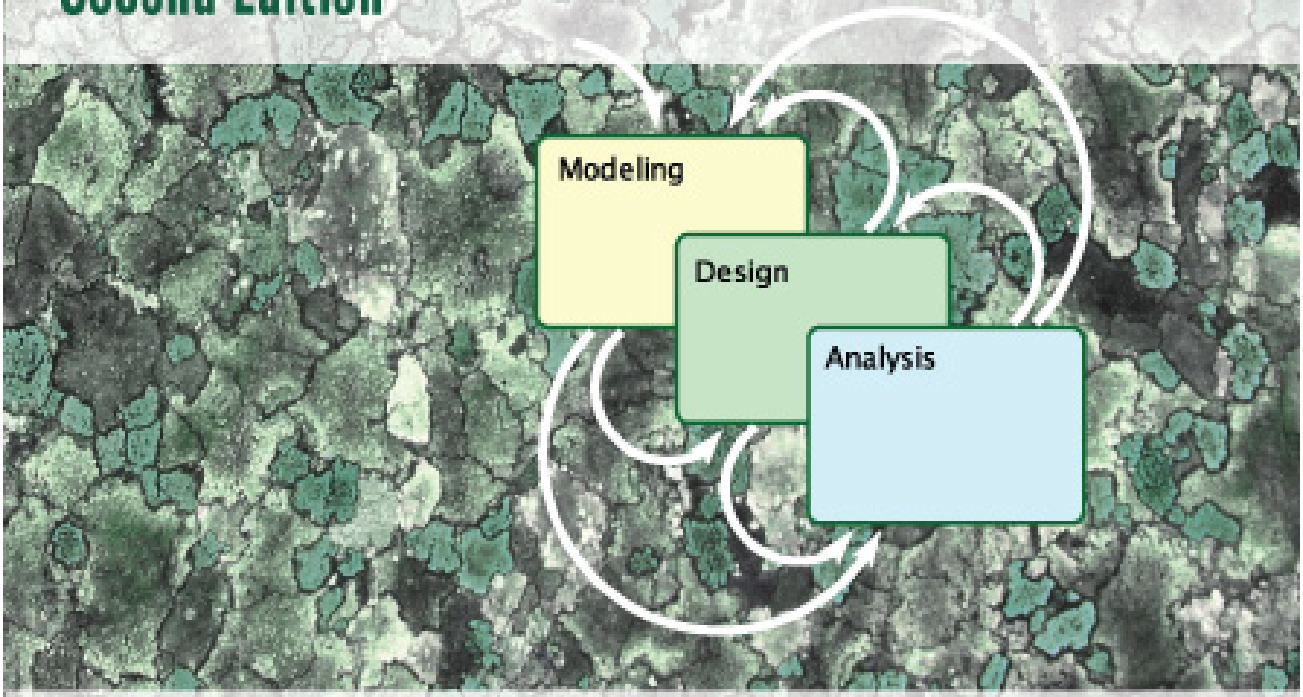
Introduction to Embedded Systems

A Cyber-Physical Systems Approach

Second Edition



Edward Ashford Lee
Sanjit Arunkumar Seshia

To go with Edition 2.0, Printing 2.0

December 17, 2016

Contents

1 Introduction

Chapter 1 has no exercises.

Part I

Modeling Dynamic Behaviors

2 Continuous Dynamics — Exercises

1. A **tuning fork**, shown in Figure 2.1, consists of a metal finger (called a **tine**) that is displaced by striking it with a hammer. After being displaced, it vibrates. If the tine has no friction, it will vibrate forever. We can denote the displacement of the tine after being struck at time zero as a function $y\colon \mathbb{R}_+ \to \mathbb{R}$. If we assume that the initial displacement introduced by the hammer is one unit, then using our knowledge of physics we can determine that for all $t \in \mathbb{R}_+$, the displacement satisfies the differential equation

$$\ddot{y}(t) = -\omega_0^2 y(t)$$

where ω_0^2 is a constant that depends on the mass and stiffness of the tine, and where $\ddot{y}(t)$ denotes the second derivative with respect to time of y. It is easy to verify that y given by

$$\forall\, t \in \mathbb{R}_+, \quad y(t) = \cos(\omega_0 t)$$

is a solution to the differential equation (just take its second derivative). Thus, the displacement of the tuning fork is sinusoidal. If we choose materials for the tuning fork so that $\omega_0 = 2\pi \times 440$ radians/second, then the tuning fork will produce the tone of A-440 on the musical scale.

   (a) Is $y(t) = \cos(\omega_0 t)$ the only solution? If not, give some others.

Solution: The following is a solution for any constant α:

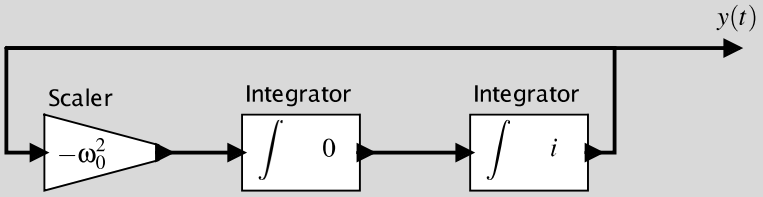
$$y(t) = \alpha \cos(\omega_0 t)\ .$$

   (b) Assuming the solution is $y(t) = \cos(\omega_0 t)$, what is the initial displacement?

Solution: $y(0) = \cos(\omega_0 \times 0) = 1$.

   (c) Construct a model of the tuning fork that produces y as an output using generic actors like Integrator, adder, scaler, or similarly simple actors. Treat the initial displacement as a parameter. Carefully label your diagram.

Solution: The following model will do the job:



Here, i is the initial displacement.

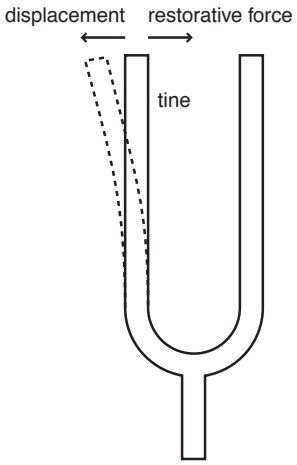


Figure 2.1: A tuning fork.

2. Show that if a system $S: A^{\mathbb{R}} \to B^{\mathbb{R}}$ is strictly causal and memoryless then its output is constant. Constant means that the output $(S(x))(t)$ at time t does not depend on t.

Solution: Since the system is memoryless, there exists a function $f: A \to B$ such that for all $x \in X$,

$$(S(x))(t) = f(x(t))$$

We need to show that for all $a_1, a_2 \in A$, $f(a_1) = f(a_2)$.

Since the system is strictly causal, for all $x_1, x_2 \in X$ and $\tau \in \mathbb{R}$,

$$x_1|_{t<\tau} = x_2|_{t<\tau} \Rightarrow S(x_1)|_{t\leq\tau} = S(x_2)|_{t\leq\tau}$$

Using the function f, this becomes

$$x_1|_{t<\tau} = x_2|_{t<\tau} \Rightarrow f(x_1(t)) = f(x_2(t))$$

for all $t \leq \tau$. The left side of this expression imposes no constraint at all on the values of $x_1(\tau)$ and $x_2(\tau)$, so these can be arbitrary values $a_1, a_2 \in A$. Hence, the right hand side asserts that $f(a_1) = f(a_2)$ for all $a_1, a_2 \in A$.

3. This exercise studies linearity.

   (a) Show that the helicopter model defined in Example 2.1 is linear if and only if the initial angular velocity $\dot{\theta}_y(0) = 0$.

Solution: The input T_y and output $\dot{\theta}_y$ are related by

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

First, we need to show that if $\dot{\theta}_y(0) = 0$, then superposition applies. Then we need to show if $\dot{\theta}_y(0) \neq 0$, superposition does not apply. For the first problem, if $\dot{\theta}_y(0) = 0$ then we have

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

Suppose the input is given by

$$T_y = aT_1 + bT_2$$

where a and b are real numbers and T_1 and T_2 are signals. Then the output is

$$\begin{aligned} \dot{\theta}_y(t) &= \frac{1}{I_{yy}} \int_0^t (aT_1(\tau) + bT_2(\tau)) d\tau \\ &= \frac{a}{I_{yy}} \int_0^t T_1(\tau) d\tau + \frac{b}{I_{yy}} \int_0^t T_1(\tau) d\tau. \end{aligned}$$

It is easy to see that the first term is a times what the output would be if the input were only T_1, and the second term is b times what the output would be if the input were only T_2. That is, if the system function is S, the output is

$$\dot{\theta}_y(t) = a(S(T_1))(t) + b(S(T_2))(t).$$

Next, assume that $\dot{\theta}_y(0) \neq 0$. With the same input as above, we get the output

$$\begin{aligned} \dot{\theta}_y(t) &= \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t (aT_1(\tau) + bT_2(\tau)) d\tau \\ &= \dot{\theta}_y(0) + \frac{a}{I_{yy}} \int_0^t T_1(\tau) d\tau + \frac{b}{I_{yy}} \int_0^t T_1(\tau) d\tau. \end{aligned}$$

We can now see that the output is

$$\dot{\theta}_y(t) = a(S(T_1))(t) + b(S(T_2))(t) - \dot{\theta}_y(0),$$

so superposition does not apply.

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

Solution: Given an actor with function S_1 and another with function S_2, the cascade composition is an actor with function $S_1 \circ S_2$, the composition of the two functions. If S_1 and S_2 both satisfy the superposition property, then

$$\begin{aligned} S_2(S_1(ax_1 + bx_2)) &= S_2(aS_1(x_1) + bS_1(x_2)) \\ &= aS_2(S_1(x_1)) + bS_2(S_1(x_2)). \end{aligned}$$

Hence, the composition also satisfies superposition.

(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.

Solution: A system model $S\colon X_1 \times X_2 \to Y$, where X_1, X_2, and Y are sets of signals, is linear if it satisfies the superposition property:

$$\begin{gathered} \forall\, x_1, x_1' \in X_1 \text{ and } \forall\, x_2, x_2' \in X_2 \text{ and } \forall\, a, b \in \mathbb{R}, \\ S(ax_1 + bx_1', ax_2 + bx_2') = aS(x_1, x_2) + bS(x_1', x_2'). \end{gathered}$$

The adder component is given by

$$S(x_1, x_2) = x_1 + x_2.$$

Hence

$$\begin{aligned} S(ax_1 + bx_1', ax_2 + bx_2') &= ax_1 + bx_1' + ax_2 + bx_2' \\ &= a(x_1 + x_2) + b(x_1' + x_2') \\ &= aS(x_1, x_2) + bS(x_1', x_2'). \end{aligned}$$

4. Consider the helicopter of Example 2.1, 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. Specifically, let the *x*-*y* plane be the plane orthogonal to the rotor shaft, and let the position of the tail at time t be given by a tuple $((x(t), y(t))$. Is this model LTI? Is it BIBO stable?

Solution: In this case, the system can be modeled as a function with two output signals,

$$S: (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})^2,$$

where

$$(S(T_y))(t) = (x(t), y(t)),$$

where $(x(t), y(t))$ is the position of the tail in the *x*-*y* plane. This model is clearly not linear. If the input torque doubles, for example, the output values will not double. In fact, the output values are constrained to lie on a circle centered at the origin, regardless of the input. For this reason, the model is BIBO stable. The output is always bounded. Thus, while our previous model was linear and unstable, this one is nonlinear and stable. Which model is more useful?

5. 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.

Solution:

$$\forall\, t \in \mathbb{R}, \quad \theta(t) = \theta(0) + \int_0^t \dot{\theta}(\tau) d\tau,$$

where $\theta(0)$ is the initial position.

   (b) Is this model BIBO stable?

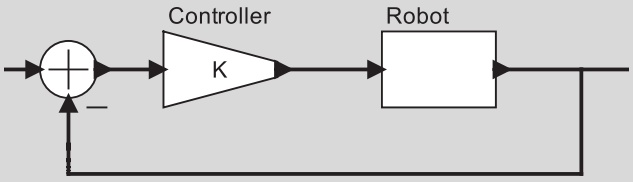
Solution: The model is not BIBO stable. For example, the input

$$\dot{\theta}(t) = u(t)$$

is bounded but yields an unbounded output.

   (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 $\psi(t) = au(t)$, where u is the unit step function. 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?

Solution: A proportional controller has the same structure as the helicopter controller:



Just as with the helicopter controller, we can solve the integral equation to get

$$\theta(t) = au(t)(1 - e^{-Kt}).$$

The output at zero is $\theta(0) = 0$, as expected. As t gets large, the output approaches a.

6. A DC motor produces a torque that is proportional to the current through the windings of the motor. Neglecting friction, the net torque on the motor, therefore, is this torque minus the torque applied by whatever load is connected to the motor. Newton's second law (the rotational version) gives

$$k_T i(t) - x(t) = I \frac{d}{dt}\omega(t), \tag{2.1}$$

where k_T is the motor torque constant, $i(t)$ is the current at time t, $x(t)$ is the torque applied by the load at time t, I is the moment of inertia of the motor, and $\omega(t)$ is the angular velocity of the motor.

(a) Assuming the motor is initially at rest, rewrite (2.1) as an integral equation.

Solution: Integrating both sides, we get

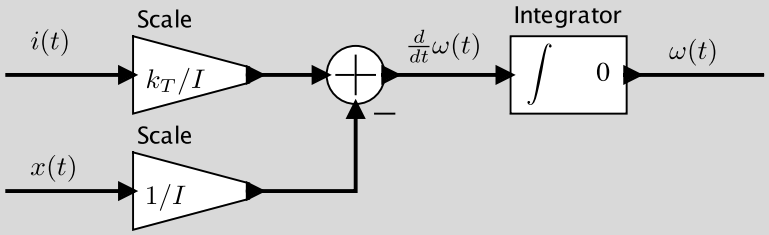
$$\int_0^t (k_T i(\tau) - x(\tau))d\tau = I\omega(t).$$

Solving for $\omega(t)$ we get

$$\omega(t) = \frac{1}{I}\int_0^t (k_T i(\tau) - x(\tau))d\tau.$$

(b) Assuming that both x and i are inputs and ω is an output, construct an actor model (a block diagram) that models this motor. You should use only primitive actors such as integrators and basic arithmetic actors such as scale and adder.

Solution: A solution is shown below:



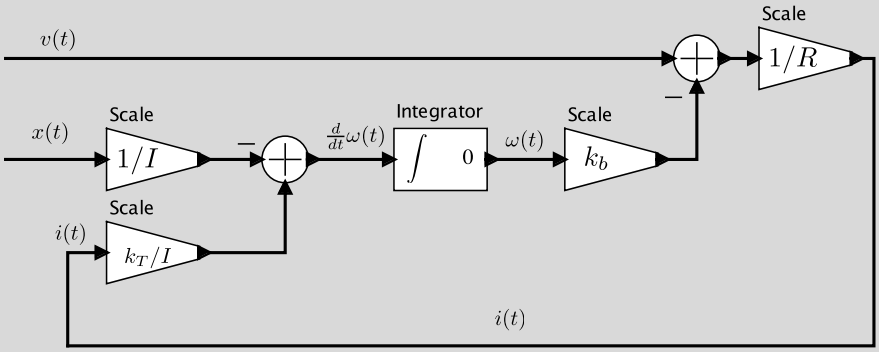
(c) In reality, the input to a DC motor is not a current, but is rather a voltage. If we assume that the inductance of the motor windings is negligible, then the relationship between voltage and current is given by

$$v(t) = Ri(t) + k_b\omega(t),$$

where R is the resistance of the motor windings and k_b is a constant called the motor back electromagnetic force constant. The second term appears because a rotating motor also functions as an electrical generator, where the voltage generated is proportional to the angular velocity.

