Introduction to Embedded Systems

Notes for EECS 124
Spring 2008

Edward A. Lee, Sanjit A. Seshia, and Claire Tomlin
eal@eecs.berkeley.edu, sseshia@eecs.berkeley.edu, tomlin@eecs.berkeley.edu
Electrical Engineering & Computer Sciences
University of California, Berkeley

February 25, 2008

Chapter 1

Model-Based Design

Models of cyber-physical systems must include both the physical parts and the computing and networking parts. The models will typically need to represent both static and dynamic properties. Static properties are those that do not change during the operation of the system. They include for example logical structure, physical structure, data types, or data ontologies (interpretations of the meaning of the data). Dynamic properties are those that do change during the operation of the system. They include physical dynamics (e.g. the motion of mechanical parts of the system), timing of actions, mode changes, energy consumption, noise, etc.

1.1 Modeling Physical Dynamics

Physical motion of mechanical parts of cyber-physical system can often be modeled using differential equations, or equivalently, integral equations. Such models really only work well for "smooth" motion (a concept that we can make more precise using notions of linearity, time invariance, and continuity). For motions that are not smooth, such as those modeling collisions of mechanical parts, we can use modal models that represent distinct modes of operation with abrupt (conceptually instantaneous) transitions between modes. We can often use differential equations to model dynamics within each mode. The combination of modal models and differential equations is often called **hybrid systems**. We begin with simple equations of motion.

1.1.1 Newtonian Mechanics

In this section, we give a brief working review of some principles of classical mechanics. This is intended to be just enough to be able to construct interesting models, but is by no means comprehensive. The interested reader is referred to many excellent texts on classical mechanics, including [1, 2, 4].

Motion in space of physical objects can be represented with **six degrees of freedom**, illustrated in Figure 1.1. Three of these represent position in three dimensional space, and three represent

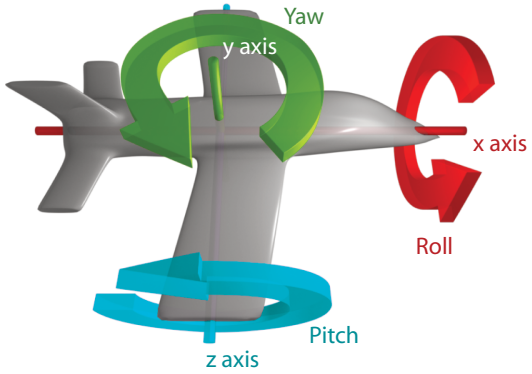


Figure 1.1: Modeling position with six degrees of freedom.

orientation in space. We assume three axes, x, y, and z, where by convention x is drawn increasing to the right, y is drawn increasing upwards, and z is drawn increasing out of the page. Roll θ_x is an angle of rotation around the x axis, where by convention an angle of 0 radians represents horizontally flat along the z axis (i.e., the angle is given relative to the z axis). Yaw θ_y is the rotation around the y axis, where by convention 0 radians represents pointing directly to the right (i.e., the angle given relative to the x axis). Pitch θ_z is rotation around the z axis, where by convention 0 radians represents pointing horizontally (i.e. the angle is given relative to the x axis).

The position of an object in space, therefore, is represented by six functions of the form $f\colon \mathbb{R} \to \mathbb{R}$, where the domain represents time and the codomain represents either distance along an axis or angle relative to an axis. These are often collected into vector-valued functions $\mathbf{x}\colon \mathbb{R} \to \mathbb{R}^3$ and $\theta\colon \mathbb{R} \to \mathbb{R}^3$, where $\mathbf{x}$ represents position, and θ represents orientation.

Changes in position or orientation are governed by Newton's second law, relating force with acceleration. Acceleration is the second derivative of position. Our first equation handles the position information,

$$\mathbf{F}(t) = M\ddot{\mathbf{x}}(t), \tag{1.1}$$

where $\mathbf{F}$ is the force vector in three directions, M is the mass of the object, and $\ddot{\mathbf{x}}$ is the second derivative of $\mathbf{x}$ with respect to time (i.e. the acceleration). Velocity is the integral of acceleration, given by

$$\forall\, t > 0, \quad \dot{\mathbf{x}}(t) \quad = \quad \dot{\mathbf{x}}(0) + \int_0^t \ddot{\mathbf{x}}(\tau)\, d\tau$$

where $\dot{\mathbf{x}}(0)$ is the initial velocity in three directions. Using (1.1), this becomes

$$\forall\, t > 0, \quad \dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \frac{1}{M}\int_0^t \mathbf{F}(\tau)\, d\tau,$$

Position is the integral of velocity,

$$\begin{aligned}\mathbf{x}(t) &= \mathbf{x}(0) + \int_0^t \dot{\mathbf{x}}(\tau)d\tau \\ &= \mathbf{x}(0) + t\dot{\mathbf{x}}(0) + \frac{1}{M}\int_0^t \int_0^\tau \mathbf{F}(\alpha)d\alpha d\tau,\end{aligned}$$

where $\mathbf{x}(0)$ is the initial position. Using these equations, if you know the initial position and initial velocity of an object and the forces on the object in all three directions as a function of time, you can determine the acceleration, velocity, and position of the object at any time.

The versions of these motions of equation that affect orientation use **torque**, the rotational version of force. It is again a three-element vector as a function of time, representing the net twisting force on an object. It can be related to angular velocity in a manner similar to (1.1),

$$\mathbf{T}(t) = \frac{d}{dt}\left(I(t)\dot{\theta}(t)\right), \tag{1.2}$$

where $\mathbf{T}$ is the torque vector in three axes and $I(t)$ is the **moment of inertia tensor** of the object. The moment of inertia is a 3×3 matrix that depends on the geometry and orientation of the object. Intuitively, it represents the reluctance that an object has to spin around any axis as a function of its orientation along the three axes. If the object is spherical, for example, this reluctance is the same around all axes, so it reduces to a constant scalar I. The equation then looks much more like (1.1),

$$\mathbf{T}(t) = I\ddot{\theta}(t). \tag{1.3}$$

To be explicit about the three dimensions, we might write (1.2) as

$$\begin{bmatrix} T_x(t) \\ T_y(t) \\ T_z(t) \end{bmatrix} = \frac{d}{dt}\left(\begin{bmatrix} I_{xx}(t) & I_{xy}(t) & I_{xz}(t) \\ I_{yx}(t) & I_{yy}(t) & I_{yz}(t) \\ I_{zx}(t) & I_{zy}(t) & I_{zz}(t) \end{bmatrix} \begin{bmatrix} \dot{\theta}_x(t) \\ \dot{\theta}_y(t) \\ \dot{\theta}_z(t) \end{bmatrix} \right).$$

Here, for example, $T_y(t)$ is the net torque around the y axis (which would cause changes in yaw), $I_{yx}(t)$ is the inertia that determines how acceleration around the x axis is related to torque around the y axis.

