transition type in an HCFSM. A second, new type of transition, *transition-on-completion* (TOC), is taken only if the condition is true AND the source program-state is complete. Graphically, a TOC transition is drawn originating from a filled square inside a state, rather than from the state's perimeter. We used a TOC transition in Figure 8.8 to transition from FireMode to NormalMode only after FireMode completed, meaning that the elevator had reached the first floor. By supporting both types of transitions, PSM elegantly merges the reactive nature of HCFSM models (using TI transitions) with the transformational nature of sequential program models (using TOC transitions).

#### 8.4 Concurrent process model

In a *concurrent process* model, we describe system behavior as a set of processes, which communicate with one another. A process refers to a repeating sequential program. While many embedded systems are most easily thought of as one process, other systems are more easily thought of as having multiple processes running concurrently.

For example, consider the following made-up system. The system allows a user to provide two numbers X and Y. We then want to write "Hello World" to a display every X seconds, and "How are you" to the display every Y seconds. A very simple way to



describe this system using concurrent processes is shown in Figure 8.9(a). After reading in  $X$  and  $Y$ , we call two subroutines concurrently. One subroutine prints "Hello World" every  $X$  seconds, the other prints "How are you" every  $Y$  seconds. (Note that you can't call two subroutines concurrently in a pure sequential program model, such as the model supported by the basic version of the C language). As shown in Figure 8.9(b), these two subroutines execute simultaneously. Sample output for  $X=1$  and  $Y=2$  is shown in Figure 8.9(c).

To see why concurrent processes were helpful, try describing the same system using a sequential program model (i.e., one process). You'll find yourself exerting effort figuring out how to schedule the two subroutines into one sequential program. Since this example is a trivial one, this extra effort is not a serious problem, but for a complex system, this extra effort can be significant and can detract from the time you have to focus on the desired system behavior.

Recall that we described our elevator controller using two "blocks." Each block is really a process. The controller was simply easier to comprehend if we thought of the two blocks independently.

### 8.4.1 Processes and threads

In operating system terminology, a distinction is made between regular processes and threads. A regular process is a process that has its own virtual address space (stack, data, code) and system resources (e.g., open files). A *thread*, in contrast, is really a sub-process within a process. It is a lightweight process that typically has only a program counter, stack and register; it shares its address space and system resources with other threads. Since threads are small compared to regular processes, they can be created quickly, and switching between threads by an operating system does not incur very heavy costs. Furthermore, threads can share resources and variables so they can communicate quickly and efficiently.

### 8.4.2 Communication

Two concurrent processes communicate using one of two techniques: message passing, or shared data. In the *shared data* technique, processes read and write variables

that both processes can access, called global variables. For example, in the elevator example above, the `RequestResolver` process writes to a variable `req`, which is also read by the `UnitControl` process.

In *message passing*, communication occurs using send and receive constructs that are part of the computation model. Specifically, a process P explicitly sends data to another process Q, which must explicitly receive the data. In the elevator example, `RequestResolver` would include a statement: `Send(UnitControl, rr_req)`. Likewise, `UnitControl` would include statements of the form: `Receive(RequestResolver, uc_req)`. `rr_req` and `uc_req` are variables local to each process.

Message passing may be blocking or non-blocking. In *blocking* message passing, a sending process must wait until the receiving process receives the data before executing the statement following the send. Thus, the processes synchronize at their send/receive points. In fact, a designer may use a send/receive with no actual message being passed, in order to achieve the synchronization. In *non-blocking* message passing, the sending process need not wait for the receive to occur before executing more statements. Therefore, a queue is implied in which the sent data must be stored before being received by the receiving process.

### 8.4.3 Implementing concurrent processes

The most straightforward method for implementing concurrent processes on processors is to implement each process on its own processor. This method is common when each process is to be implemented using a single-purpose processor.

However, we often decide that several processes should be implemented using general-purpose processors. While we could conceptually use one general-purpose processor per process, this would likely be very expensive and in most cases is not necessary. It is not necessary because the processes likely do not require 100% of the processor's processing time; instead, many processes may share a single processor's time and still execute at the necessary rates.

One method for sharing a processor among multiple processes is to *manually rewrite* the processes as a single sequential program. For example, consider our Hello World program from earlier. We could rewrite the concurrent process model as a sequential one by replacing the concurrent running of the `PrintHelloWorld` and `PrintHowAreYou` routines by the following:

```
I = 1;
T = 0;
while (1) {
    Delay(I); T = T + I
    if X modulo T is 0 then call PrintHelloWorld
    if Y modulo T is 0 then call PrintHowAreYou
}
```

We would also modify each routine to have no parameter, no loop and no delay; each would merely print its message. If we wanted to reduce iterations, we could set I to the greatest common divisor of X and Y rather than to 1.

Manually rewriting a model may be practical for simple examples, but extremely difficult for more complex examples. While some automated techniques have evolved to assist with such rewriting of concurrent processes into a sequential program, these techniques are not very commonly used.