Modify your actor model so that the inputs are v and x rather than i and x.

Solution: A solution is shown below:


$v(t)$
Scale
$1/R$
$x(t)$
Scale
$1/I$
$\frac{d}{dt}\omega(t)$
Integrator
$\int$ 0
$\omega(t)$
Scale
k_b
Scale
$i(t)$
k_T/I
$i(t)$

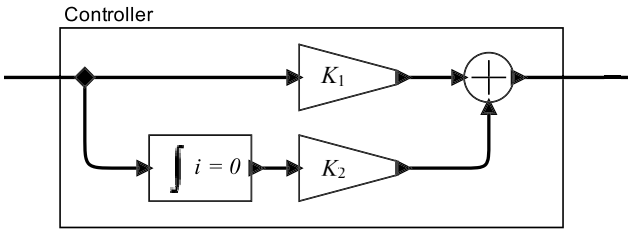
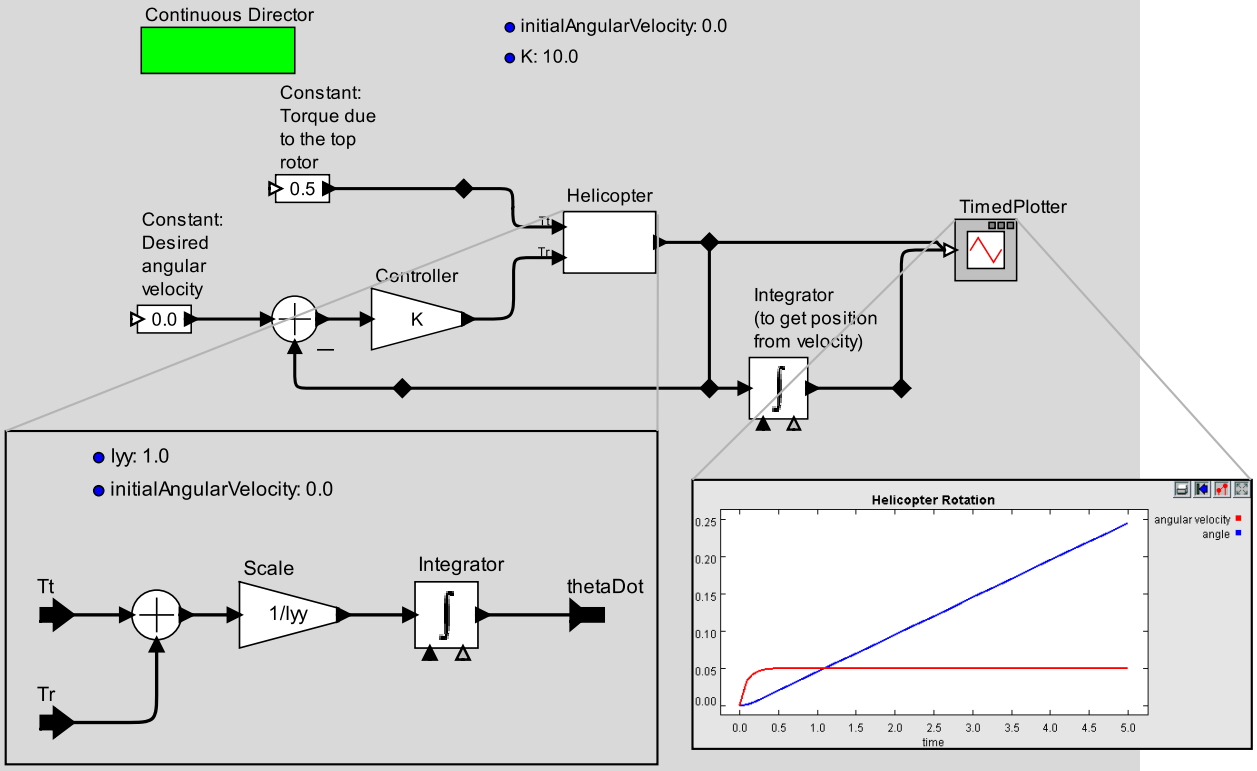


Figure 2.2: A PI controller for the helicopter.

7. (a) Using your favorite continuous-time modeling software (such as LabVIEW, Simulink, or Ptolemy II), construct a model of the helicopter control system shown in Figure 2.4. 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.

Solution: A Ptolemy model for the P controller is shown below:

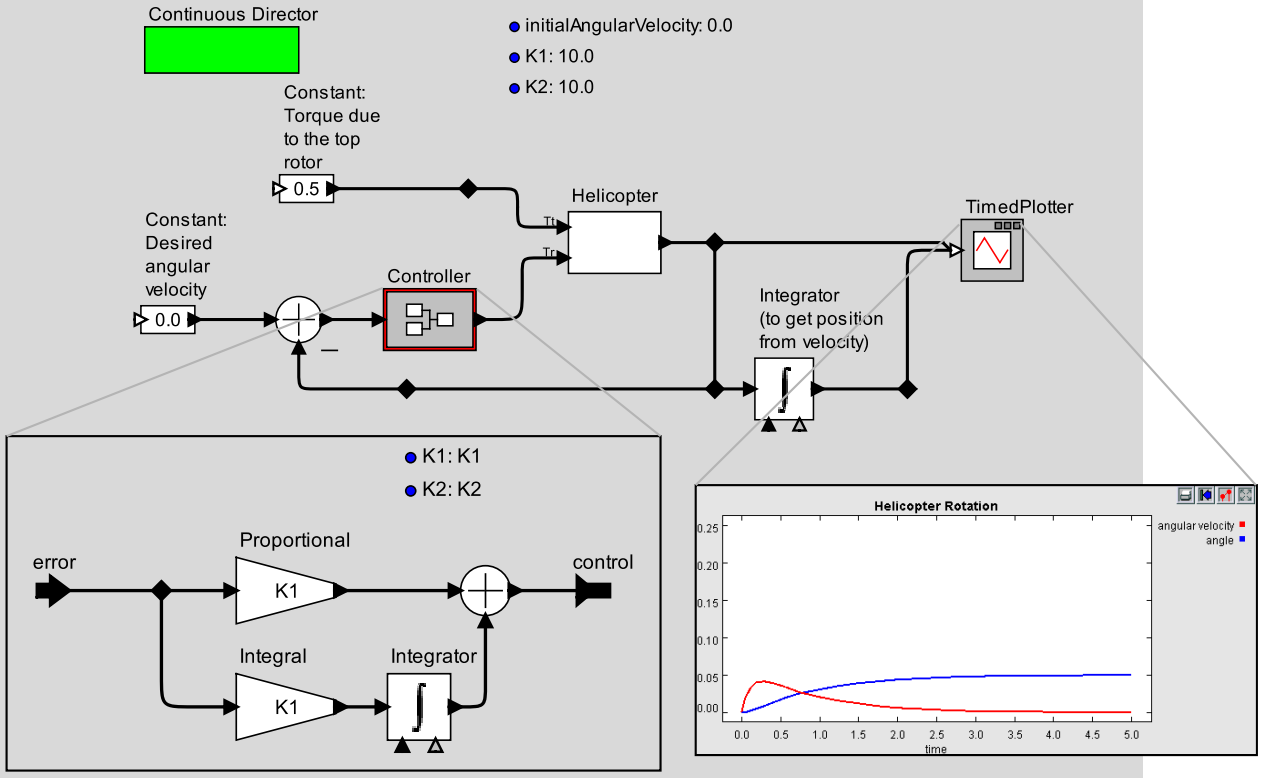


With the controller gain set to $K = 10$ and the top-rotor torque at $T_t(t) = 0.5u(t)$, we see that the angular velocity settles quickly to a constant $0.05 = 0.5/K$. Increasing K to 100 reduces this steady-state error to 0.005. Since the steady state error is in the angular velocity, the angle of the helicopter slowly increases (i.e., the helicopter rotates despite a desired angular velocity of zero).

(b) Modify the model of part (a) to replace the Controller of Figure 2.4 (the simple scale-by-K actor) with the alternative controller shown in Figure 2.2. 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).

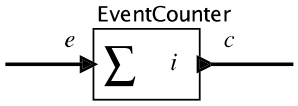
Solution: A Ptolemy model for the PI controller is shown below:



With the controller gains set to $K_1 = 10$ and $K_2 = 10$ and the top-rotor torque at $T_t(t) = 0.5u(t)$, we see that the angular velocity settles eventually to zero. Increasing K_1 results in a smaller peak error. Increasing K_2 results in fast settling, but also some overshoot.

3 Discrete Dynamics — Exercises

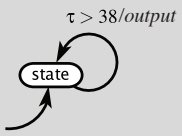
1. Consider an event counter that is a simplified version of the counter in Section 3.1. It has an icon like this:



This actor starts with state i and upon arrival of an event at the input, increments the state and sends the new value to the output. Thus, e is a pure signal, and c has the form $c\colon \mathbb{R} \to \{absent\} \cup \mathbb{N}$, assuming $i \in \mathbb{N}$. Suppose you are to use such an event counter in a weather station to count the number of times that a temperature rises above some threshold. Your task in this exercise is to generate a reasonable input signal e for the event counter. You will create several versions. For all versions, you will design a state machine whose input is a signal $\tau\colon \mathbb{R} \to \{absent\} \cup \mathbb{Z}$ that gives the current temperature (in degrees centigrade) once per hour. The output $e\colon \mathbb{R} \to \{absent, present\}$ will be a pure signal that goes to an event counter.

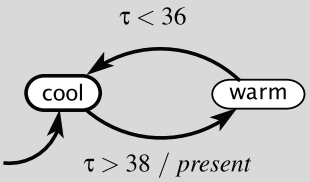
(a) For the first version, your state machine should simply produce a *present* output whenever the input is *present* and greater than 38 degrees. Otherwise, the output should be absent.

Solution: This state machine does not require more than one state:



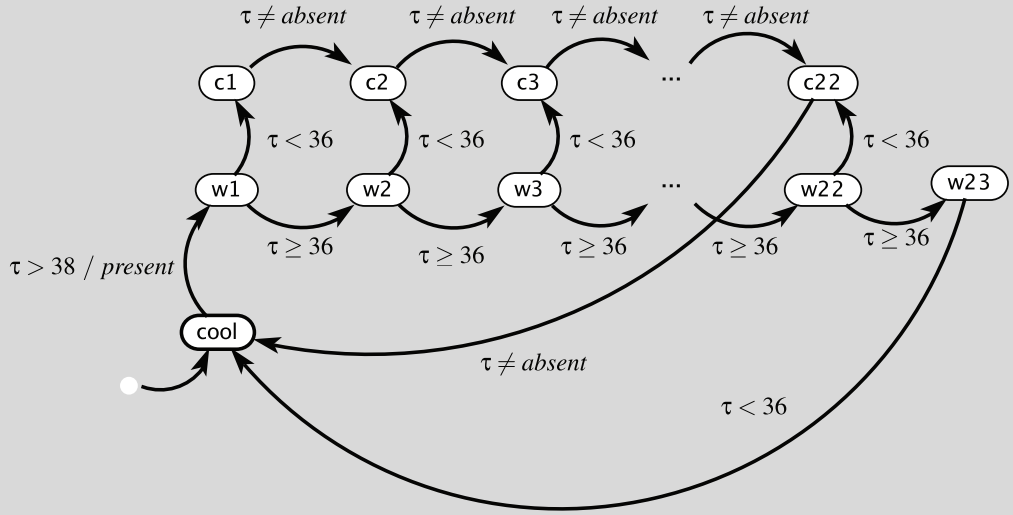
(b) For the second version, your state machine should have hysteresis. Specifically, it should produce a *present* output the first time the input is greater than 38 degrees, and subsequently, it should produce a *present* output anytime the input is greater than 38 degrees but has dropped below 36 degrees since the last time a *present* output was produced.

Solution:



(c) For the third version, your state machine should implement the same hysteresis as in part (b), but also produce a *present* output at most once per day.

Solution: Note that this problem statement is ambiguous. What is meant by "at most once per day?" Is it OK to produce a *present* output at 11 PM and again at 1 AM? Or does it mean that at least 24 hours should elapse between *present* outputs? Either would be correct, given the problem statement. Here is a solution under the second interpretation:

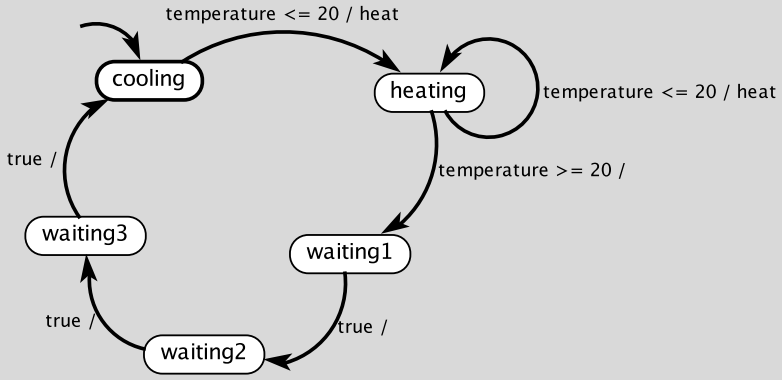


Note that with this solution, if the temperature stays high for several days, there will nonetheless be no *present* output for those several days. Is this likely to be what we intended? The problem appears to ask for this behavior, but it is probably not the behavior we want.

2. Consider a variant of the thermostat of example 3.5. In this variant, there is only one temperature threshold, and to avoid chattering the thermostat simply leaves the heat on or off for at least a fixed amount of time. In the initial state, if the temperature is less than or equal to 20 degrees Celsius, it turns the heater on, and leaves it on for at least 30 seconds. After that, if the temperature is greater than 20 degrees, it turns the heater off and leaves it off for at least 2 minutes. It turns it on again only if the temperature is less than or equal to 20 degrees.

   (a) Design an FSM that behaves as described, assuming it reacts exactly once every 30 seconds.

Solution: A solution is shown below:



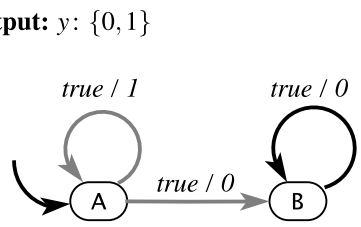
   (b) How many possible states does your thermostat have? Is this the smallest number of states possible?

Solution: The FSM has five states. Many alternative designs with more states exist.

   (c) Does this model thermostat have the time-scale invariance property?

Solution: The model does not have the hysteresis property because the timeout is a fixed amount of time, so varying the time scale of the input will yield distinctly different behavior.

3. Consider the following state machine:



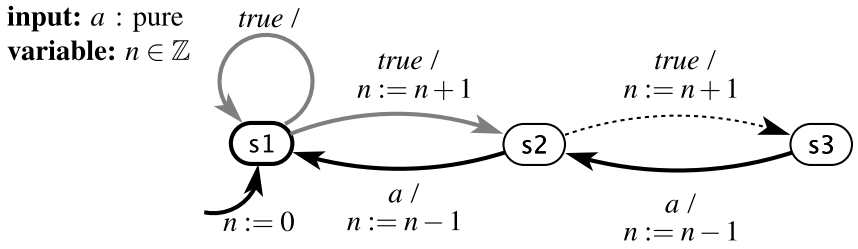
Determine whether the following statement is true or false, and give a supporting argument:

> The output will eventually be a constant 0, or it will eventually be a constant 1. That is, for some $n \in \mathbb{N}$, after the n-th reaction, either the output will be 0 in every subsequent reaction, or it will be 1 in every subsequent reaction.