Rotational velocity is the integral of acceleration,

$$\dot{\theta}(t) = \dot{\theta}(0) + \int_0^t \ddot{\theta}(\tau)d\tau,$$

where $\dot{\theta}(0)$ is the initial rotational velocity in three axes. For a spherical object, using (1.3), this becomes

$$\dot{\theta}(t) = \dot{\theta}(0) + \frac{1}{I}\int_0^t \mathbf{T}(\tau)d\tau.$$

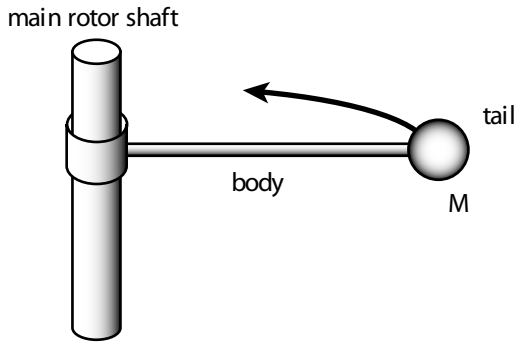


Figure 1.2: Simplified model of a helicopter.

Orientation is the integral of rotational velocity,

$$\begin{aligned}\theta(t) &= \theta(0) + \int_0^t \dot{\theta}(\tau) d\tau \\ &= \theta(0) + t\dot{\theta}(0) + \frac{1}{I}\int_0^t \int_0^\tau \mathbf{T}(\alpha) d\alpha d\tau\end{aligned}$$

where $\theta(0)$ is the initial orientation. Using these equations, if you know the initial orientation and initial rotational velocity of an object and the torques on the object in all three axes as a function of time, you can determine the rotational acceleration, velocity, and orientation of the object at any time.

Often, as we have done for a spherical object, we can simplify by reducing the number of dimensions that are considered. For example, if an object is a moving vehicle on a flat surface, there is little reason to consider the y axis movement or the pitch or roll of the object.

> **Example 1.1:** Consider a simple control problem that admits such reduction of dimensionality. A helicopter has two rotors, one above, which provides lift, and one on the tail. Without the rotor on the tail, the body of the helicopter would start to spin. The rotor on the tail counteracts that spin. Specifically, the force produced by the tail rotor must perfectly counter the friction with the main rotor, or the body will spin. Here we consider this role of the tail rotor independently from all other motion of the helicopter.
>
> A highly simplified version of the helicopter is shown in figure 1.2. In this version, we assume that the helicopter position is fixed at the origin, and hence there is no need to consider the equations governing the dynamics of position. Moreover, we will assume that the helicopter remains vertical, so pitch and roll are fixed at zero.[1]
>
> With these assumptions, the moment of inertia reduces to a scalar that resists changes in yaw. The torque causing changes in yaw will be due to the friction with the main

[1] Note that these assumptions are not as unrealistic as they may seem since we can define the coordinate system to be fixed to the helicopter.

rotor. This will tend to cause the helicopter to rotate in the same direction as the rotor rotation. The tail rotor has the job of countering that torque to keep the body of the helicopter from spinning.

We model the simplified helicopter by a system that takes as input a continuous-time signal T_y, the torque around the y axis (which causes changes in yaw). This torque is the net difference between that caused by the friction of the main rotor and that caused by the tail rotor. The output of our system will be the angular velocity $\dot{\theta}_y$ around the y axis. The dimensionality-reduced version of (1.2) can be written as

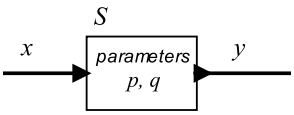
$$\ddot{\theta}_y(t) = T_y(t)/I_{yy}.$$

Integrating both sides, we get the output as a function of the input,

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau. \tag{1.4}$$

1.1.2 Actor Models

In the previous section, a model of a physical system is given by a differential or an integral equation that relates input signals (force or torque) to output signals (position, orientation, velocity, or rotational velocity). Such a physical system can be viewed as a component in a larger model. In particular, a continuous-time system may be represented by a box with an input port and an output port as follows:



where the input signal x and the output signal y are functions of the form

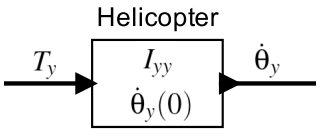
$$x\colon \mathbb{R} \to \mathbb{R}, \quad y\colon \mathbb{R} \to \mathbb{R}.$$

Here the domain represents time and the codomain represents the value of the signal at a particular time. The system itself is a function of the form

$$S\colon X \to Y, \tag{1.5}$$

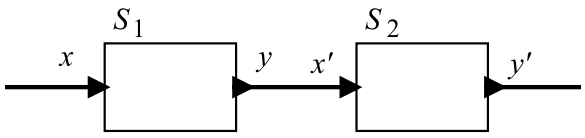
where $X = Y = (\mathbb{R} \to \mathbb{R})$, the set of functions of the form of x and y above. The function S may depend on **parameters** of the system, in which case the parameters may be optionally shown in the box, and may be optionally included in the function notation. For example, in the above figure, if there are parameters p and q, we might write the system function as $S_{p,q}$ or even $S(p,q)$, keeping in mind that both of notations represent functions of the form in 1.5.

Example 1.2: The actor model for the helicopter of example 1.1 can be depicted as follows:



The input and output are both continuous-time functions. The parameters of the actor are the initial angular velocity and the moment of inertia. The function of the actor is defined by (1.4).

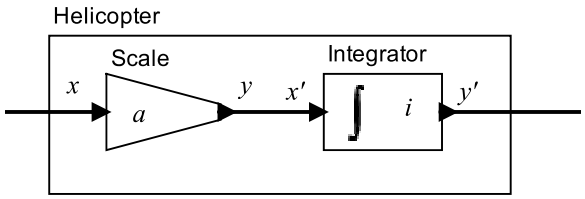
Actor models are composable. In particular, given two systems S_1 and S_2, we can form a **cascade composition** as follows:



In the diagram, the "wire" between the output of S_1 and the input of S_2 means precisely that $y = x'$, or more pedantically,

$$\forall\, t \in \mathbb{R}, \quad y(t) = x'(t).$$

Example 1.3: The actor model for the helicopter can be represented as a cascade composition of two systems as follows:



The left subsystem represents a simple scaling function parameterized by the constant a defined by

$$\forall\, t \in \mathbb{R}, \quad y(t) = ax(t). \tag{1.6}$$

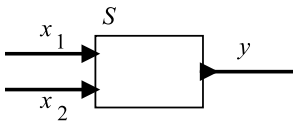
More compactly, we can write $y = ax$, where it is understood that the product of a scalar a and a function x is interpreted as in (1.6). The right system represents an integrator parameterized by the initial value i defined by

$$\forall\, t \in \mathbb{R}, \quad y'(t) = i + \int_0^t x'(\tau)d\tau.$$

If we give the parameter values $a = 1/I_{yy}$ and $i = \dot{\theta}_y(0)$, we see that this system represents (1.4) where the input $x = T_y$ and the output $y' = \dot{\theta}_y$.

In the above figure, we have customized the **icons**, which are the boxes representing the subsystems. These particular subsystems (scaler and integrator) are particularly useful building blocks for building up models of physical dynamics, so assigning them recognizable visual notations is useful.

