UTAustinX: UT.6.01x Embedded Systems - Shape the World

KarenWest (/dashboard) ▼

Courseware (/courses/UTAustinX/UT.6.01x/1T2014/courseware) **Course Info (/courses/UTAustinX/UT.6.01x/1T2014/info)**

Discussion (/courses/UTAustinX/UT.6.01x/1T2014/discussion/forum) **Progress (/courses/UTAustinX/UT.6.01x/1T2014/progress)**

Questions (/courses/UTAustinX/UT.6.01x/1T2014/a3da417940af4ec49a9c02b3eae3460b/)

Syllabus (/courses/UTAustinX/UT.6.01x/1T2014/a827a8b3cc204927b6efaa49580170d1/)

Software abstraction allows us to define a complex problem with a set of basic abstract principles. If we can construct our software system using these abstract building blocks, then we have a better understanding of both the problem and its solution. This is because we can separate what we are doing (policies) from the details of how we are getting it done (mechanisms). This separation also makes it easier to optimize. Abstraction provides for a proof of correct function and simplifies both extensions and customization. The abstraction presented in this section is the **F**inite **S**tate **M**achine (FSM). The abstract principles of FSM development are the inputs, outputs, states, and state transitions. The FSM state graph defines the time-dependent relationship between its inputs and outputs. If we can take a complex problem and map it into a FSM model, then we can solve it with simple FSM software tools. Our FSM software implementation will be easy to understand, debug, and modify. Other examples of software abstraction include **P**roportional **I**ntegral **D**erivative digital controllers, fuzzy logic digital controllers, neural networks, and linear systems of differential equations. In each case, the problem is mapped into a well-defined model with a set of abstract yet powerful rules. Then, the software solution is a matter of implementing the rules of the model. In our case, once we prove our software correctly solves one FSM, then we can make changes to the state graph and be confident that our software solution correctly implements the new FSM.

Help

A Finite State Machine (FSM) is an abstraction that describes the solution to a problem very much like an Algorithm. Unlike an algorithm which gives a sequence of steps that need to be followed to realize the solution to a problem, a FSM describes the system (the solution being a realization of the system's behavior) as a machine that changes states in reaction to inputs and produces appropriate outputs. Many systems in engineering can be described using an FSM. First let's define what are the essential elements that constitute an FSM. A Finite Statement Machine can be described by these five essential elements:

1. A finite set of states that you can find the system in. One of these states has to be identified as the initial state
2. A finite set of external inputs to the system
3. A finite set of external outputs that the system generates
4. An explicit specification of all state transitions. That is, for every state, what happens (as in, which state will the system transition to) when you are in that state and a specific input occurs?
5. An explicit specification of how the outputs are determined. That is, when does a specific output get generated?

A representation of a system's behavior involves describing all five of these essential elements. Elements 4 and 5 are visually described using a State Transition Graph. We can also state 4 and 5 mathematically as follows:

- Element 4: The next state that the system goes into is a function of the input received and the current state. i.e.

NextState = f(Input, CurrentState)

- Element 5: The output that the system generates is a function of only the current state. i.e.,

  Output = g(CurrentState)

A State Transition Graph (STG) has nodes and edges, where the nodes relate to the states of the FSM and the edges represent the transitions from one state to another when a particular input is received. Edges are accordingly labeled with the input that caused the transition. The output can also be captured in the Graph. Note that a FSM where the output is only dependent on the current state and not the input is called a *Moore* FSM. FSMs where the output is dependent on both the current state and the input are called *Mealy* FSMs i.e.,

Output = h(Input, CurrentState)

We note that some systems lend themselves better to a Mealy description while others are more naturally expressed as Moore machines. However, both machine descriptions are equivalent in that any system that can be described using a Mealy machine can also be expressed equivalently as a Moore machine and vice versa. See Figure 10.5.