Note that Chapter 13 gives mechanisms for making such statements precise and for reasoning about them.

Solution: TRUE. In an infinite execution, if the transition from A to B is ever taken, then after that point, the output will always be 0. If that transition is never taken, then the output will be a constant 1 for the entire execution. This too is allowed behavior for the state machine.

4. How many reachable states does the following state machine have?



Solution: Three. The variable n is redundant, because it always has value 0 in state s1, 1 in s2, and 2 in s3.

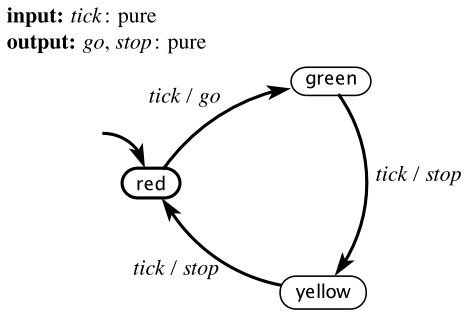


Figure 3.1: Deterministic finite-state machine for Exercise 5

5. Consider the deterministic finite-state machine in Figure 3.1 that models a simple traffic light.

   (a) Formally write down the description of this FSM as a 5-tuple:

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

Solution: The FSM description is:

$$\begin{aligned} States &= \{\mathsf{red}, \mathsf{yellow}, \mathsf{green}\} \\ Inputs &= (\{tick\} \to \{present, absent\}) \\ Outputs &= (\{go, stop\} \to \{present, absent\}) \\ initialState &= \mathsf{red} \end{aligned}$$

The update function is defined as:

$$update(s,i) = \begin{cases} (\mathsf{green}, go) & \text{if } s = \mathsf{red} \wedge i(tick) = present \\ (\mathsf{yellow}, stop) & \text{if } s = \mathsf{green} \wedge i(tick) = present \\ (\mathsf{red}, stop) & \text{if } s = \mathsf{yellow} \wedge i(tick) = present \\ (s, absent) & \text{otherwise} \end{cases}$$

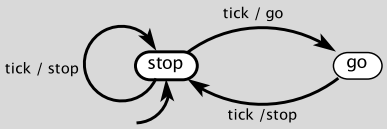
   (b) Give an execution trace of this FSM of length 4 assuming the input *tick* is *present* on each reaction.

Solution:

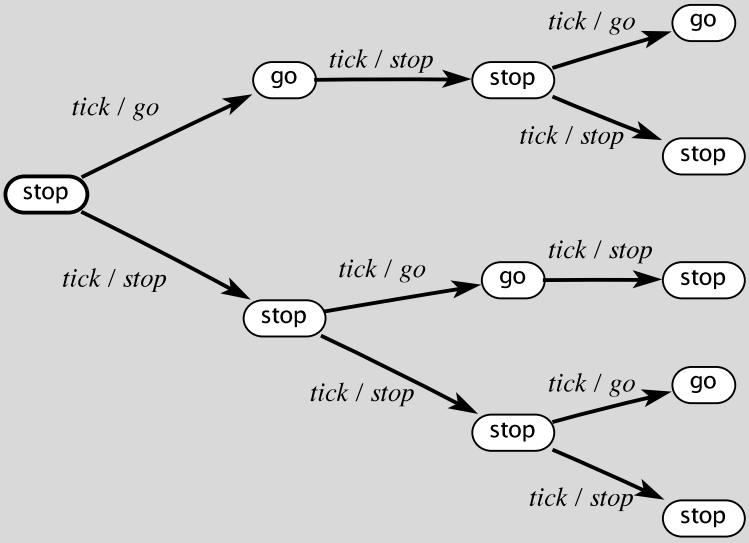
$$\mathsf{red} \xrightarrow{tick/go} \mathsf{green} \xrightarrow{tick/stop} \mathsf{yellow} \xrightarrow{tick/stop} \mathsf{red} \xrightarrow{tick/go} \cdots$$

   (c) Now consider merging the red and yellow states into a single stop state. Transitions that pointed into or out of those states are now directed into or out of the new stop state. Other transitions and the inputs and outputs stay the same. The new stop state is the new initial state. Is the resulting state machine deterministic? Why or why not? If it is deterministic, give a prefix of the trace of length 4. If it is nondeterministic, draw the computation tree up to depth 4.

Solution: The resulting state machine is given below. It is nondeterministic because there are two distinct transitions possible from state stop on input *tick*.

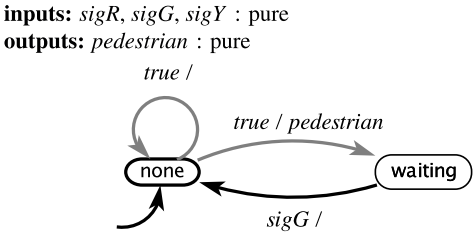


We have renamed the green state go. The computation tree for this FSM, up to depth 4, is given below:



6. This problem considers variants of the FSM in Figure 3.11, which models arrivals of pedestrians at a crosswalk. We assume that the traffic light at the crosswalk is controlled by the FSM in Figure 3.10. In all cases, assume a time triggered model, where both the pedestrian model and the traffic light model react once per second. Assume further that in each reaction, each machine sees as inputs the output produced by the other machine *in the same reaction* (this form of composition, which is called synchronous composition, is studied further in Chapter 6).

   (a) Suppose that instead of Figure 3.11, we use the following FSM to model the arrival of pedestrians:

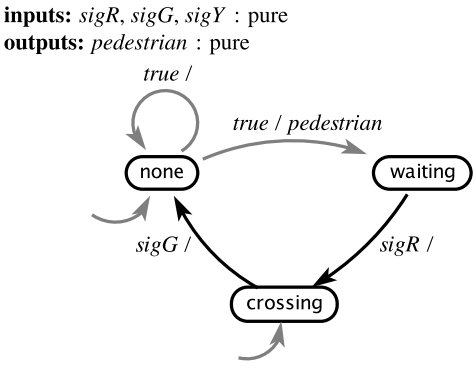


   Find a trace whereby a pedestrian arrives (the above machine transitions to waiting) but the pedestrian is never allowed to cross. That is, at no time after the pedestrian arrives is the traffic light in state red.

   Solution: The traffic light begins in state red while the pedestrian model begins in state none. Suppose that the pedestrian model transitions to waiting in exactly the same reaction where the traffic light transitions to state green. The system will now perpetually remain in the same state, where the pedestrian model is in waiting and the traffic light is in state green.

   Put another way, in the same reaction, we get red $\rightarrow$ green, which emits *sigG*, and none $\rightarrow$ waiting, which emits *pedestrian*. Once the composition is in state (green, none), all remaining reactions are stuttering transitions.

   (b) Suppose that instead of Figure 3.11, we use the following FSM to model the arrival of pedestrians:



   Here, the initial state is nondeterministically chosen to be one of none or crossing. Find a trace whereby a pedestrian arrives (the above machine transitions from none to waiting) but the pedestrian is never allowed to cross. That is, at no time after the pedestrian arrives is the traffic light in state red.

   Solution: Suppose the initial state is chosen to be (red, none) and sometime in the first 60 reactions transitions to (red, waiting). Then eventually the composite machine will transition to (green, waiting), after which all reactions will stutter.

7. Consider the state machine in Figure 3.2. State whether each of the following is a behavior for this machine. In each of the following, the ellipsis "$\cdots$" means that the last symbol is repeated forever. Also, for readability, *absent* is denoted by the shorthand a and *present* by the shorthand p.

   (a) $x = (p, p, p, p, p, \cdots)$, $y = (0, 1, 1, 0, 0, \cdots)$

   (b) $x = (p, p, p, p, p, \cdots)$, $y = (0, 1, 1, 0, a, \cdots)$

   (c) $x = (a, p, a, p, a, \cdots)$, $y = (a, 1, a, 0, a, \cdots)$

   (d) $x = (p, p, p, p, p, \cdots)$, $y = (0, 0, a, a, a, \cdots)$

   (e) $x = (p, p, p, p, p, \cdots)$, $y = (0, a, 0, a, a, \cdots)$

Solution:

(a) no

(b) yes

(c) no

(d) yes

(e) no

input: x: pure
output: y: $\{0, 1\}$

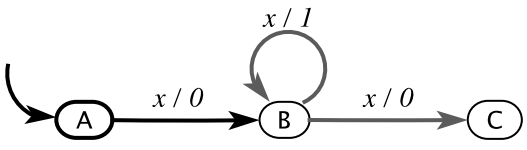


Figure 3.2: State machine for Exercise 7.

8. (NOTE: This exercise is rather advanced.) This exercise studies properties of discrete signals as formally defined in the sidebar on page 44. Specifically, we will show that discreteness is not a compositional property. That is, when combining two discrete behaviors in a single system, the resulting combination is not necessarily discrete.

   (a) Consider a pure signal $x: \mathbb{R} \to \{present, absent\}$ given by

   $$x(t) = \begin{cases} present & \text{if } t \text{ is a non-negative integer} \\ absent & \text{otherwise} \end{cases}$$

   for all $t \in \mathbb{R}$. Show that this signal is discrete.

   Solution: We need only to give an order-preserving one-to-one function of the form $f: T \to \mathbb{N}$, where T is defined as in the sidebar. In this case, the set of times when the signal is present is $T = \mathbb{N}$, so we can choose the identity function for f, which is trivial order preserving and one-to-one.

   (b) Consider a pure signal $y: \mathbb{R} \to \{present, absent\}$ given by

   $$y(t) = \begin{cases} present & \text{if } t = 1 - 1/n \text{ for any positive integer } n \\ absent & \text{otherwise} \end{cases}$$

   for all $t \in \mathbb{R}$. Show that this signal is discrete.

   Solution: We need only to give an order-preserving one-to-one function of the form $f: T \to \mathbb{N}$. In this case, the set of times when the signal is present is

   $$T = \{1 - 1/1, 1 - 1/2, 1 - 1/3, \cdots, 1 - 1/n, \cdots\}$$

   or

   $$T = \{0, 1/2, 2/3, \cdots\}$$

   Thus, we can define f as

   $$\forall\, t \in T, \quad f(t) = n \text{ where } t = 1 - 1/n\ .$$

   This is clearly one-to-one and order preserving. Hence, y is discrete.

   (c) Consider a signal w that is the merge of x and y in the previous two parts. That is, $w(t) = present$ if either $x(t) = present$ or $y(t) = present$, and is *absent* otherwise. Show that w is not discrete.

   Solution: Assume to the contrary that w is discrete. Then there exists an order-preserving one-to-one function $f: T \to \mathbb{N}$. Note that $1 \in T$ and that $1 - 1/n \in T$ for all $n \in \mathbb{N}$, $n > 0$. Moreover, $1 > 1 - 1/n$ for all $n \in \mathbb{N}$, $n > 0$. Since f is one-to-one and order preserving, it must be true that

   $$f(1) > f(1 - 1/n)\ .$$

   However, $1 - 1/n$ has no upper bound for $n \in \mathbb{N}$, $n > 0$, so $f(1 - 1/n)$ has no upper bound in $\mathbb{N}$, a contradiction. Hence, w must not be discrete.

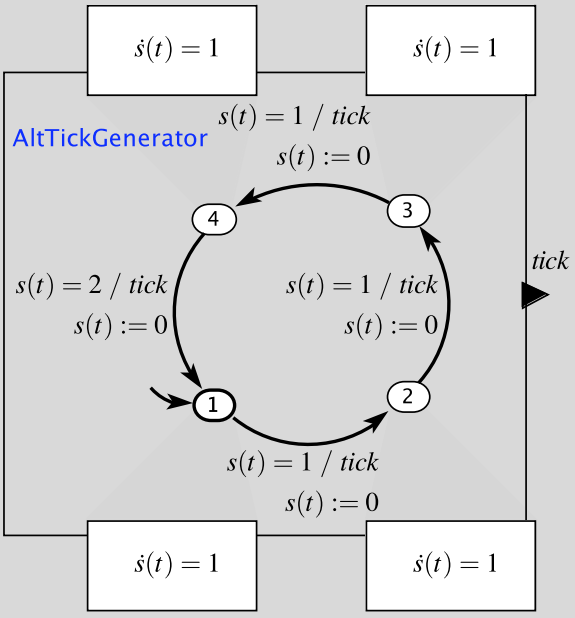
   (d) Consider the example shown in Figure 3.1. Assume that each of the two signals *arrival* and *departure* is discrete. Show that this does not imply that the output *count* is a discrete signal.

Solution: Let *arrival* $= x$ from part (a) and *departure* $= y$ from part (b). Then the set T of times when *count* is present is the same as the set T in part (c). Therefore, *count* is not discrete.

4 Hybrid Systems — Exercises

1. Construct (on paper is sufficient) a timed automaton similar to that of Figure 4.7 which produces *tick* at times $1,2,3,5,6,7,8,10,11,\cdots$. That is, ticks are produced with intervals between them of 1 second (three times) and 2 seconds (once).

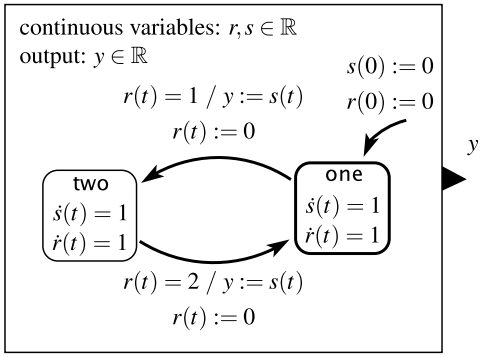
Solution: The following hybrid system will do the job:



Notice that all states have the same refinement, and that all transitions except the one into mode 1 have the same guard. The transition into mode 1 results in a delay of two units before the production of a *tick* event.

2. The objective of this problem is to understand a timed automaton, and then to modify it as specified.

   (a) For the timed automaton shown below, describe the output y. Avoid imprecise or sloppy notation.



Solution: The system generates a discrete signal with the event sequence

$$(1,3,4,6,7,9,10,\cdots)$$

at times

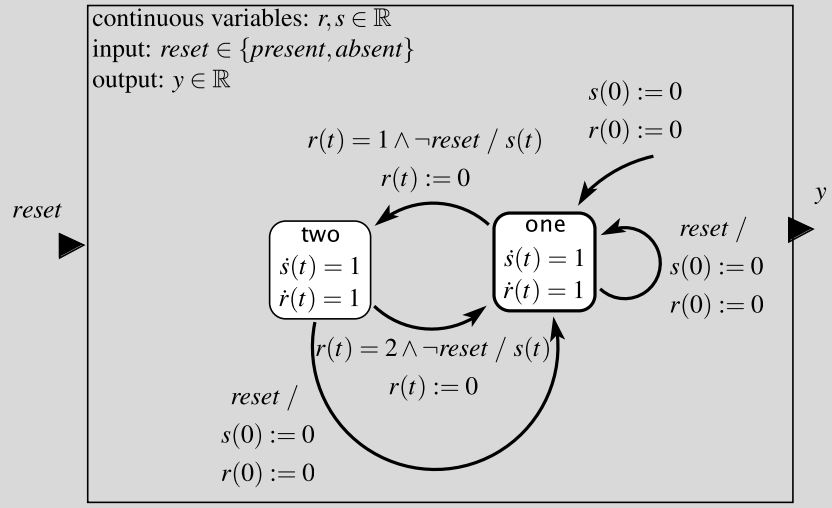
$$1,3,4,6,7,9,10,\cdots.$$

That is, the value of each output event is equal to the time at which it is produced, and the intervals between events alternate between one and two seconds. Precisely,

$$y(t) = \begin{cases} t & \text{if } t = 3k \text{ for some } k \in \mathbb{N} \\ t & \text{if } t = 3k+1 \text{ for some } k \in \mathbb{N}_0 \\ \textit{absent} & \text{otherwise} \end{cases}$$

   (b) Assume there is a new pure input *reset*, and that when this input is present, the hybrid system starts over, behaving as if it were starting at time 0 again. Modify the hybrid system from part (a) to do this.