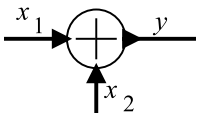
We can have systems that have multiple input signals and/or multiple output signals. These are represented similarly, as in the following example which has two input signals and one output signal:



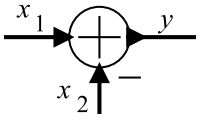
A particularly useful building block with this form is a signal **adder**, defined by

$$\forall\, t \in \mathbb{R}, \quad y(t) = x_1(t) + x_2(t).$$

This will often be represented by a custom icon as follows:



Sometimes, one of the inputs will be subtracted rather than added, in which case the icon is further customized with minus sign near that input, as below:



This (sub)system represents a function $S\colon (\mathbb{R} \to \mathbb{R})^2 \to (\mathbb{R} \to \mathbb{R})$ given by

$$\forall\, t \in \mathbb{R},\ \forall\, x_1, x_2 \in (\mathbb{R} \to \mathbb{R}), \quad (S(x_1, x_2))(t) = y(t) = x_1(t) - x_2(t).$$

1.1.3 Linearity and Time Invariance

Systems that are linear and time invariant (LTI) have particularly nice mathematical properties. Much of the theory of control systems depends on these properties. These properties form the main body of courses on signals and systems, and are beyond the scope of this text. But we will occasionally exploit simple versions of the properties, so it is useful to determine when a system is LTI.

A system $S\colon X \to Y$, where X and Y are sets of signals, is linear if it satisfies the **superposition** property:

$$\forall\, x_1, x_2 \in X \text{ and } \forall\, a, b \in \mathbb{R}, \quad S(ax_1 + bx_2) = aS(x_1) + bS(x_2).$$

It is easy to see that the helicopter system defined in example 1.1 is linear if and only if the initial angular velocity $\dot{\theta}_y(0) = 0$ (see exercise 1).

More generally, it is easy to see that an integrator as defined in example 1.3 is linear if and only if the initial value $i = 0$, that the Scale actor is always linear, and that the cascade of any two linear actors is linear. We can trivially extend the definition of linearity to actors with more than one input or output signal and then determine that the adder is also linear.

To define time invariance, we first define a specialized continuous-time system called a delay. Let $D_\tau \colon X \to Y$, where X and Y are sets of continuous-time signals, be defined by

$$\forall\, x \in X \text{ and } \forall\, t \in \mathbb{R}, \quad (D_\tau(x))(t) = x(t - \tau). \tag{1.7}$$

Here, τ is a parameter of the system. A system $S \colon X \to Y$ is time invariant if

$$\forall\, x \in X \text{ and } \forall\, \tau \in \mathbb{R}, \quad S(D_\tau(x)) = D_\tau(S(x)).$$

The helicopter system defined in example 1.1 is not time-invariant as defined in (1.4). A minor variant, however, is:

$$\dot{\theta}_y(t) = \frac{1}{I_{yy}} \int_{-\infty}^{t} T_y(\tau) d\tau.$$

This version does not allow for an initial angular rotation.

A **linear time-invariant system** (**LTI**) is a system that is both linear and time invariant. A major objective in modeling physical dynamics is to choose an LTI model whenever possible. It is often easy to construct models that are more complicated than they need to be (see exercise 2).

1.1.4 Stability

A system is said to be **bounded-input bounded-output stable** (**BIBO stable** or just **stable**) if the output signal is bounded for all input signals that are bounded.

Consider a continuous-time system with input w and output v. An input is bounded if there is a real number $A < \infty$ such that $|w(t)| \leq A$ for all $t \in \mathbb{R}$. An output is bounded if there is a real number $B < \infty$ such that $|v(t)| \leq B$ for all $t \in \mathbb{R}$. The system is stable if for any input bounded by A, there is some bound B on the output.

> **Example 1.4:** It is now easy to see that the helicopter system developed in example 1.1 is unstable. Let the input be $T_y = u$, where u is the **unit step**, given by
>
> $$\forall\, t \in \mathbb{R}, \quad u(t) = \begin{cases} 0, & t < 0 \\ 1, & t \geq 0 \end{cases}. \tag{1.8}$$
>
> This means that prior to time zero, there is no torque applied to the system, and starting at time zero, we apply a torque of unit magnitude. This input is clearly bounded. It never exceeds one in magnitude. However, the output grows without bound.

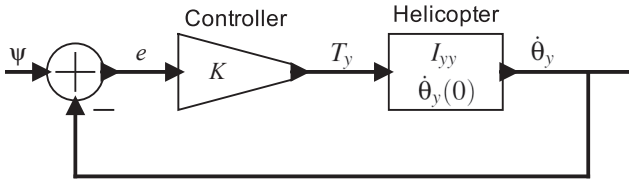


Figure 1.3: Proportional control system that stabilizes the helicopter.

In practice, a helicopter uses a feedback system to determine how much torque to apply at the tail rotor to keep the body of the helicopter straight. We study how to do that next.

1.1.5 Feedback Control

Feedback control is a sophisticated topic, easily occupying multiple texts and complete courses. Here, we only barely touch on the subject, just enough to motivate the interactions between software and physical systems. Feedback control systems are often implemented using embedded software, and the overall physical dynamics is a composition of the software and physical dynamics. More detail can be found in chapters 12-14 of Lee and Varaiya [3].

Example 1.5: Recall that the helicopter model of example 1.1 is not stable. We can stabilize with a simple feedback control system, as shown in figure 1.3. The input ψ to this system is a continuous-time system specifying the desired angular velocity. The **error signal** e represents the difference between the actual and the desired angular velocity. If **controller** simply scales the error signal by a constant K, providing a control input to the helicopter. We use (1.4) to write

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau \tag{1.9}$$

$$= \dot{\theta}_y(0) + \frac{K}{I_{yy}} \int_0^t (\psi(\tau) - \dot{\theta}_y(\tau)) d\tau, \tag{1.10}$$

where we have used the facts (from the figure),

$$e(t) = \psi(t) - \dot{\theta}_y(t)$$

and

$$T_y(t) = Ke(t).$$

Equation (1.10) has $\dot{\theta}_y(t)$ on both sides, and therefore is not trivial to solve. The easiest solution technique uses Laplace transforms (see [3] chapter 14). However, for our purposes here, we can use a more brute-force technique from calculus. To make this as simple as possible, we assume that $\psi(t) = 0$ for all t; i.e., we wish to control the

helicopter simply to keep it from rotating at all. The desired angular velocity is zero. In this case, (1.10) simplifies to

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) - \frac{K}{I_{yy}} \int_0^t \dot{\theta}_y(\tau) d\tau. \tag{1.11}$$

Using the fact from calculus that

$$\int_0^t a e^{a\tau} d\tau = 1 - e^{at} u(t),$$

where u is given by (1.8), we can infer that the solution to (1.11) is

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) e^{-Kt/I_{yy}} u(t). \tag{1.12}$$

(Note that although it is easy to verify that this solution is correct, deriving the solution is not so easy. For this purpose, Laplace transforms provide a far better mechanism.)

We can see from (1.12) that the angular velocity approaches the desired angular velocity (zero) as t gets large as long as K is positive. For larger K, it will approach more quickly. For negative K, the system is unstable, and angular velocity will grow without bound.

In the previous example, we derived the solution to a **proportional control** feedback loop. It is called this because the control signal is proportional to the error. We assumed a desired signal of zero. It is equally easy to assume that the helicopter is initially at rest (the angular velocity is zero) and then determine the behavior for a particular non-zero desired signal, as we do in the following example.

Example 1.6: Assume that helicopter is **initially at rest**,

$$\dot{\theta}(0) = 0,$$

and that the desired signal is

$$\psi(t) = au(t)$$

for some constant a. That is, we wish to control the helicopter to get it to rotate at a fixed rate.