The FSM controller employs a well-defined model or framework with which we solve our problem. STG will be specified using either a linked or table data structure. An important aspect of this method is to create a 1-1 mapping from the STG into the data structure. The three advantages of this abstraction are 1) it can be faster to develop because many of the building blocks preexist; 2) it is easier to debug (prove correct) because it separates conceptual issues from implementation; and 3) it is easier to change.

When designing a FSM, we begin by defining what constitutes a state. In a simple system like a single intersection traffic light, a state might be defined as the pattern of lights (i.e., which lights are on and which are off). In a more sophisticated traffic controller, what it means to be in a state might also include information about traffic volume at this and other adjacent intersections. The next step is to make a list of the various states in which the system might exist. As in all designs, we add outputs so the system can affect the external environment, and inputs so the system can collect information about its environment or receive commands as needed. The execution of a Moore FSM repeats this sequence over and over

1. Perform output, which depends on the current state

2. Wait a prescribed amount of time (optional)

3. Input

4. Go to next state, which depends on the input and the current state

The execution of a Mealy FSM repeats this sequence over and over

1. Wait a prescribed amount of time (optional)

2. Input

3. Perform output, which depends on the input and the current state

4. Go to next state, which depends on the input and the current state

There are other possible execution sequences. Therefore, it is important to document the sequence before the state graph is drawn. The high-level behavior of the system is defined by the state graph. The states are drawn as circles. Descriptive states names help explain what the machine is doing. Arrows are drawn from one state to another, and labeled with the input value causing that state transition.

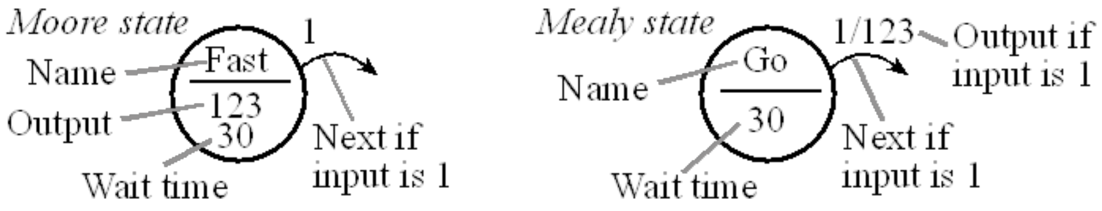

Figure 10.5. The output in a Moore depends just on the state. In a Mealy the output depends on state and input.

Observation: If the machine is such that a specific output value is necessary "to be a state", then a Moore implementation will be more appropriate.

Observation: If the machine is such that no specific output value is necessary "to be a state", but rather the output is required to transition the machine from one state to the next, then a Mealy implementation will be more appropriate.

A linked structure consists of multiple identically-structured nodes. Each node of the linked structure defines one state. One or more of the entries in the node is a link to other nodes. In an embedded system, we usually use statically-allocated fixed-size linked structures, which are defined at compile time and exist throughout the life of the software. In a simple embedded system the state graph is fixed, so we can store the linked data structure in nonvolatile memory. For complex systems where the control functions change dynamically (e.g., the state graph itself varies over time), we could implement dynamically-allocated linked structures, which are constructed at run time and number of nodes can grow and shrink in time. We will use a table structure to define the state graph, which consists of contiguous multiple identically-structured elements. Each element of the table defines one state. One or more of the entries is an index to other elements. The index is essentially a link to another state. An important factor when implementing FSMs is that there should be a clear and one-to-one mapping between the FSM state graph and the data structure. I.e., there should be one element of the structure for each state. If each state has four arrows, then each node of the linked structure should have four links.

The outputs of Moore FSM are only a function of the current state. In contrast, the outputs are a function of both the input and the current state in a Mealy FSM. Often, in a Moore FSM, the specific output pattern defines what it means to be in the current state. In the following videos we take a simplistic system where we wish to detect if a input stream has a odd number of 1s so far. It does not lend itself to an easy implementation because we do not have a notion of how long a bit has to persist for it to be inferred as the same bit as opposed to a new bit. It serves though as a simple example. The later examples are more rigorously specified and therefore lend themselves to proper implementations.