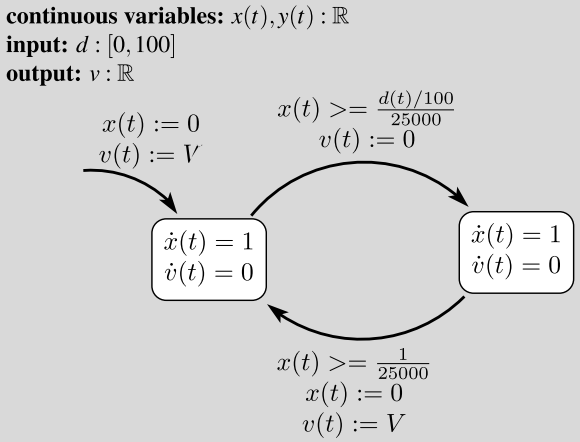
Solution:



3. In Exercise 6 of Chapter 2, we considered a DC motor that is controlled by an input voltage. Controlling a motor by varying an input voltage, in reality, is often not practical. It requires analog circuits that are capable of handling considerable power. Instead, it is common to use a fixed voltage, but to turn it on and off periodically to vary the amount of power delivered to the motor. This technique is called pulse width modulation (PWM).

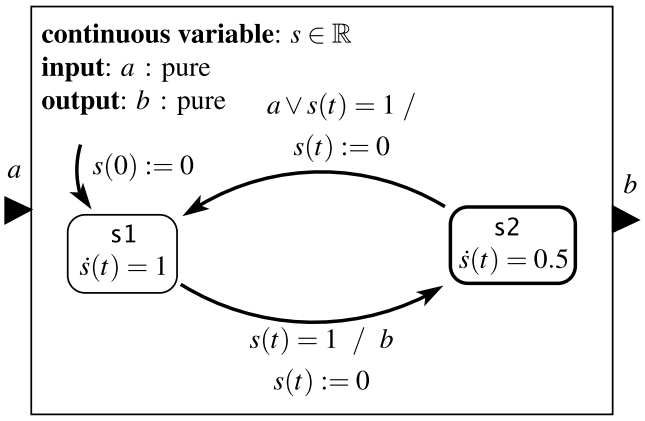
   Construct a timed automaton that provides the voltage input to the motor model from Exercise 6. Your hybrid system should assume that the PWM circuit delivers a 25 kHz square wave with a duty cycle between zero and 100%, inclusive. The input to your hybrid system should be the duty cycle, and the output should be the voltage.

Solution: A solution is shown below:



V is the high output voltage. The left state represents the situation where the output voltage is high, and the right state represents the situation where the output voltage is low. The input d represents the desired duty cycle, and here it assumed to be continuous (always present). If it were discrete, we would need for the automaton to store each provided value in a local variable.

4. Consider the following timed automaton:



Assume that the input signals a and b are discrete continuous-time signals, meaning that each can be given as a function of form $a: \mathbb{R} \to \{present, absent\}$, where at almost all times $t \in \mathbb{R}$, $a(t) = absent$. Assume that the state machine can take at most one transition at each distinct time t, and that machine begins executing at time $t = 0$.

(a) Sketch the output b if the input a is present only at times

$$t = 0.75, 1.5, 2.25, 3, 3.75, 4.5, \cdots$$

Include at least times from $t = 0$ to $t = 5$.

Solution: The output is present at times $t = 1, 2.5, 4, \cdots$.

(b) Sketch the output b if the input a is present only at times $t = 0, 1, 2, 3, \cdots$.

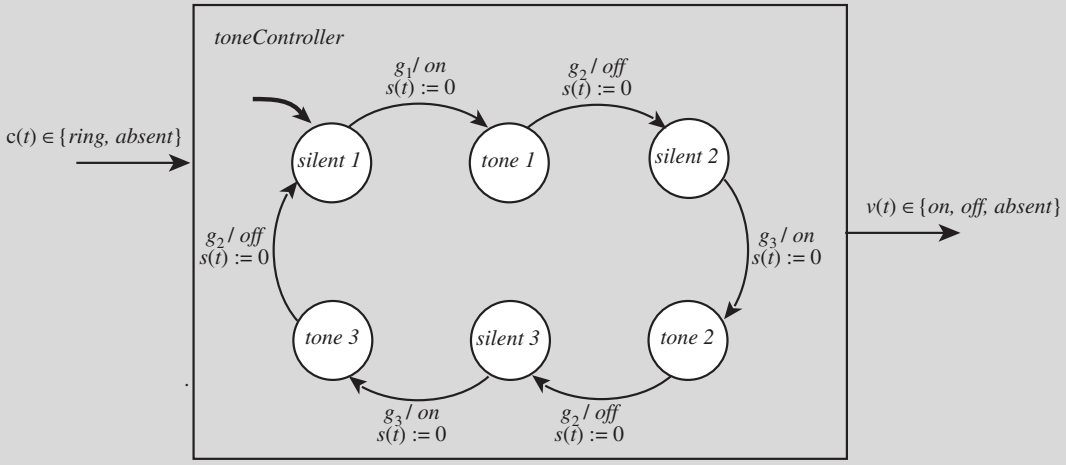
Solution: The output is present at times $t = 1, 3, 5, 7, \cdots$.

(c) Assuming that the input a can be any discrete signal at all, find a lower bound on the amount of time between events b. What input signal a (if any) achieves this lower bound?

Solution: The lower bound is 1. There is no input a that achieves this bound, but an input that comes arbitrarily close is where a is present at times $t = 1 + \varepsilon, 2 + 2\varepsilon, 3 + 3\varepsilon, \cdots$, for any $\varepsilon > 0$.

5. You have an analog source that produces a pure tone. You can switch the source on or off by the input event *on* or *off*. Construct a timed automaton that provides the *on* and *off* signals as outputs, to be connected to the inputs of the tone generator. Your system should behave as follows. Upon receiving an input event *ring*, it should produce an 80 ms-long sound consisting of three 20 ms-long bursts of the pure tone separated by two 10 ms intervals of silence. What does your system do if it receives two *ring* events that are 50 ms apart?

Solution: Assume the input alphabet is $\{ring, absent\}$ and the output alphabet is $\{on, off, absent\}$. Then the following timed automaton will control the source of the tone:



All states except *silent 1* have as a refinement the system given by

$$\dot{s}(t) = 1.$$

The guards are given by

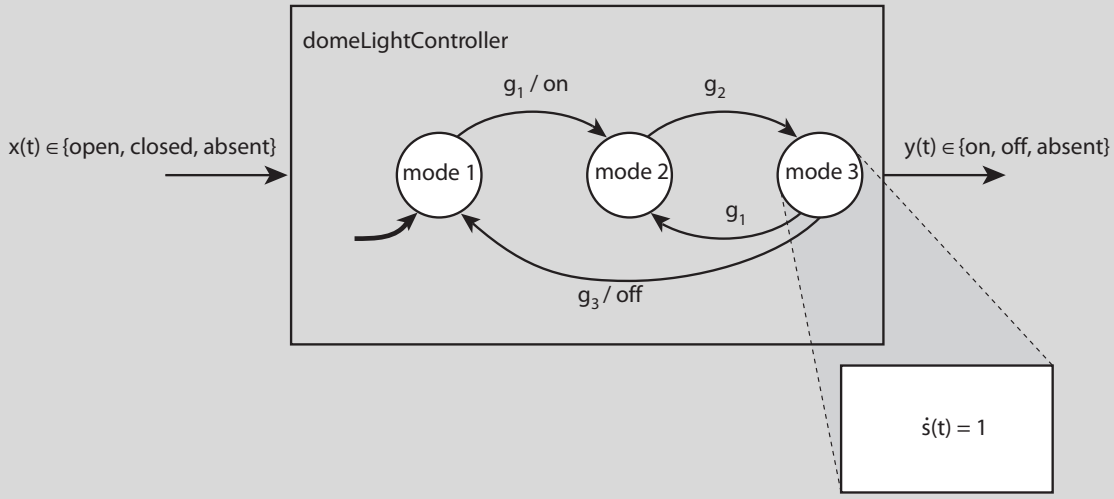
$$\begin{aligned} g_1 &= \{(c(t), s(t)) \mid c(t) = ring\} \\ g_2 &= \{(c(t), s(t)) \mid s(t) = 20\} \\ g_3 &= \{(c(t), s(t)) \mid s(t) = 10\}. \end{aligned}$$

This system ignores a *ring* input event that occurs less than 80 ms after the previous *ring* event.

6. Automobiles today have the features listed below. Implement each feature as a timed automaton.

   (a) The dome light is turned on as soon as any door is opened. It stays on for 30 seconds after all doors are shut. What sensors are needed?

Solution: Assume that the automobile provides the input *open* when the first door is opened and *closed* when the last open door is closed. The following machine provides *on* to turn on the dome light and *off* to turn it off:



The guards are given by

$$\begin{aligned} g_1 &= \{(x(t), s(t)) \mid x(t) = open\} \\ g_2 &= \{(x(t), s(t)) \mid x(t) = closed\} \\ g_3 &= \{(x(t), s(t)) \mid s(t) = 30\}. \end{aligned}$$

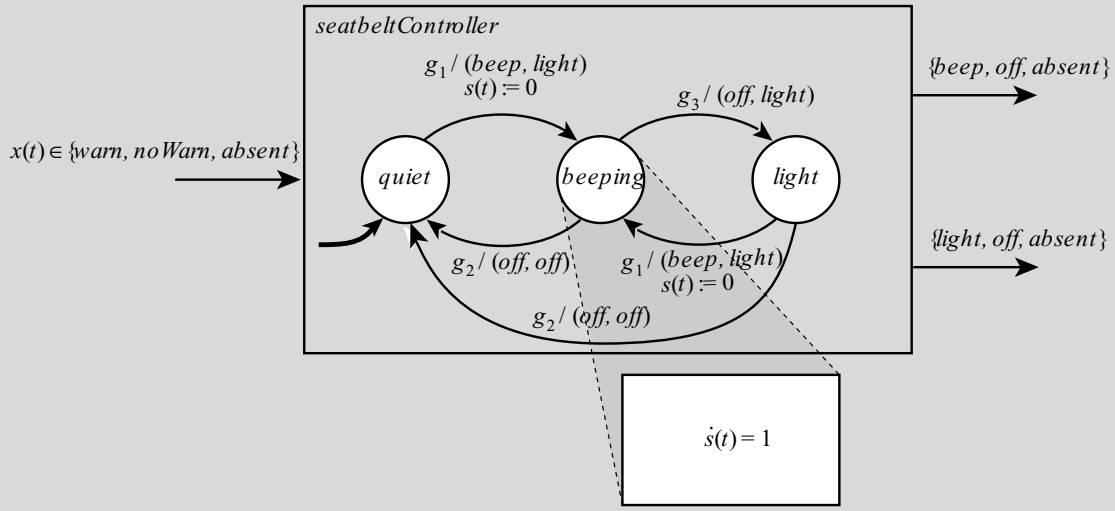
Such sensors can be easily implemented as a **daisy chain**, a series connection of switches so that if any door is open, the electrical circuit is open.

   (b) Once the engine is started, a beeper is sounded and a red light warning is indicated if there are passengers that have not buckled their seat belt. The beeper stops sounding after 30 seconds, or as soon the seat belts are buckled, whichever is sooner. The warning light is on all the time the seat belt is unbuckled. **Hint:** Assume the sensors provide a *warn* event when the ignition is turned on and there is a seat with passenger not buckled in, or if the ignition is already on and a passenger sits in a seat without buckling the seatbelt. Assume further that the sensors provide a *noWarn* event when a passenger departs from a seat, or when the buckle is buckled, or when the ignition is turned off.

Solution: Assume that the vehicle sensors provide the following input alphabet,

$$Inputs = \{warn, noWarn, absent\},$$

as suggested in the hint. The following model provides the requisite control:

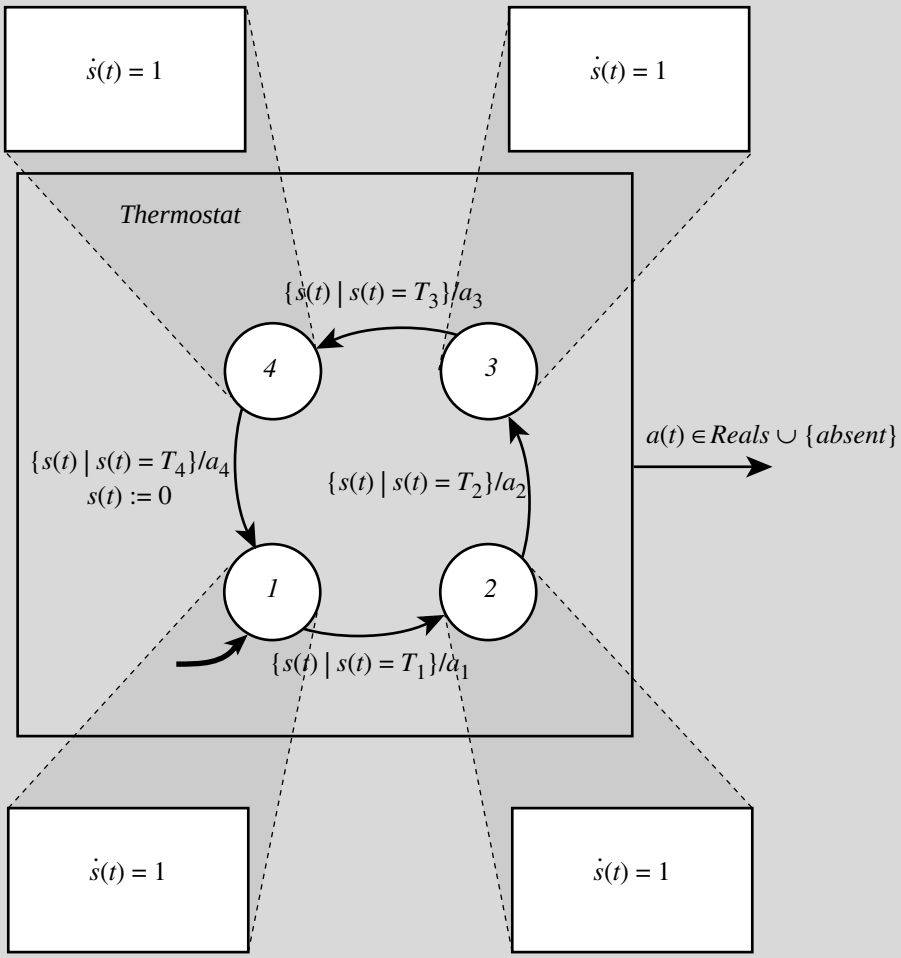


The guards are given by

$$
\begin{aligned}
g_1 &= \{(x(t), s(t)) \mid x(t) = warn\} \\
g_2 &= \{(x(t), s(t)) \mid x(t) = noWarn\} \\
g_3 &= \{(x(t), s(t)) \mid s(t) = 30 \wedge x(t) = absent\}.
\end{aligned}
$$

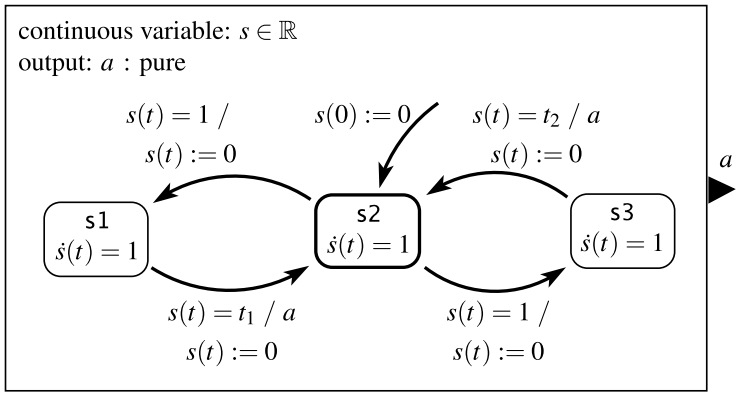
7. A programmable thermostat allows you to select 4 times, $0 \leq T_1 \leq \cdots \leq T_4 < 24$ (for a 24-hour cycle) and the corresponding setpoint temperatures $a_1, \cdots, a_4$. Construct a timed automaton that sends the event a_i to the heating systems controller. The controller maintains the temperature close to the value a_i until it receives the next event. How many timers and modes do you need?