We use (1.4) to write

$$\begin{aligned}
\dot{\theta}_y(t) &= \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau \\
&= \frac{K}{I_{yy}} \int_0^t (\psi(\tau) - \dot{\theta}_y(\tau)) d\tau \\
&= \frac{K}{I_{yy}} \int_0^t a d\tau - \frac{K}{I_{yy}} \int_0^t \dot{\theta}_y(\tau) d\tau \\
&= \frac{Kat}{I_{yy}} - \frac{K}{I_{yy}} \int_0^t \dot{\theta}_y(\tau) d\tau.
\end{aligned}$$

Using the same (black magic) technique of inferring and then verifying the solution, we can see that the solution is

$$\dot{\theta}_y(t) = au(t)(1 - e^{-Kt/I_{yy}}). \tag{1.13}$$

Again, the angular velocity approaches the desired angular velocity as t gets large as long as K is positive. For larger K, it will approach more quickly. For negative K, the system is unstable, and angular velocity will grow without bound.

Note that the first term in the above solution is exactly the desired angular velocity. The second term is an error called the **tracking error**, that for this example asymptotically approaches zero.

The above example is somewhat unrealistic because we cannot independently control the *net* torque of the helicopter. In particular, the net torque T_y is the sum of the torque T_t due to the friction of the top rotor and the torque T_r due to the tail rotor,

$$\forall\, t \in \mathbb{R}, \quad T_y(t) = T_t(t) + T_r(t)\ .$$

T_t will be determined by the rotation required to maintain or achieve a desired altitude, quite independent of the rotation of the helicopter. Thus, we will actually need to design a control system that controls T_r and stabilizes the helicopter for any T_t (or, more precisely, any T_t within operating parameters). In the next example, we study how this changes the performance of the control system.

Example 1.7: In Figure 1.4(a), we have modified the helicopter model so that it has two inputs, T_t and T_r, the torque due to the top rotor and tail rotor respectively. The feedback control system is now controlling only T_r, and T_t is treated as an external (uncontrolled) input signal. How well will this control system behave?

Again, a full treatment of the subject is beyond the scope of this text, but we will study a specific example. Suppose that the torque due to the top rotor is given by

$$T_t = bu(t)$$

for some constant b. That is, at time zero, the top rotor starts spinning a constant velocity, and then holds that velocity. Suppose further that the helicopter is initially at rest. We can use the results of Example 1.6 to find the behavior of the system.

First, we transform the model into the equivalent model shown in Figure 1.4(b). This transformation simply relies on the algebraic fact that for any real numbers a_1, a_2, K,

$$Ka_1 + a_2 = K(a_1 + a_2/K).$$

We further transform the model to get the equivalent model shown in Figure 1.4(c), which has used the fact that addition is commutative. In Figure 1.4(c), we see that the portion of the model enclosed in the box is exactly the same as the control system analyzed in Example 1.6, shown in Figure 1.3. Thus, the same analysis as in Example 1.6 still applies. Suppose that desired angular rotation is

$$\psi(t) = 0.$$

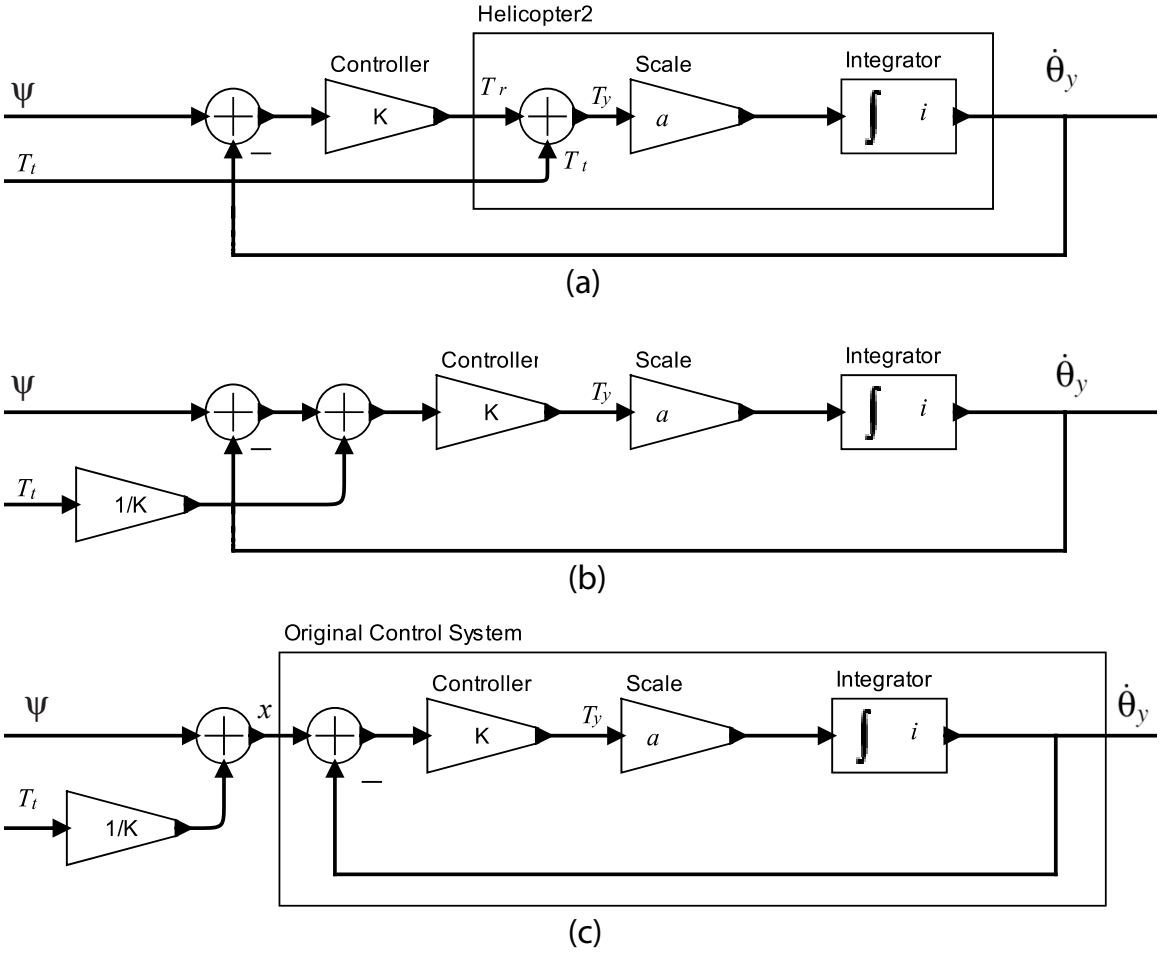


Figure 1.4: (a) Helicopter model with separately controlled torques for the top and tail rotors. (b) Tranformation to an equivalent model. (c) Further transformation to an equivalent model that we can use to understand the behavior of the controller.

Then the input to the original control system will be

$$x(t) = \psi(t) + T_t(t)/K = (b/K)u(t).$$

From (1.13), we see that the solution is

$$\dot{\theta}_y(t) = (b/K)u(t)(1 - e^{-Kt/I_{yy}}). \tag{1.14}$$

The desired angular rotation is zero, but the control system asymptotically approaches a non-zero angular rotation of b/K. This tracking error can be made arbitrarily small by increasing the control system feedback gain K, but with this controller design, it cannot be made to go to zero. An alternative controller design that yields an asymptotic tracking error of zero is studied in Exercise 4.