Instead, a second, far more common method for sharing a processor among multiple processes is to rely on a multi-tasking *operating system*. An operating system is a low-level program that runs on a processor, responsible for scheduling processes, allocating storage, and interfacing to peripherals, among other things. A real-time operating system (RTOS) is an operating system that allows one to specify constraints on the rate of

processes, and that guarantees that these rate constraints will be met. In such an approach, we would describe our concurrent processes using either a language with processes built-in (such as Ada or Java), or a sequential program language (like C or C++) using a library of routines that extends the language to support concurrent processes. POSIX threads were developed for the latter purpose.

A third method for sharing a processor among multiple processes is to convert the processes to a sequential program that includes a process scheduler right in the code. This method results in less overhead since it does not rely on an operating system, but also yields code that may be harder to maintain.

## 8.5 Other models

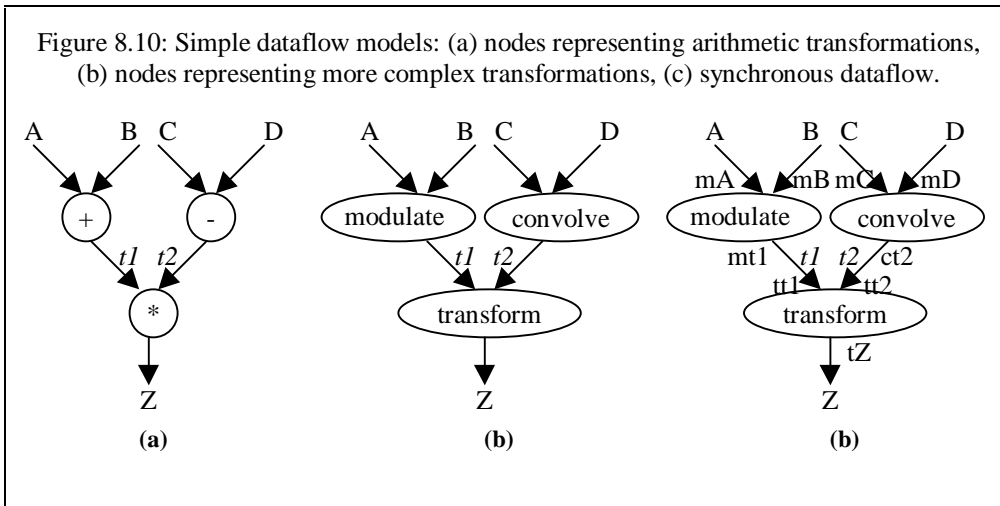
### 8.5.1 Dataflow

In a *dataflow* model, we describe system behavior as a set of nodes representing transformations, and a set of directed edges representing the flow of data from one node to another. Each node consumes data from its input edges, performs its transformation, and produces data on its output edge. All nodes may execute concurrently.

For example, Figure 8.10(a) shows a dataflow model of the computation  $Z = (A+B)*(C-D)$ . Figure 8.10(b) shows another dataflow model having more complex node transformations. Each edge may or not have data. Data present on an edge is called a *token*. When all input edges to a node have at least one token, the node may *fire*. When a node fires, it consumes one token from each input edge, executes its data transformation on the consumed token, and generates a token on its output edge. Note that multiple nodes may fire simultaneously, depending only on the presence of tokens.

Several commercial tools support graphical languages for the capture of dataflow models. These tools can automatically translate the model to a concurrent process model for implementation on a microprocessor. We can translate a dataflow model to a concurrent process model by converting each node to a process, and each edge to a channel. This concurrent process model can be implemented either by using a real-time operating system, or by mapping the concurrent processes to a sequential program.

Lee observed that in many digital signal-processing systems, data flows in to and out of the system at a fixed rate, and that a node may consume and produce many tokens per firing. He therefore created a variation of dataflow called *synchronous dataflow*. In this model, we annotate each input and output edge of a node with the number of tokens that node consumes and produces, respectively, during one firing. The advantage of this model is that, rather than translating to a concurrent process model for implementation, we can instead statically schedule the nodes to produce a sequential program model. This model can be captured in a sequential program language like C, thus running without a real-time operating system and hence executing more efficiently. Much effort has gone into developing algorithms for scheduling the nodes into "single-appearance" schedules, in which the C code only has one statement that calls each node's associated procedure (though this call may be in a loop). Such a schedule allows for procedure inlining, which further improves performance by reducing the overhead of procedure calls, without resulting in an explosion of code size that would have occurred had there been many statements that called each node's procedure.



## 8.6 Summary

Embedded system behavior is becoming increasingly complex, and the sequential program model alone is no longer sufficient for describing this behavior. Additional models can make describing this behavior much easier. The state machine model excels at describing control-dominated systems. Extensions to the basic FSM model include: FSM<sub>D</sub>, which adds complex data types and variables; HCFSM, which adds hierarchy and concurrency; and PSM, which merges the FSM<sub>D</sub> and HCFSM models. The concurrent process model excels at describing behavior with several concurrent threads of execution. The dataflow model excels at describing data-dominated systems. An extension to dataflow is synchronous dataflow, which assumes a fixed rate of data input and output and specifies the number of tokens consumed and produced per node firing, thus allowing for a static scheduling of the nodes and thus an efficient sequential program implementation.

## 8.7 References and further reading

## 8.8 Exercises