Solution: The following hybrid system will do the job:



You need four modes and one timer.

8. Consider the following timed automaton:



Assume t_1 and t_2 are positive real numbers. What is the minimum amount of time between events a? That is, what is the smallest possible time between two times when the signal a is present?

Solution: $1 + \min(t_1, t_2)$.

9. Figure 4.1 depicts the intersection of two one-way streets, called Main and Secondary. A light on each street controls its traffic. Each light goes through a cycle consisting of a red (R), green (G), and yellow (Y) phases. It is a safety requirement that when one light is in its green or yellow phase, the other is in its red phase. The yellow phase is always 5 seconds long.

   The traffic lights operate as follows. A sensor in the secondary road detects a vehicle. While no vehicle is detected, there is a 4 minute-long cycle with the main light having 3 minutes of green, 5 seconds of yellow, and 55 seconds of red. The secondary light is red for 3 minutes and 5 seconds (while the main light is green and yellow), green for 50 seconds, then yellow for 5 seconds.

   If a vehicle is detected on the secondary road, the traffic light quickly gives a right of way to the secondary road. When this happens, the main light aborts its green phase and immediately switches to its 5 second yellow phase. If the vehicle is detected while the main light is yellow or red, the system continues as if there were no vehicle.

   Design a hybrid system that controls the lights. Let this hybrid system have six pure outputs, one for each light, named *mG*, *mY*, and *mR*, to designate the main light being green, yellow, or red, respectively, and *sG*, *sY*, and *sR*, to designate the secondary light being green, yellow, or red, respectively. These signals should be generated to turn on a light. You can implicitly assume that when one light is turned on, whichever has been on is turned off.

Solution: The hybrid system is described below. Time t is in seconds. Four modes are needed corresponding to the four possible light configurations.

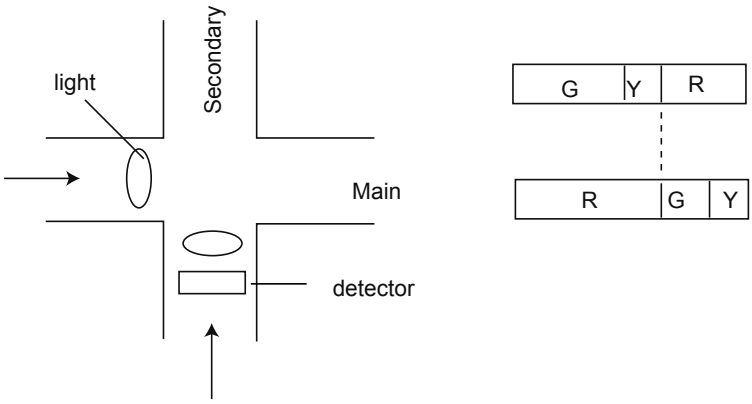
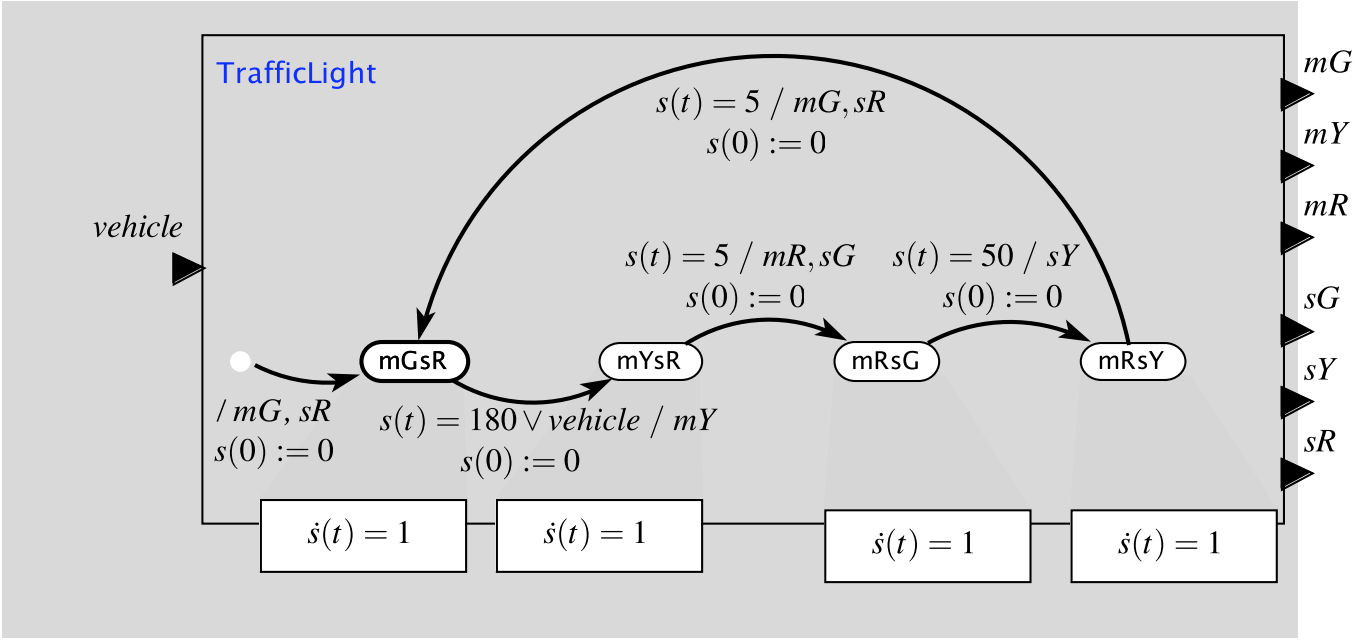


Figure 4.1: Traffic lights control the intersection of a main street and a secondary street. A detector senses when a vehicle crosses it. The red phase of one light must coincide with the green and yellow phases of the other light.


TrafficLight
vehicle
mG
mY
mR
sG
sY
sR
s(t) = 5 / mG, sR
s(0) := 0
s(t) = 5 / mR, sG
s(0) := 0
s(t) = 50 / sY
s(0) := 0
mGsR
mYsR
mRsG
mRsY
/ mG, sR
s(0) := 0
s(t) = 180 ∨ vehicle / mY
s(0) := 0
ṡ(t) = 1
ṡ(t) = 1
ṡ(t) = 1
ṡ(t) = 1

10. For the bouncing ball of Example 4.7, let t_n be the time when the ball hits the ground for the n-th time, and let $v_n = \dot{y}(t_n)$ be the velocity at that time.

(a) Find a relation between v_{n+1} and v_n for $n > 1$, and then calculate v_n in terms of v_1.

Solution: Suppose v_n is the velocity just before the nth bounce. Then $v_{n+1} = av_n$ and so $v_n = a^{n-1}v_1$.

(b) Obtain t_n in terms of v_1 and a. Use this to show that the bouncing ball is a Zeno system. **Hint:** The **geometric series identity** might be useful, where for $|b| < 1$,

$$\sum_{m=0}^{\infty} b^m = \frac{1}{1-b}.$$

Solution: The time $t_n - t_{n-1}$ is the time taken by the ball to return to the ground, starting at velocity v_n. So for $n > 1$, $t_n - t_{n-1} = 2v_n/g = 2a^{n-1}v_1$, where $t_1 = \sqrt{2h_0/g}$, as derived in Example 4.7. Hence, for $n > 1$,

$$t_n = \sqrt{2h_0/g} + 2v_1 \sum_{k=1}^{n-1} a^k.$$

By change of variables, this can be written

$$t_n = \sqrt{2h_0/g} + 2av_1 \sum_{m=0}^{m-2} a^k.$$

As n goes to infinity, this approaches a limit of

$$t_\infty = \sqrt{2h_0/g} + \frac{2av_1}{1-a},$$

where we have used the geometric series identity. Since this limit is finite, the system is Zeno.

(c) Calculate the maximum height reached by the ball after successive bumps.

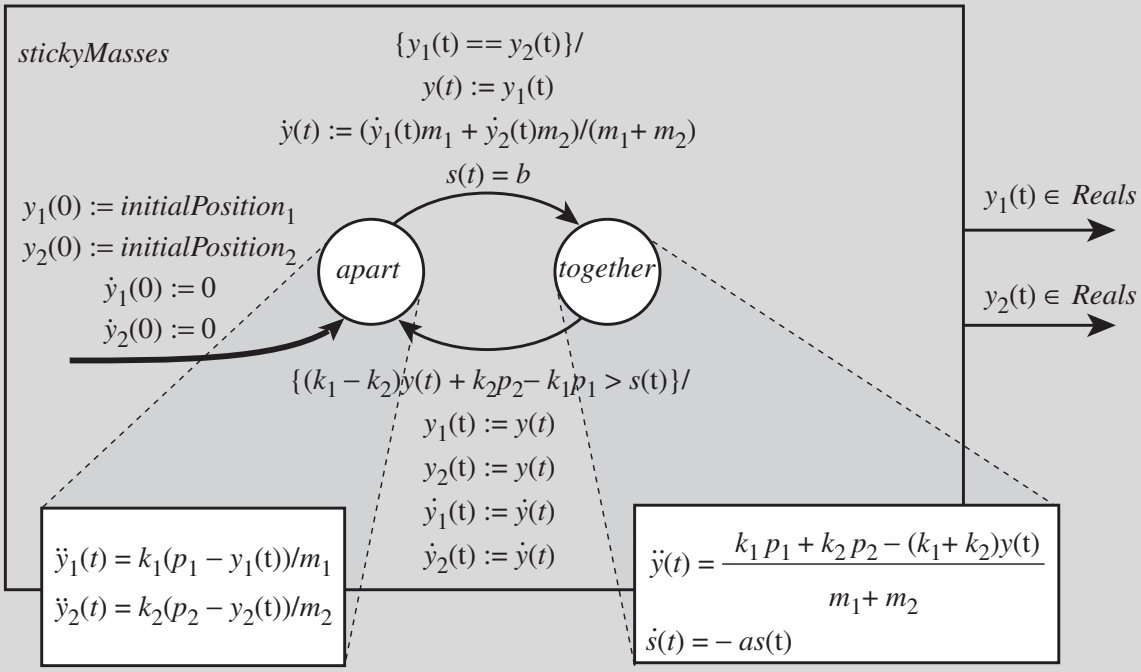
Solution: The maximum height reached after the $(n-1)$st bounce is $\frac{1}{2}\frac{(v_n)^2}{g}$.

11. Elaborate the hybrid system model of Figure 4.10 so that in the *together* mode, the stickiness decays according to the differential equation

$$\dot{s}(t) = -as(t)$$

where $s(t)$ is the stickiness at time t, and a is some positive constant. On the transition into this mode, the stickiness should be initialized to some starting stickiness b.

Solution: The model is shown below



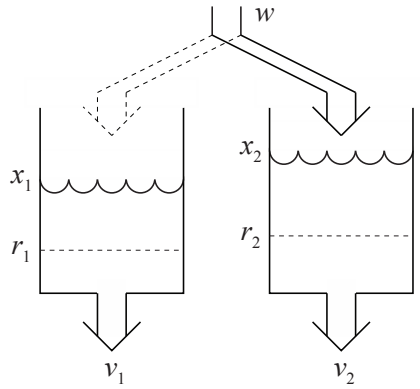


Figure 4.2: Water tank system.

12. Show that the trajectory of the AGV of Figure 4.13 while it is in *left* or *right* mode is a circle. What is the radius of this circle, and how long does it take to complete a circle?

Solution: In *left* mode, the refinement state evolves according to

$$\begin{aligned} \dot{x}(t) &= 10\cos\phi(t) \\ \dot{y}(t) &= 10\sin\phi(t) \\ \dot{\phi}(t) &= \pi \end{aligned}$$

Intuitively this suggests that the vehicle moves in a circle with speed 10 mph. By integrating these differential equations we get

$$\begin{aligned} x(t)-x(0) &= \int_0^t \cos(\phi(0)+\pi t)dt = \frac{10}{\pi}\sin(\phi(0)+\pi t) \\ y(t)-y(0) &= \int_0^t \sin(\phi(0)+\pi t)dt = -\frac{10}{\pi}\cos(\phi(0)+\pi t) \\ \phi(t)-\phi(0) &= \pi t \end{aligned}$$

So $(x(t)-x(0))^2+(y(t)-y(0))^2 = \frac{10^2}{\pi^2}$, which means the vehicle is moving in a circle of radius $10/\pi$, and it takes 2 seconds to complete a circle.

13. Consider Figure 4.2 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 $x_1(t) \leq r_1(t)$ and to Tank 2 whenever $x_2(t) \leq r_2(t)$.

The hybrid automaton representing this two-tank system is given in Figure 4.3.

Answer the following questions:

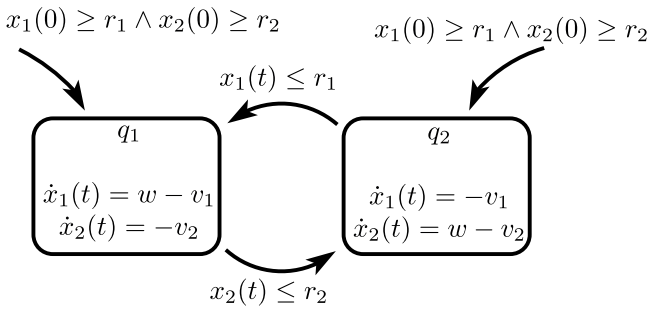
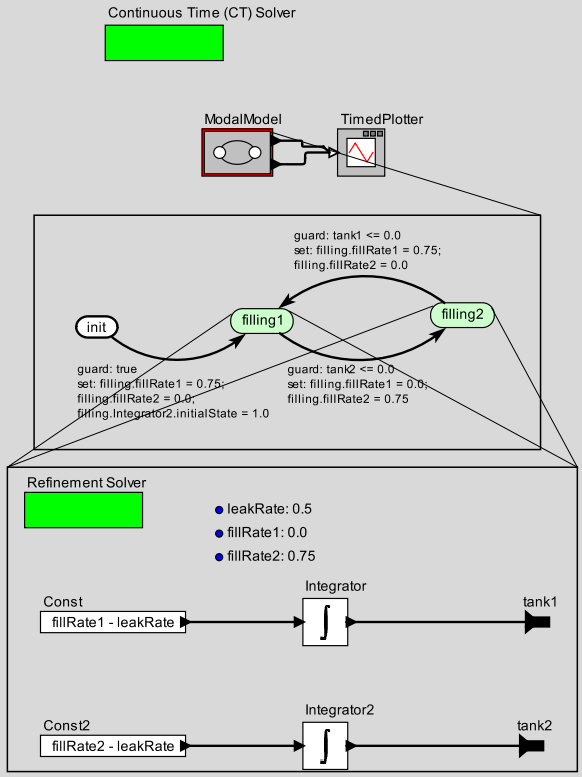


Figure 4.3: Hybrid automaton representing water tank system.