1.2 Modeling Modal Behavior

Models of embedded systems include both discrete and continuous components. The process of modeling discrete components typically involves specifying a finite set of *modes*, along with the switching conditions that cause the system to transition from **mode** to mode. As discussed earlier in this chapter, the physical dynamics of the system in each mode are modeled with differential or integral equations. This approach to modeling systems is termed **modal modeling**.

In general, the **state** of a system comprises both the system mode as well as the values of internal system parameters that determine its dynamics. We will focus, in this section, on *purely* modal modeling, where the physical dynamics is hidden. The resulting formalism is called a *finite-state machine*. In this setting, the terms *state* and *mode* can be used interchangeably.

1.2.1 Finite-State Machines

A state machine maps input signals to output signals based on its current state. The vector of input signals is assumed to take values from a finite domain *Inputs*; for output signals, the corresponding domain is denoted by *Outputs*. Elements of the sets *Inputs* and *Outputs* are termed **symbols**. For example, if the system has n input signals and m output signals, each of which maps $\mathbb{N}$ to $\mathbb{B} = \{0, 1\}$, then $Inputs = \mathbb{B}^n$ and $Outputs = \mathbb{B}^m$.

Before we introduce the formal definition of a finite-state machine, it is useful to further discuss the forms of input and output symbols relevant to this section.

Symbols and Pure Signals

In this section, we will restrict our attention to so-called **pure signals**. Informally speaking, a pure signal models the presence or absence of a corresponding **event** of interest at a particular time point. Formally, a pure signal is a function mapping $\mathbb{N}$ to the set $\{present, absent\}$.

Suppose the system has more than one pure input signal. An element in *Inputs* is then a vector indicating which of the signals have their corresponding events present. For instance, if x_1 and x_2 are two pure input signals, the input symbol at any time point can be one of $(present, present)$, $(present, absent)$, $(absent, present)$, $(absent, absent)$. A similar approach can be used for representing output symbols for pure output signals.

Often, rather than writing a long tuple with entries *present* and *absent*, it is easier to write a set comprising the names of only those signals that map the current time step to *present*. Thus, if x_1, x_2, x_3 are input signals and $x_1(n) = x_3(n) = present$, then the input symbol at step n is written as the set $\{x_1, x_3\}$. Thus, the set of input symbols *Inputs* can be viewed as $2^{InputSignals}$ where *InputSignals* denotes the set of names of input signals. Similarly, *Outputs* can be viewed as $2^{OutputSignals}$ where *OutputSignals* denotes the set of names of output signals. [2]

It is also convenient to introduce a shorthand to represent groups of symbols. Let $x_1, x_2, \ldots, x_n$ be pure input signals. Suppose that, for some $t \in \mathbb{N}$, we want to represent a set of input symbols that correspond to $x_{i_1}(t) = x_{i_2}(t) = \ldots = x_{i_k}(t) = present$, with all other input signals free to take any value at time t. We will represent this set by the expression $x_{i_1} \wedge x_{i_2} \wedge \ldots \wedge x_{i_k}$, where $\wedge$ is the *logical AND operator*.

The case when no signal has a corresponding event present is a special one and will be handled later in this section.

An Example

A finite-state machine (FSM) can be visualized as a directed graph, where nodes correspond to states and edges correspond to *transitions* that move the system from state to state. Figure 1.5 gives an example of a finite-state machine. This FSM models the controller for a traffic light in a mostly-pedestrian zone, so that the traffic light turns green only when a sensor embedded in the pavement of the road detects a vehicle at the intersection. The FSM has three states, `red`, `yellow`, and `green`, corresponding to the three possible colors of the traffic light. There are four pure input signals to the controller: *isCar*, which models the presence of a car on the pavement, and *timeR*, *timeG*, and *timeY*, which model the expiration of timers monitoring time spent in the `red`, `green`, and `yellow` states respectively. Finally, there are three output signals: *sigR*, *sigG*, and *sigY* indicating the command to the device lighting the red, green, and yellow lights at the intersection. Notice, in particular, how we model the set of input symbols causing the transition from `red` to `green`. Notice also the edges marked *else*, specifying what the FSM does in the default case in each state, on all other input symbols.

Formal Definition

Formally, a **finite-state machine** is a five-tuple $(States, Inputs, Outputs, update, initialState)$ where

States is a finite set of states;

[2] Note that this set notation conflicts with notation used in Chapter 3 of Lee and Varaiya [3], where a similar set notation is used for grouping transitions on different input symbols.

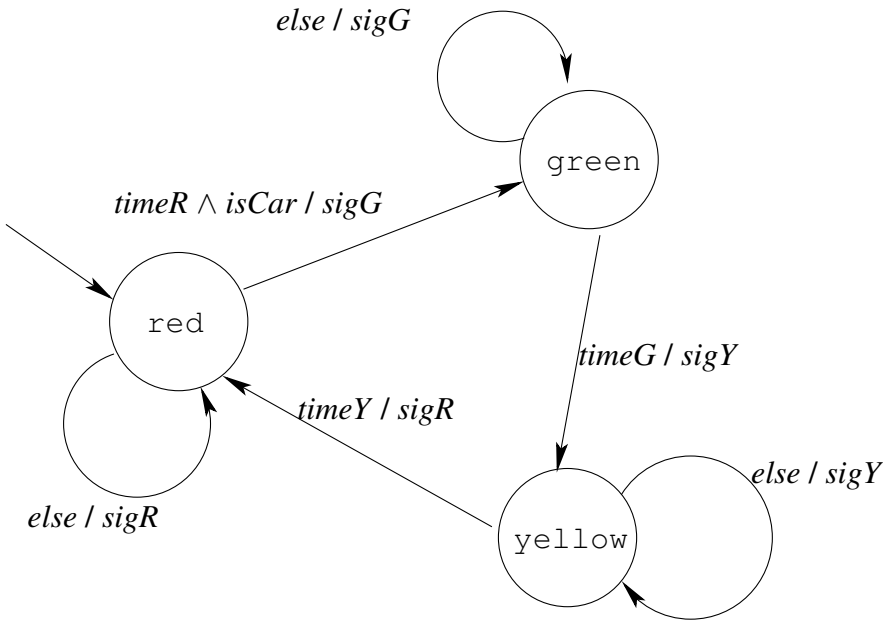


Figure 1.5: Finite-state machine modeling a traffic light controller.

Inputs is a finite set of **input symbols**;

Outputs is a finite set of **output symbols**;

update : $States \times Inputs \rightarrow States \times Outputs$ is an **update function**, mapping a state and an input symbol to a *next* state and an output symbol;

initialState is the **initial state**.

Thus, if $s(n) \in States$ and $x(n) \in Inputs$ are respectively the state and input at step n, then the **next state** $s(n+1)$ and output $y(n)$ at step n are given by

$$(s(n+1), y(n)) = update(s(n), x(n)) \tag{1.15}$$

The change in state and output according to the update function at a step is called a **transition** or a **reaction**. The term **transition function** is often used in place of *update function*.

Example 1.8: The FSM in Figure 1.5 can be formally represented as follows:

$$\begin{aligned} States &= \{\texttt{red}, \texttt{yellow}, \texttt{green}\} \\ Inputs &= 2^{\{timeR, timeY, timeG, isCar\}} \\ Outputs &= 2^{\{sigR, sigY, sigG\}} \\ initialState &= \texttt{red} \end{aligned}$$

Notice how the sets *Inputs* and *Outputs* are defined as power sets.