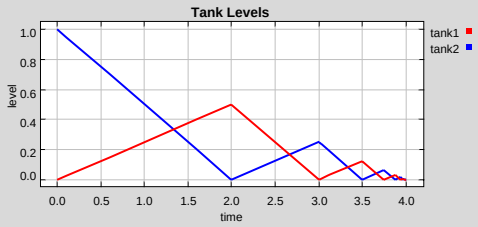
(a) Construct a model of this hybrid automaton in Ptolemy II, LabVIEW, or Simulink. Use the following parameter values: $r_1 = r_2 = 0$, $v_1 = v_2 = 0.5$, and $w = 0.75$. Set the initial state to be $(q_1, (0, 1))$. (That is, initial value $x_1(0)$ is 0 and $x_2(0)$ is 1.)

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

Solution: The Ptolemy II model is shown below:



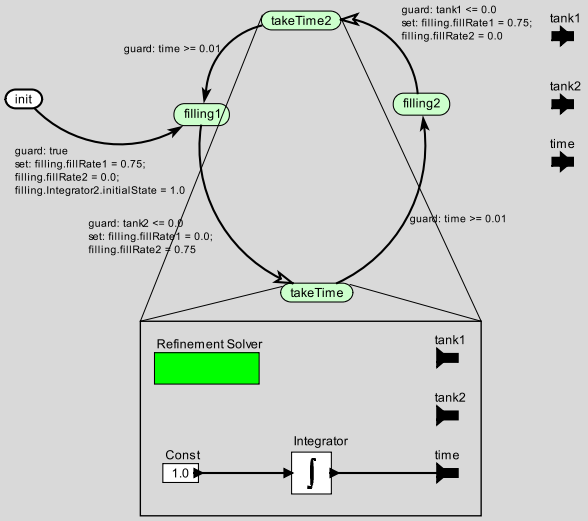
A plot of the execution is shown below:



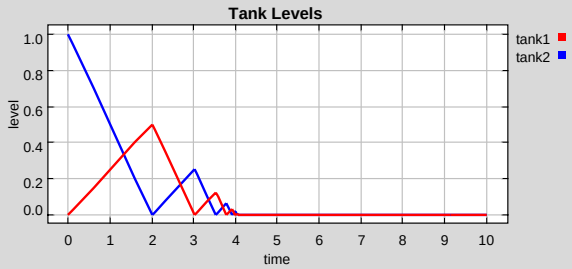
The model exhibits Zeno behavior because the net outflow is $0.5 + 0.5 = 1.0$, which is less than the net inflow of 0.75. Thus, no matter how we switch the inflow between tanks, we cannot keep water in both tanks past time 4.0. The hybrid system is designed to switch whenever one tank becomes empty, and the time between these events gets infinitely small.

(b) A Zeno system may be **regularized** by ensuring that the time between transitions is never less than some positive number ε. This can be emulated by inserting extra modes in which the hybrid automaton dwells for time ε. Use regularization to make your model from part (a) 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.

Solution: The Ptolemy II model for the regularized model is shown below:



In the above implementation, two modes have been added to take time, set to $\varepsilon = 0.01$. Otherwise, the model is the same as the original. A plot of the execution is shown below:



Note that there is no longer any Zeno behavior. Zooming in to the plot shows that the tanks spend a non-zero amount of time being empty before they start to get filled. Note that this is not the only way to regularize this model. In fact, it's probably not the best way, since in the "takeTime"

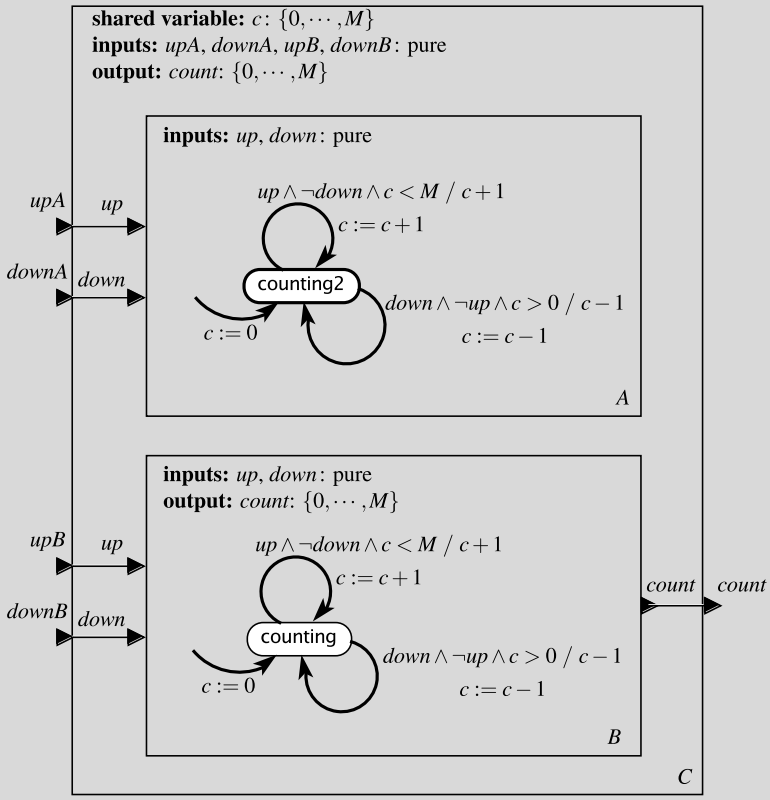
modes, the incoming water flow does not go into either tank. This could be a reasonable model if the filling hose takes time to move from one tank to the other and the water is spilled or shut off during that time.

Include printouts of your plots with your answer.

5 Composition of State Machines — Exercises

1. Consider the extended state machine model of Figure 3.8, the garage counter. Suppose that the garage has two distinct entrance and exit points. Construct a side-by-side concurrent composition of two counters that share a variable c that keeps track of the number of cars in the garage. Specify whether you are using synchronous or asynchronous composition, and define exactly the semantics of your composition by giving a single machine modeling the composition. If you choose synchronous semantics, explain what happens if the two machines simultaneously modify the shared variable. If you choose asynchronous composition, explain precisely which variant of asynchronous semantics you have chosen and why. Is your composition machine deterministic?

Solution: A solution is shown here:



It would not be acceptable in this case to miss events, so asynchronous semantics 1 will not be a good choice. We can choose, for example, a synchronous interleaving semantics where machine *A* always reacts before machine *B*. Note that if we allow the order of reactions to be nondeterministic, then the *count* output will not necessarily reflect the final number of cars in the garage. This is analogous to a synchronous cascade composition, where the upstream machine reacts before the downstream machine, even though logically both machines react simultaneously.

2. For semantics 2 in Section 5.1.2, give the five tuple for a single machine representing the composition C,

$$(States_C, Inputs_C, Outputs_C, update_C, initialState_C)$$

for the side-by-side asynchronous composition of two state machines A and B. Your answer should be in terms of the five-tuple definitions for A and B,

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

and

$$(States_B, Inputs_B, Outputs_B, update_B, initialState_B)$$

Solution:

$$\begin{aligned} States_C &= States_A \times States_B \\ Inputs_C &= Inputs_A \times Inputs_B \\ Outputs_C &= Outputs_A \times Outputs_B \\ initialState_C &= (initialState_A, initialState_B) \end{aligned}$$

The update function is:

$$update_C((s_A, s_B), (i_A, i_B)) = ((s'_A, s'_B), (o'_A, o'_B)),$$

where either

$$(s'_A, o'_A) = update_A(s_A, i_A) \text{ and } s'_B = s_B \text{ and } o'_B = absent$$

or

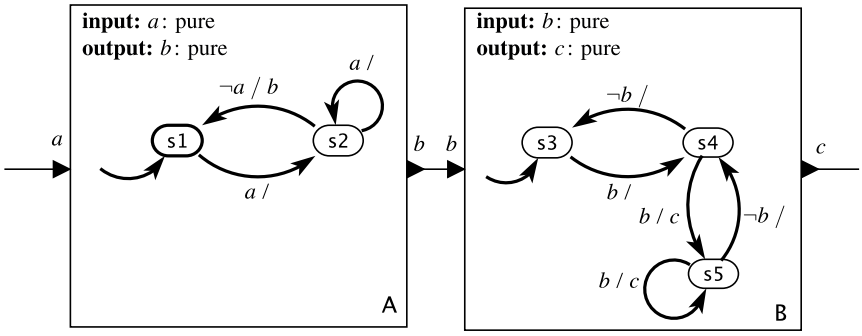
$$(s'_B, o'_B) = update_B(s_B, i_B) \text{ and } s'_A = s_A \text{ and } o'_A = absent$$

or

$$(s'_A, o'_A) = update_A(s_A, i_A) \text{ and } (s'_B, o'_B) = update_B(s_B, i_B)$$

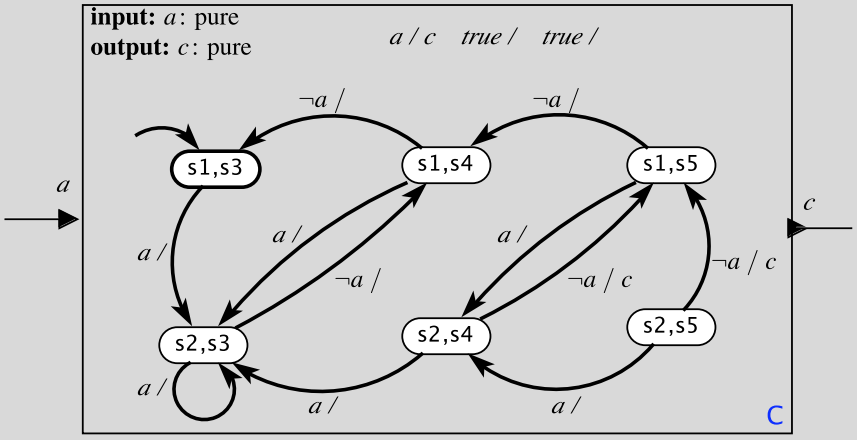
for all $s_A \in States_A$, $s_B \in States_B$, $i_A \in Inputs_A$, and $i_B \in Inputs_B$.

3. Consider the following synchronous composition of two state machines A and B:



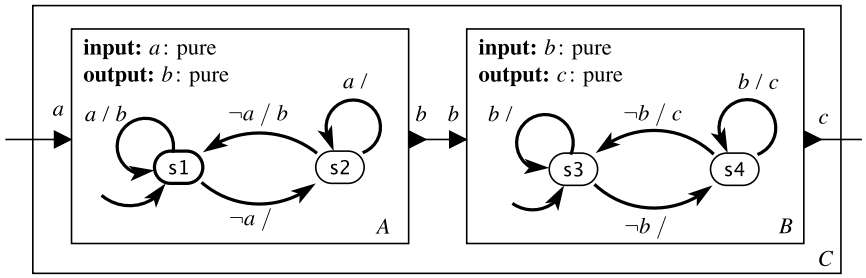
Construct a single state machine C representing the composition. Which states of the composition are unreachable?

Solution:



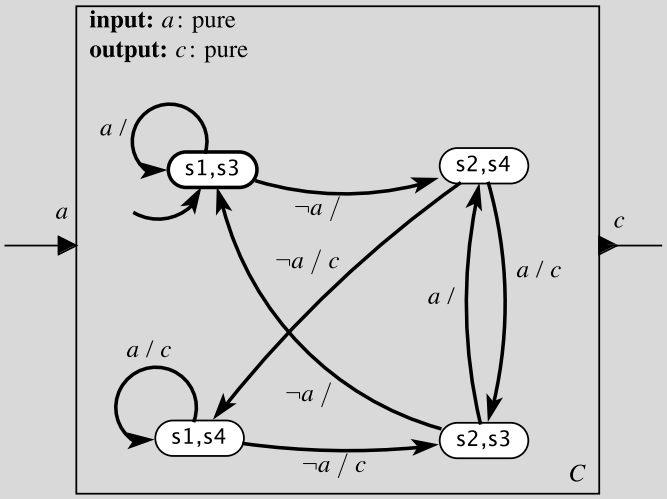
The following states are unreachable: (s1,s5), (s2,s4), and (s2,s5).

4. Consider the following synchronous composition of two state machines A and B:



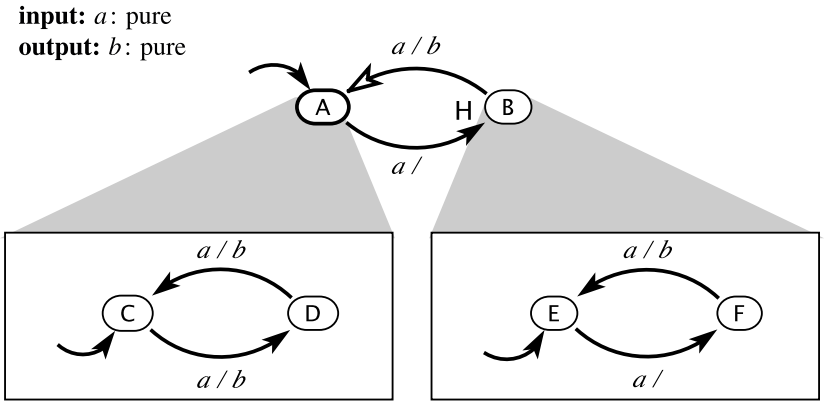
Construct a single state machine C representing the composition. Which states of the composition are unreachable?

Solution:



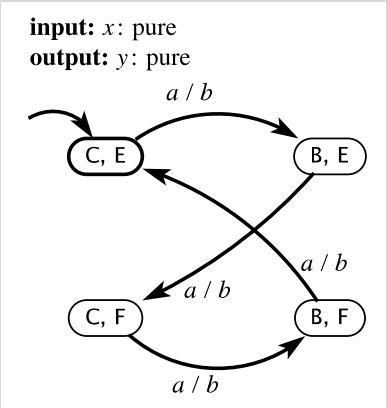
All states are reachable.

5. Consider the following hierarchical state machine:

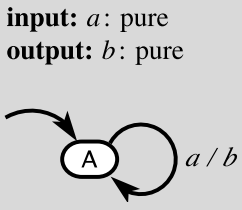


Construct an equivalent flat FSM giving the semantics of the hierarchy. Describe in words the input/output behavior of this machine. Is there a simpler machine that exhibits the same behavior? (Note that equivalence relations between state machines are considered in Chapter 14, but here, you can use intuition and just consider what the state machine does when it reacts.)

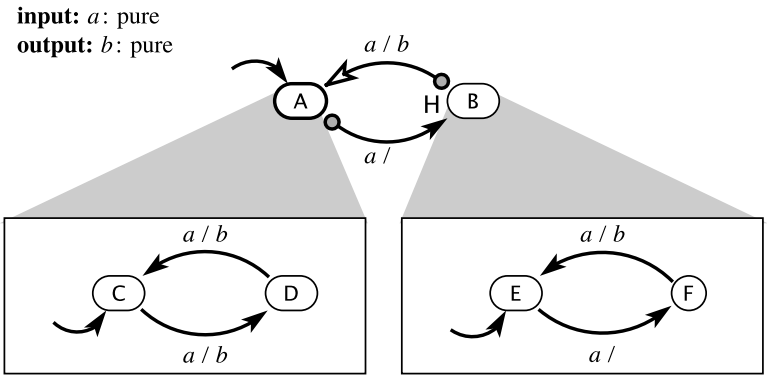
Solution: Without showing unreachable states, the flattened state machine looks like this:



The behavior of the machine is extremely simple. Given a present input a, it produces a present output b. If the input is absent, the output is absent. Thus, a simpler equivalent machine looks like this:

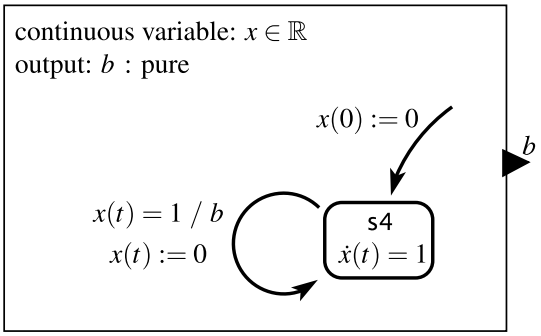


6. How many reachable states does the following state machine have?



Solution: Two. Both top-level transitions are preemptive, and the guards on those transitions will be enabled whenever a transition out of the initial state of the refinements would be enabled. So the refinements never leave their initial state.

7. Suppose that the machine of Exercise 8 of Chapter 4 is composed in a synchronous side-by-side composition with the following machine:



Find a tight lower bound on the time between events a and b. That is, find a lower bound on the time gap during which there are no events in signals a or b. Give an argument that your lower bound is tight.

Solution: The lower bound is zero. The second machine produces an event b at integer times 1, 2, 3, etc. So the question becomes, how close can the first machine get to integer times for events a? If t_1 or t_2 is an integer, then it can produce events that are simultaneous with those in b, so the gap is zero. If $t_1 = M/N$ is rational, then one possible trace produces the n-th event a at times

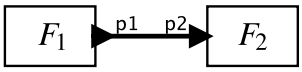
$$t_n = n + nt_1.$$

(This trace never enters state s3.) At time $n = N$ this is an integer, so the event in b is simultaneous with the one in a. If t_1 is irrational, then the above trace will never produce events at an integer time, but t_1 can be arbitrarily closely approximated by a rational number, so there is no non-zero lower bound on how close the events are to an integer time.

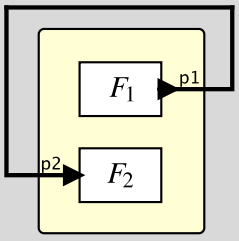
6 Concurrent Models of Computation — Exercises

1. Show how each of the following actor models can be transformed into a feedback system by using a reorganization similar to that in Figure 6.1(b). That is, the actors should be aggregated into a single side-by-side composite actor.

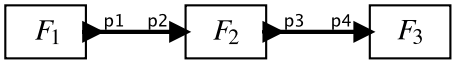
   (a)



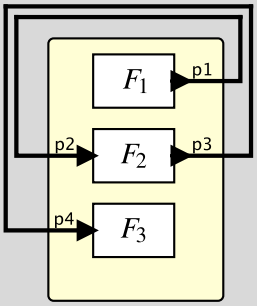
Solution:



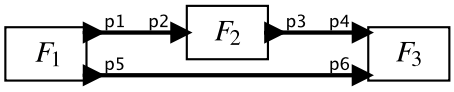
(b)



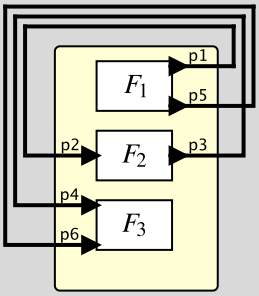
Solution:



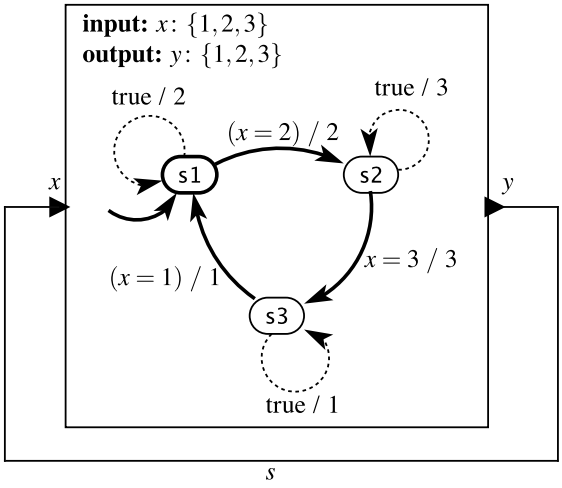
(c)



Solution:



2. Consider the following state machine in a synchronous feedback composition:



(a) Is it well-formed? Is it constructive?

Solution: Yes, it is well formed and constructive because in each state, even if the input is unknown, the output can be determined.

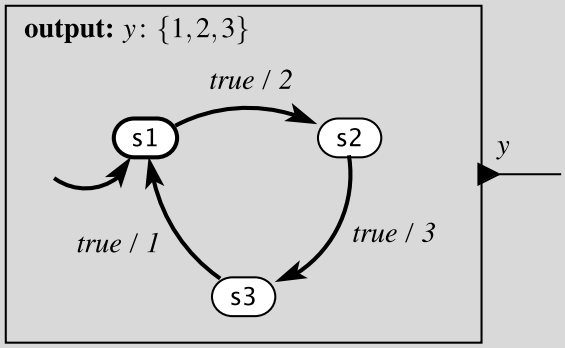
(b) If it is well-formed and constructive, then find the output symbols for the first 10 reactions. If not, explain where the problem is.

Solution: The output sequence for the first 10 reactions is

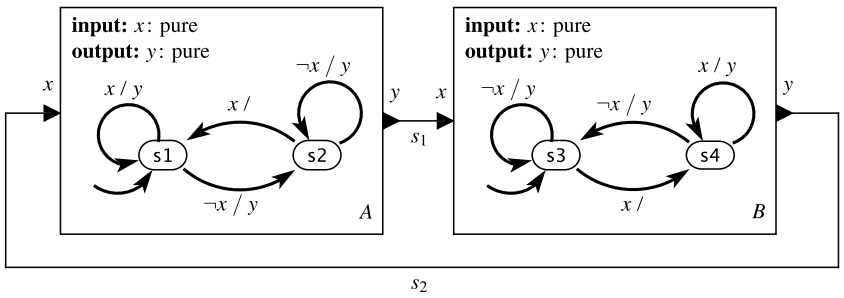
$$(2,3,1,2,3,1,2,3,1,2).$$

(c) Show the composition machine, assuming that the composition has no input and that the only output is y.

Solution:



3. For the following synchronous model, determine whether it is well formed and constructive, and if so, determine the sequence of values of the signals s_1 and s_2.



Solution: It is well formed and constructive and the sequences of values are as follows:

$$s_1 : (present, absent, present, absent, \cdots),$$

and

$$s_2 : (absent, present, absent, present, \cdots),$$

The above can be determined by enumerating reachable state of the product automaton (you do not need to construct the product automaton for this) and checking for each such state that there is a unique fixed point (FP):

(a) State (s1, s3): Unique FP is $s_2 = absent, s_1 = present$.

(b) State (s2, s4): Unique FP is $s_2 = present, s_1 = absent$.

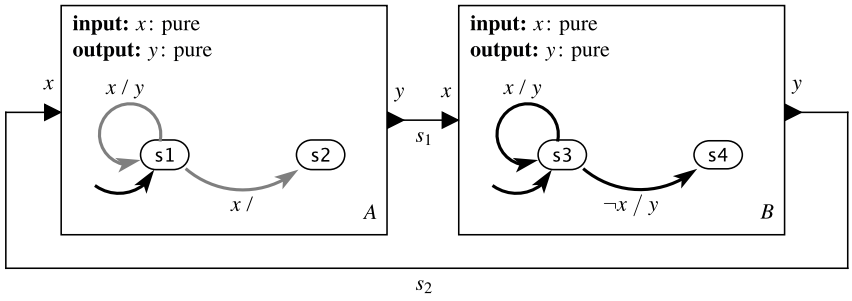
The above are the only reachable states of the composition. It is possibly instructive to examine the unreachable states:

(a) State (s1, s4): Unique FP is $s_2 = s_1 = present$.

(b) State (s2, s3): Non-unique FP, $s_2 = present, s_1 = absent$ or $s_2 = absent, s_1 = present$

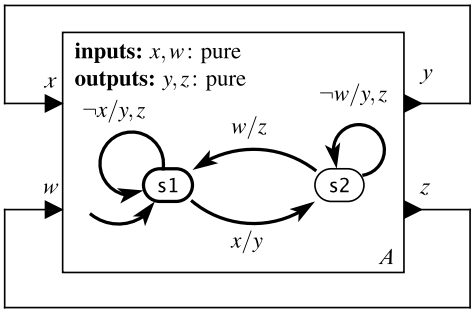
But since state (s2, s3) is not reachable, the non-unique fixed point does not make the machine ill-formed.

4. For the following synchronous model, determine whether it is well formed and constructive, and if so, determine the possible sequences of values of the signals s_1 and s_2. Note that machine A is nondeterministic.



Solution: It is well formed and constructive. The possible behavior include one where s_1 and s_2 consist of an infinite sequence of *present* events, and behaviors where s_1 consists of n consecutive *present* events and s_2 consists of $n+1$ consecutive *present* events, where $n \in \mathbb{N}$, $n \geq 1$, followed by an infinite tail of *absent* events.

5. (a) Determine whether the following synchronous model is well formed and constructive:

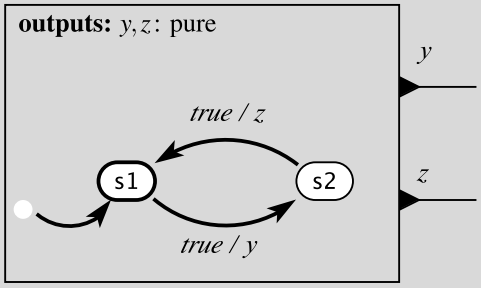


Explain.

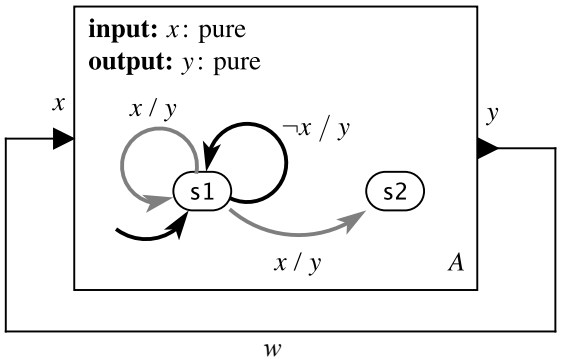
Solution: The machine is constructive, and therefore well formed. In state s1, the output y is known to be present, and therefore the input x is known to be present. Hence, every reaction in this state will transition to state s2. In s2, the output z is known to be present, and hence the input w is known to be present. Hence, the machine will transition back to state s1.

(b) For the model in part (a), give the semantics by giving an equivalent flat state machine with no inputs and two outputs.

Solution: An equivalent state machine is below:



6. Consider the following synchronous feedback composition:



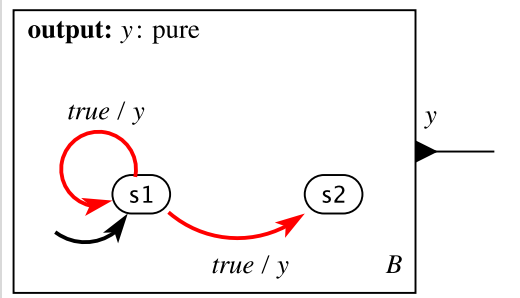
Notice that the FSM *A* is nondeterministic.

(a) Is this composition well formed? Constructive? Explain.

Solution: Yes to both. In state s1, the output is always present, and is state s2, the output is always absent. In state s1 machine will nondeterministically choose either the self loop transition or the transition to s2. If it transitions to s2, thereafter, it will stutter.

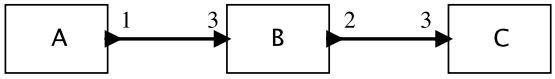
(b) Give an equivalent flat FSM (with no input and no connections) that produces exactly the same possible sequences *w*.

Solution: A solution is shown below:



7. Recall the traffic light controller of Figure 3.10. Consider connecting the outputs of this controller to a pedestrian light controller, whose FSM is given in Figure 5.10. Using your favorite modeling software that supports state machines (such as Ptolemy II, LabVIEW Statecharts, or Simulink/Stateflow), construct the composition of the above two FSMs along with a deterministic extended state machine 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.

8. Consider the following SDF model:



The numbers adjacent to the ports indicate the number of tokens produced or consumed by the actor when it fires. Answer the following questions about this model.

(a) Let q_A, q_B, and q_C denote the number of firings of actors A, B, and C, respectively. Write down the balance equations and find the least positive integer solution.

Solution:

$$\begin{aligned} q_A &= 3q_B \\ 2q_B &= 3q_C \,. \end{aligned}$$

The least positive inter solution is

$$\begin{aligned} q_A &= 9 \\ q_B &= 3 \\ q_C &= 2 \,. \end{aligned}$$

(b) Find a schedule for an unbounded execution that minimizes the buffer sizes on the two communication channels. What is the resulting size of the buffers?

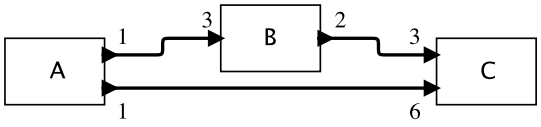
Solution: The following schedule minimizes buffer sizes:

$$\mathsf{A}, \mathsf{A}, \mathsf{A}, \mathsf{B}, \mathsf{A}, \mathsf{A}, \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{A}, \mathsf{A}, \mathsf{A}, \mathsf{B}, \mathsf{C} \,.$$

The buffer between A and B has size 3, and the buffer between B and C has size 4.

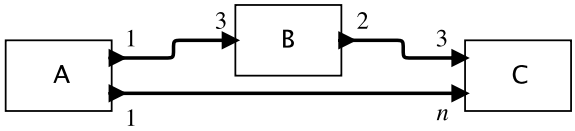
9. For each of the following dataflow models, determine whether there is an unbounded execution with bounded buffers. If there is, determine the minimum buffer size.

(a)



Solution: There is no unbounded execution with bounded buffers.

(b)



where n is some integer.

Solution: There is no unbounded execution with bounded buffers for any n because the balance equations require that

$$2q_A = 9q_C$$

due to the upper path and

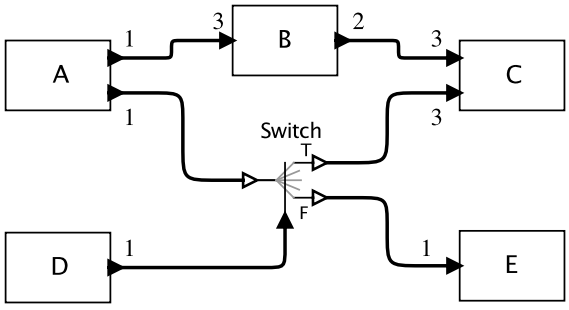
$$q_A = nq_C$$

due to the lower path. Multiplying the second equation through by 2 and subtracting the two equations yields

$$0 = (9-2n)q_C$$

which implies that either $q_C = 0$ or $9-2n = 0$. The latter yields $n = 4.5$, which is not an integer.

(c)



where D produces an arbitrary boolean sequence.

Solution: There is no unbounded execution with bounded buffers.

(d) For the same dataflow model as in part (c), assume you can specify a periodic boolean output sequence produced by D. Find such a sequence that yields bounded buffers, give a schedule that minimizes buffer sizes, and give the buffer sizes.