For the update function, we give only a few representative mappings:

$$\begin{aligned} \mathit{update}(\texttt{red}, \{\mathit{timeR}, \mathit{isCar}\}) &= (\texttt{green}, \{\mathit{sigG}\}) \\ \mathit{update}(\texttt{red}, \{\mathit{timeR}\}) &= (\texttt{red}, \{\mathit{sigR}\}) \end{aligned}$$

The latter update corresponds to the edge labeled with *else* out of state `red`.

To reduce clutter while representing an FSM, we often combine transitions from a state that map different input symbols to the same state and output symbol into a single transition annotated with a *guard*. Formally, a **guard** is a subset of *Inputs* that specifies all conditions under which a given state can be entered while generating a given output symbol.

The example state machine presented in this section has two important properties:

Determinacy: A state machine is said to be **deterministic** if, for each state, there is at most one transition possible on each input symbol.

Clearly, the definition of a finite-state machine given earlier ensures that it is deterministic, since *update* is a function, not a one-to-many mapping.

If we use guards on transitions, the state machine is deterministic if guards leaving each state are non-overlapping.

Receptiveness: A state machine is said to be **receptive** if, for each state, there is at least one transition possible on each input symbol.

In other words, receptiveness ensures that a state machine is always ready to react to any input, and does not "get stuck" in any state.

It follows that if a state machine is both deterministic and receptive, for every state, there is *exactly* one transition possible on each input symbol.

Stuttering

In our definition of an FSM, we introduced the set *Inputs* as set of abstract values that system inputs can take. This abstraction is is a powerful modeling tool, since it allows us to model cases where the system reacts to *events* in environment rather than specific values of environment parameters.

For example, consider modeling an edge-triggered flip-flop in a digital circuit. Suppose the flip-flop only reacts to the rising edge of the clock. We can model this behavior by defining an input symbol `clk_rise` that represents the occurrence of a rising edge of the clock.

However, such modeling raises the question: what is the input to the flip-flop at a time point when there is a falling edge or no change in the clock level?

In order to model the above scenario, it is useful to include a special symbol *absent* in the set *Inputs*. This technique is also useful to model the absence of an output. The symbol *absent* is called a **stuttering symbol**. We will assume that $\mathit{absent} \in \mathit{Inputs}$ and $\mathit{absent} \in \mathit{Outputs}$.

For the case where we have only *pure* signals, the symbol *absent* at a step indicates that *all* pure signals evaluate to *absent* at that step.

The introduction of the stuttering symbol *absent* requires us to define how the update function reacts to *absent*. If the input is absent, it is reasonable to assume that the system state remains unchanged. The rule governing change in output, however, depends on what is best for the system being modeled.

One convention, followed in Chapter 3 of Lee and Varaiya [3], is to require that an absent input symbol generates an absent output symbol, as given below:

$$update(s, absent) = (s, absent)$$

However, if the ouput is a function of the current state alone, it might also be reasonable to allow an output symbol other than *absent*. For instance, if the output $y = \rho(s)$, where s is the current state, then we can define

$$update(s, absent) = (s, \rho(s))$$

1.2.2 Non-Determinism

In order to generate compact modal models, it is necessary to hide inessential details. For example, consider modeling the environment of the traffic light controller given in Figure 1.5 that generates the input *isCar*. Rather than model the details of how cars arrive at traffic lights (say according to some random process), we can simply create a two-state finite-state machine model of the environment, as given in Figure 1.6.

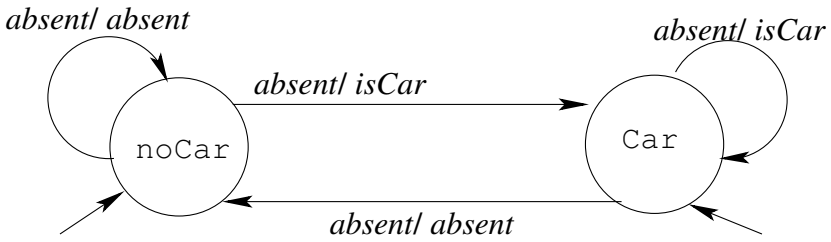


Figure 1.6: Finite-state machine model of cars at a traffic light.

Notice that for each of the two states, the guards leaving those states overlap completely! In other words, the FSM in Figure 1.6 is *non-deterministic*.

Formally, a **non-deterministic** finite-state machine (NDFSM) is also represented as a five-tuple $(States, Inputs, Outputs, possibleUpdates, initialStates)$ where

States is a finite set of states;

Inputs is a finite set of input symbols;

Outputs is a finite set of output symbols;

$possibleUpdates : States \times Inputs \rightarrow 2^{States \times Outputs}$ is an **update relation**, mapping a state and an input symbol to a *set of possible* (next state, output symbol) pairs;

initialStates is a set of initial states.

Note that there are two changes from the definition of a deterministic FSM that we encountered earlier. First, there can be more than one (next state, output) possible from a given state on a given input. This is reflected in the function *possibleUpdates*, whose co-domain is the power set of $States \times Outputs$. We refer to the *possibleUpdates* function as an *update relation*, to emphasize this difference. The term **transition relation** is also often used in place of *update relation*. The second difference is that there can be more than one initial state for the NDFSM. This feature can be used to model uncertainty or hide unnecessary detail in the initialization of the finite-state system.

Example 1.9: The FSM in Figure 1.6 can be formally represented as follows:

$$\begin{aligned} States &= \{\texttt{noCar}, \texttt{Car}\} \\ Inputs &= \{absent\} \\ Outputs &= 2^{\{isCar\}} \\ initialStates &= \{\texttt{noCar}, \texttt{Car}\} \end{aligned}$$

The update relation is given below:

$$\begin{aligned} possibleUpdates(\texttt{noCar}, absent) &= \{(\texttt{noCar}, absent), (\texttt{Car}, \{isCar\})\} \\ possibleUpdates(\texttt{Car}, absent) &= \{(\texttt{noCar}, absent), (\texttt{Car}, \{isCar\})\} \end{aligned}$$

Uses of Non-Determinism

While non-determinism is an interesting mathematical concept in itself, it has two major uses in modeling embedded systems:

Environment Modeling: It is often useful to hide irrelevant detail about how an environment operates, resulting in a non-deterministic FSM model. We have already seen one example of such environment modeling in Figure 1.6.

Specifications: System specifications typically only impose strict requirements on some system features, while leaving others unconstrained. Non-determinism is a useful modeling technique in such settings as well.

For example, consider a specification that the traffic light cycles through `red`, `green`, `yellow`, in that order, without regard to the timing between the outputs. The non-deterministic FSM in Figure 1.7 models this specification: The guard *true* on each transition indicates that the transition can be taken at any step; technically it means that there are transitions corresponding to each edge for all elements in *Inputs*.

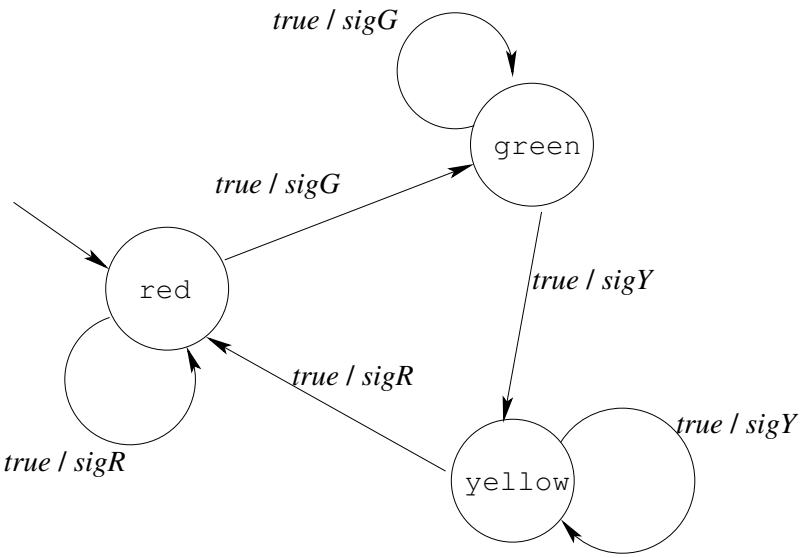


Figure 1.7: Non-deterministic FSM specifying order of lighting signals

1.2.3 Behaviors and Traces

A deterministic finite-state machine defines a function F_d from input sequences to output sequences, as given below:

$$F_d : (\mathbb{N} \to \mathit{Inputs}) \to (\mathbb{N} \to \mathit{Outputs})$$

A **behavior** of a deterministic FSM is a pair (x, y) such that $y = F_d(x)$.

A non-deterministic FSM, on the other hand, can map a single input sequence to many possible output sequences:

$$F_n : (\mathbb{N} \to \mathit{Inputs}) \to 2^{(\mathbb{N} \to \mathit{Outputs})}$$

In this case, a behavior is a pair (x, y) such that $y \in F_n(x)$.

The set of all *behaviors* of a finite-state machine, deterministic or otherwise, is a subset of $(\mathbb{N} \to \mathit{Inputs}) \times (\mathbb{N} \to \mathit{Outputs})$.

Software tools for modeling and analyzing embedded systems often also need to represent and reason about system state. For this purpose, it is useful to include the state as part of system "behavior," generating a system *trace*.

Formally, for a deterministic FSM, a **trace** is a sequence

$$(x_0, s_0, y_0), (x_1, s_1, y_1), (x_2, s_2, y_2), \ldots$$

such that $s_0 = \mathit{initialState}$ and for all $i \geq 0$, $(s_{i+1}, y_i) = \mathit{update}(s_i, x_i)$.

For a non-deterministic FSM, the definition remains the same except that $s_0 \in \mathit{initialStates}$ and for all $i \geq 0$, $(s_{i+1}, y_i) \in \mathit{possibleUpdates}(s_i, x_i)$.

Thus, notice that a non-deterministic FSM can have many possible behaviors and traces for the same input sequence.

We illustrate the concepts of this section with two examples.

Example 1.10: Consider the deterministic FSM in Figure 1.5. The output sequence of this FSM on the following input sequence

$$\{timeR\}, \{timeR, isCar\}, absent, \{timeG\}, \{timeY\}, absent, \ldots$$

is

$$\{sigR\}, \{sigG\}, \{sigG\}, \{sigY\}, \{sigR\}, \{sigR\}, \ldots$$

with corresponding trace:

$$(\{timeR\}, \texttt{red}, \{sigR\}), (\{timeR, isCar\}, \texttt{red}, \{sigG\}), (absent, \texttt{green}, \{sigG\}),$$
$$(\{timeG\}, \texttt{green}, \{sigY\}), (\{timeY\}, \texttt{yellow}, \{sigR\}), (absent, \texttt{red}, \{sigR\}), \ldots$$

The example below considers a non-deterministic FSM.

Example 1.11: Consider the non-deterministic FSM in Figure 1.7. Consider the same input sequence as in Example 1.10:

$$\{timeR\}, \{timeR, isCar\}, absent, \{timeG\}, \{timeY\}, absent, \ldots$$

One possible output sequence is

$$\{sigR\}, \{sigG\}, \{sigG\}, \{sigY\}, \{sigR\}, \{sigR\}, \ldots$$

which is the same as the one in Example 1.10. However, the following output sequence is also possible on the very same input sequence:

$$\{sigR\}, \{sigR\}, \{sigR\}, \{sigG\}, \{sigG\}, \{sigG\}, \ldots$$

In general, we can visualize the set of traces of a non-deterministic FSM on a single input sequence as a **computation tree**, each of whose paths corresponds to a trace of the FSM on the input sequence. Figure 1.8 shows the computation tree for the FSM of Example 1.7 for the first three inputs in the above input sequence. Nodes in the tree are states and edges are labeled by outputs generated while transiting from state to state.

1.3 Modeling Hybrid Systems

Introductory material on hybrid systems will appear here.

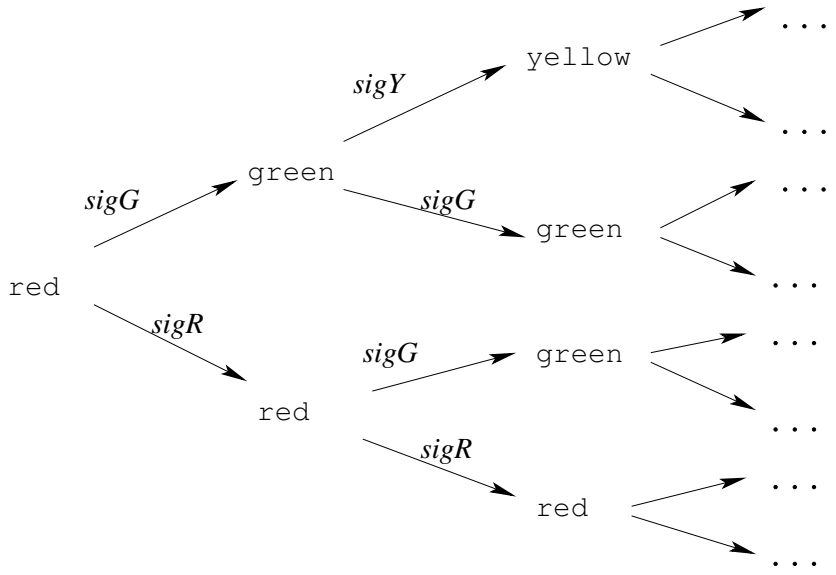


Figure 1.8: A Computation Tree for the FSM in Figure 1.7.

Exercises

1. This exercise studies linearity.

   (a) Show that the helicopter system defined in example 1.1 is linear if and only if the initial angular velocity $\dot{\theta}_y(0) = 0$.

   (b) Show that the cascade of any two linear actors is linear.

   (c) Augment the definition of linearity so that it applies to actors with two input signals and one output signal. Show that the adder actor is linear.

2. Consider the helicopter of example 1.1, but with a slightly different definition of the input and output. Suppose that, as in the example, the input is $T_y : \mathbb{R} \to \mathbb{R}$, as in the example, but the output is the position of the tail relative to the main rotor shaft. Is this an LTI system? Is this a BIBO stable system?

3. Consider a rotating robot where you can control the angular velocity around a fixed axis.

   (a) Model this as a system where the input is angular velocity $\dot{\theta}$ and the output is angle θ. Give your model as an equation relating the input and output as functions of time.

   (b) Is this system BIBO stable?

   (c) Design a proportional controller to set the robot onto a desired angle. That is, assume that the initial angle is $\theta(0) = 0$, and let the desired angle be $\psi(t) = au(t)$. Find the actual angle as a function of time and the proportional controller feedback gain K. What is your output at $t = 0$? What does it approach as t gets large?

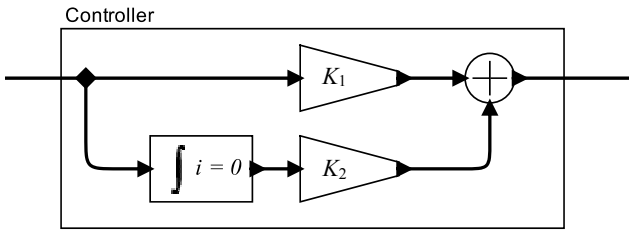


Figure 1.9: A PI controller for the helicopter.

4. (a) Using your favorite continuous-time modeling software (LabVIEW, Simulink, or Ptolemy II, for example), construct a model of the helicopter control system shown in Figure 1.4. Choose some reasonable parameters and plot the actual angular velocity as a function of time, assuming that the desired angular velocity is zero, $\psi(t) = 0$, and that the top-rotor torque is non-zero, $T_t(t) = bu(t)$. Give your plot for several values of K and discuss how the behavior varies with K.

   (b) Modify the model of part (a) to replace the Controller of Figure 1.4 (the simple scale-by-K actor) with the alternative controller shown in figure 1.9. This alternative controller is called a **proportional-integrator (PI) controller**. It has two parameter K_1 and K_2. Experiment with the values of these parameters, give some plots of the behavior with the same inputs as in part (a), and discuss the behavior of this controller in contrast to the one of part (a).

5. Recall the traffic light controller of Figure 1.5. Consider connecting the outputs of this controller to a pedestrian light controller, whose FSM is given below:

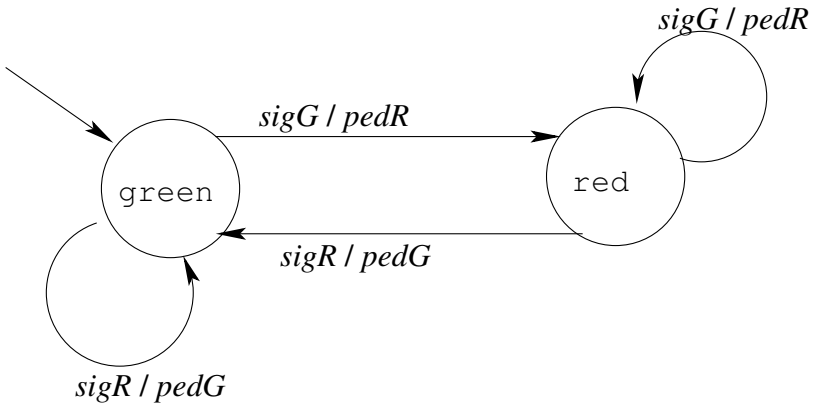


Figure 1.10: Finite-state machine modeling a pedestrian light controller.

Construct a LabVIEW Statecharts model for the composition of the above two FSMs along with a deterministic FSM modeling the environment and generating input symbols *timeR*, *timeG*, *timeY*, and *isCar*. (For example, the environment FSM can use an internal counter to

decide when to generate these symbols.) Use the LabVIEW "region" feature to construct the AND-state corresponding to the composition of these three FSMs.

Run the LabVIEW model and plot along the same time axis the signals *sigR*, *sigY*, *sigG*, *pedG*, *pedR* for one cycle of operation. Hand in printouts of your entire Statecharts model and the plots.

6. Solve Problem 5 at the back of Chapter 6 in Lee & Varaiya [3].

7. Consider Figure 1.11 depicting a system comprising two tanks containing water. Each tank is leaking at a constant rate. Water is added at a constant rate to the system through a hose, which at any point in time is filling either one tank or the other. It is assumed that the hose can switch between the tanks instantaneously. For $i \in \{1,2\}$, let x_i denote the volume of water in Tank i and $v_i > 0$ denote the constant flow of water out of Tank i. Let w denote the constant flow of water into the system. The objective is to keep the water volumes above r_1 and r_2, respectively, assuming that the water volumes are above r_1 and r_2 initially. This is to be achieved by a controller that switches the inflow to Tank 1 whenever $x_1 \leq r_1$ and to Tank 2 whenever $x_2 \leq r_2$.

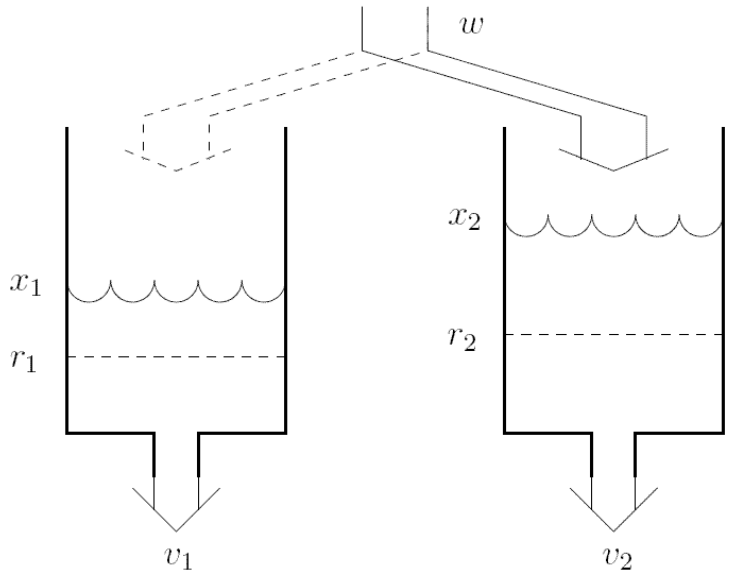


Figure 1.11: Water tank system.

The hybrid automaton representing this two-tank system is given in Figure 1.12.

Answer the following questions:

(a) Construct a model of this hybrid automaton in Ptolemy II using the HyVisual tool. Use the following parameter values: $r_1 = r_2 = 0$, $v_1 = v_2 = 0.5$, and $w = 0.75$. Set the initial state to be $(q_1,(0,1))$. (That is, initial value of x_1 is 0 and that of x_2 is 1.)

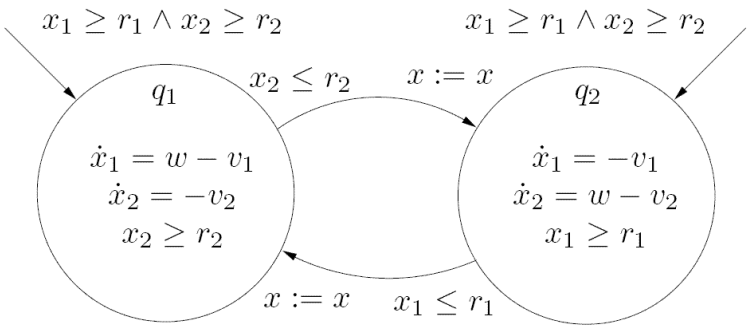


Figure 1.12: Hybrid Automaton representing water tank system.

Verify that this hybrid automaton is Zeno. What is the reason for this Zeno behavior? Simulate your model in Ptolemy II and plot how x_1 and x_2 vary as a function of time t, simulating long enough to illustrate the Zeno behavior.

(b) Use regularization to make your model non-Zeno. Again, plot x_1 and x_2 for the same length of time as in the first part. State the value of ε that you used.

Include printouts of your plots with your answer.