Solution: Let D produce the sequence

(*true, true, true, false, false, false, true, true, true*)

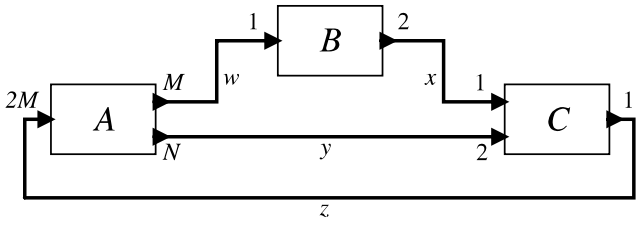
Let the schedule be

$$\mathsf{A},\mathsf{D},\mathsf{S},\mathsf{A},\mathsf{D},\mathsf{S},\mathsf{A},\mathsf{D},\mathsf{S},\mathsf{B},$$
$$\mathsf{A},\mathsf{D},\mathsf{S},\mathsf{E},\mathsf{A},\mathsf{D},\mathsf{S},\mathsf{E},\mathsf{A},\mathsf{D},\mathsf{S},\mathsf{E},\mathsf{B},\mathsf{C},$$
$$\mathsf{A},\mathsf{D},\mathsf{S},\mathsf{A},\mathsf{D},\mathsf{S},\mathsf{A},\mathsf{D},\mathsf{S},\mathsf{B},\mathsf{C}$$

where S is short for Switch. The buffer sizes are as follows:

$$\begin{aligned}
\mathsf{A} \to \mathsf{B}: &\quad 3\\
\mathsf{A} \to \mathsf{S}: &\quad 1\\
\mathsf{D} \to \mathsf{S}: &\quad 1\\
\mathsf{S} \to \mathsf{E}: &\quad 1\\
\mathsf{S} \to \mathsf{C}: &\quad 3\\
\mathsf{B} \to \mathsf{C}: &\quad 4
\end{aligned}$$

10. Consider the SDF graph shown below:



In this figure, A, B, and C are actors. Adjacent to each port is the number of tokens consumed or produced by a firing of the actor on that port, where N and M are variables with positive integer values. Assume the variables w, x, y, and z represent the number of initial tokens on the connection where these variables appear in the diagram. These variables have non-negative integer values.

(a) Derive a simple relationship between N and M such that the model is consistent, or show that no positive integer values of N and M yield a consistent model.

Solution: The balance equations are

$$\begin{aligned} Mq_A &= q_B \\ 2q_B &= q_C \\ Nq_A &= 2q_C \\ q_C &= 2Mq_A. \end{aligned}$$

From these we can determine that $N = 4M$ results in a consistent model.

(b) Assume that $w = x = y = 0$ and that the model is consistent and find the minimum value of z (as a function N and M) such that the model does not deadlock.

Solution: The solution is $z = 2M$. The minimum solution to the balance equations yields $q_A = 1$, regardless of the value of M or N (for a consistent model). The minimum number of initial tokens that enables this is $z = 2M$.

(c) Assume that $z = 0$ and that the model is consistent. Find values for w, x, and y such that the model does not deadlock and $w + x + y$ is minimized.

Solution: We need to be able to execute C at least $2M$ times to avoid deadlock. Hence $w = 0$, $x = 2M$, and $y = 4M$ will work. However, so will $w = M$, $x = 0$, and $y = 4M$. The latter has a lower value for $w + x + y$. There are no more possibilities, so this latter value is the solution.

(d) Assume that $w = x = y = 0$ and z is whatever value you found in part (b). Let b_w, b_x, b_y, and b_z be the buffer sizes for connections w, x, y, and z, respectively. What is the minimum for these buffer sizes?

Solution: The minimum positive integer solution to the balance equations is $q_A = 1$, $q_B = M$, and $q_C = 2M$. With this solution, the schedule that minimizes the buffer sizes interleaves the executions of B and C. The resulting buffer sizes are $b_w = M$, $b_x = 2$, $b_y = 4M = N$, and $b_z = 2M$.

Part II

Design of Embedded Systems

7 Sensors and Actuators — Exercises

1. Show that the composition $f \circ g$ of two affine functions f and g is affine.

Solution: Assume

$$f(x) = a_1 x + b_1$$

and

$$g(x) = a_2 x + b_2.$$

Then

$$(f \circ g)(x) = a_1(a_2 x + b_2) + b_1 = (a_1 a_2)x + (a_1 b_2 + b_1),$$

which is an affine function

$$(f \circ g)(x) = a_3 x + b_3,$$

where $a_3 = a_1 a_2$ and $b_3 = a_1 b_2 + b_1$.

2. The dynamic range of human hearing is approximately 100 decibels. Assume that the smallest difference in sound levels that humans can effectively discern is a sound pressure of about 20 μPa (micropascals).

   (a) Assuming a dynamic range of 100 decibels, what is the sound pressure of the loudest sound that humans can effectively discriminate?

Solution: We have

$$100 = 20\log_{10}\left(\frac{H-L}{p}\right)$$

where p is 20 μPa. Assume $L = 0$ (the lowest sound pressure represents no sound at all) and solve for H to get

$$H = p10^{100/20} = 2Pa.$$

   (b) Assume a perfect microphone with a range that matches the human hearing range. What is the minimum number of bits that an ADC should have to match the dynamic range of human hearing?

Solution: At 6 dB per bit, to match 100dB, we need at least 100/6 = 16.7 bits. Since fractional bits are not possible, we need at least 17 bits.

3. The following questions are about how to determine the function

$$f\colon (L,H) \to \{0,\ldots,2^B-1\},$$

for an accelerometer, which given a proper acceleration x yields a digital number $f(x)$. We will assume that x has units of "g's," where 1g is the acceleration of gravity, approximately $g = 9.8\text{meters}/\text{second}^2$.

(a) Let the bias $b \in \{0,\ldots,2^B-1\}$ be the output of the ADC when the accelerometer measures no proper acceleration. How can you measure b?

Solution: Place the accelerometer horizontally so that there is no component of gravity along the axis being measured. In theory, you could also put the accelerometer in free fall in a vacuum, but this would require a rather complicated experimental setup, and it would also require that the accelerometer not be twisting while it falls.

(b) Let $a \in \{0,\ldots,2^B-1\}$ be the *difference* in output of the ADC when the accelerometer measures 0g and 1g of acceleration. This is the ADC conversion of the sensitivity of the accelerometer. How can you measure a?

Solution: Place the accelerometer at rest so that gravity is along the axis being measured, then subtract b.

(c) Suppose you have measurements of a and b from parts (3b) and (3a). Give an affine function model for the accelerometer, assuming the proper acceleration is x in units of g's. Discuss how accurate this model is.

Solution: The affine function model is

$$f(x) = ax + b.$$

This function has two sources of inaccuracy. First, $f(x)$ can only take on integer values in the set $\{0,\ldots,2^B-1\}$, so there will be quantization errors. Second, any proper acceleration outside the measurable range will be saturated at either 0 or 2^B-1.

(d) Given a measurement $f(x)$ (under the affine model), find x, the proper acceleration in g's.

Solution:

$$x = \frac{f(x)-b}{a}.$$

(e) The process of determining a and b by measurement is called **calibration** of the sensor. Discuss why it might be useful to individually calibrate each particular accelerometer, rather than assume fixed calibration parameters a and b for a collection of accelerometers.

Solution: Sensors vary from device to device due to manufacturing variability, so even accelerometers with identical designs may exhibit different calibration parameters.

(f) Suppose you have an ideal 8-bit digital accelerometer that produces the value $f(x) = 128$ when the proper acceleration is 0g, value $f(x) = 1$ when the proper acceleration is 3g to the right, and value

$f(x) = 255$ when the proper acceleration is 3g to the left. Find the sensitivity a and bias b. What is the dynamic range (in decibels) of this accelerometer? Assume the accelerometer never yields $f(x) = 0$.

Solution: The sensitivity is $a = 127/3$ and the bias is $b = 128$. The precision is $p = 3/127 \approx 0.024$g. The range is given by $H = 3$g and $L = -3$g. The dynamic range is therefore

$$D_{dB} = 20\log_{10}(6/0.024) = 48\text{dB}.$$

4. (this problem is due to Eric Kim)

   You are a Rebel Alliance fighter pilot evading pursuit from the Galactic Empire by hovering your space ship beneath the clouds of the planet Cory. Let the positive z direction point upwards and be your ship's position relative to the ground and v be your vertical velocity. The gravitational force is strong with this planet and induces an acceleration (in a vacuum) with absolute value g. The force from air resistance is linear with respect to velocity and is equal to rv, where the drag coefficient $r \leq 0$ is a constant parameter of the model. The ship has mass M. Your engines provide a vertical force.

   (a) Let $L(t)$ be the input be the vertical lift force provided from your engines. Write down the dynamics for your ship for the position $z(t)$ and velocity $v(t)$. Ignore the scenario when your ship crashes. The right hand sides should contain $v(t)$ and $L(t)$.

   Solution: $\dot{z}(t) = v(t)$
   $\dot{v}(t) = \frac{rv(t)}{M} + \frac{L(t)}{M} - g$
   From the problem description, the coefficient r is negative so rv is the drag force applied to the ship.

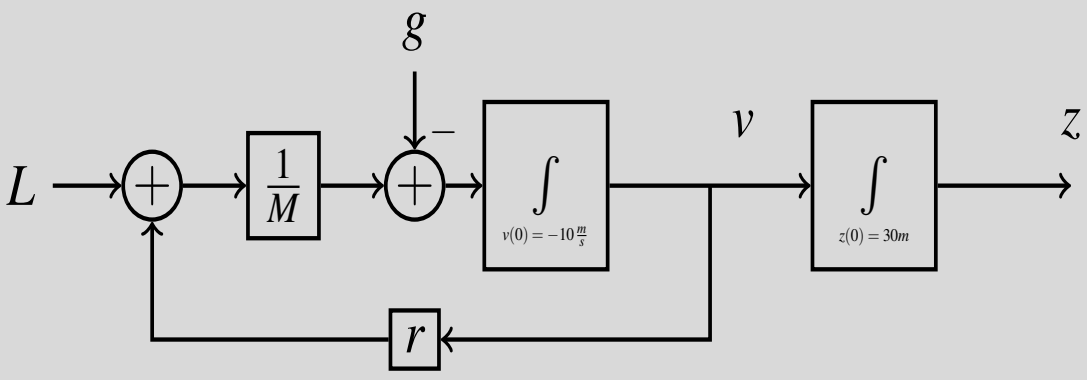
   (b) Given your answer to the previous problem, write down the explicit solution to $z(t)$ and $v(t)$ when the air resistance force is negligible and $r = 0$. At initial time $t = 0$, you are $30m$ above the ground and have an initial velocity of $-10\frac{m}{s}$. *Hint: Write $v(t)$ first then write $z(t)$ in terms of $v(t)$.*

   Solution:

$$v(t) = -10 - gt + \int_0^t \frac{L(\tau)}{M} d\tau$$
$$z(t) = 30 + \int_0^t v(\tau) d\tau$$

   (c) Draw an actor model using integrators, adders, etc. for the system that generates your vertical position and velocity. Make sure to label all variables in your actor model.

   Solution:



   (d) Your engine is slightly damaged and you can only control it by giving a pure input, switch, that when present instantaneously switches the state of the engine from on to off and vice versa. When

on, the engine creates a positive lift force L and when off $L = 0$. Your instrumentation panel contains an accelerometer. Assume your spaceship is level (i.e. zero pitch angle) and the accelerometer's positive z axis points upwards. Let the input sequence of engine switch commands be

$$\text{switch}(t) = \left\{ \begin{array}{ll} \textit{present} & \text{if } t \in \{.5, 1.5, 2.5, \ldots\} \\ \textit{absent} & \text{otherwise} \end{array} \right\}.$$

To resolve ambiguity at switching times $t = .5, 1.5, 2.5, \ldots$, at the moment of transition the engine's force takes on the new value instantaneously. Assume that air resistance is negligible (i.e. $r = 0$), ignore a crashed state, and the engine is on at $t = 0$.

Sketch the vertical component of the accelerometer reading as a function of time $t \in \mathbb{R}$. Label important values on the axes. *Hint: Sketching the graph for force first would be helpful.*

Solution: Accelerometer should have a value of $\frac{L}{m}$ on intervals $[0, .5)$ and $[k - .5, k + .5)$ for $k = 2, 4, \ldots$ and 0 on intervals $[k - .5, k + .5)$ for $k = 1, 3, \ldots$ because it is in free fall.

(e) If the spaceship is flying at a constant height, what is the value read by the accelerometer?

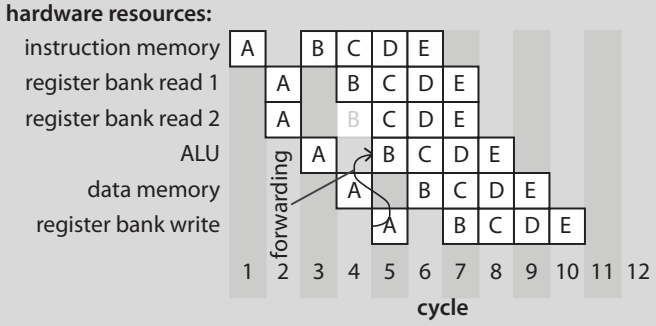
Solution: The accelerometer would read a positive g value.

8

Embedded Processors — Exercises

1. Consider the reservation table in Figure 8.4. Suppose that the processor includes forwarding logic that is able to tell that instruction *A* is writing to the same register that instruction *B* is reading from, and that therefore the result written by *A* can be forwarded directly to the ALU before the write is done. Assume the forwarding logic itself takes no time. Give the revised reservation table. How many cycles are lost to the pipeline bubble?

Solution: The revised reservation table is shown here:



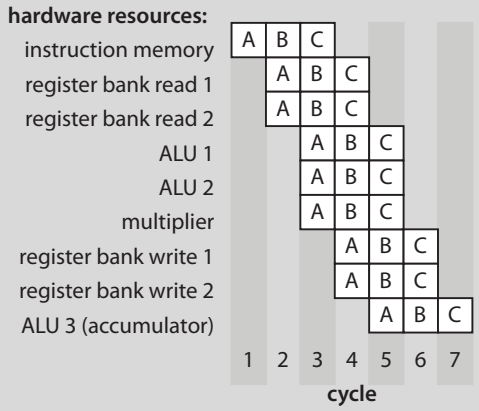
Only one cycle is lost to the pipeline bubble. Note that there is no need to do the read in B, hence it is grayed out.

2. Consider the following instruction, discussed in Example 8.6:

```
1 MAC *AR2+, *AR3+, A
```

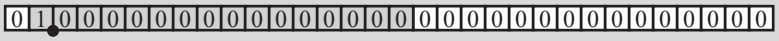
Suppose the processor has three ALUs, one for each arithmetic operation on the addresses contained in registers `AR2` and `AR3` and one to perform the addition in the `MAC` multiply-accumulate instruction. Assume these ALUs each require one clock cycle to execute. Assume that a multiplier also requires one clock cycle to execute. Assume further that the register bank supports two reads and two writes per cycle, and that the accumulator register `A` can be written separately and takes no time to write. Give a reservation table showing the execution of a sequence of such instructions.

Solution:



3. Assuming fixed-point numbers with format 1.15 as described in the boxes on pages 224 and 225, show that the *only* two numbers that cause overflow when multiplied are -1 and -1. That is, if either number is anything other than -1 in the 1.15 format, then extracting the 16 shaded bits in the boxes does not result in overflow.

Solution: Overflow occurs only if the two binary digits to the left of the binary point differ. There are two possibilities. If the digits are 01, then the product is positive. Since the numbers being multiplied are ≤ 1 in magnitude, the largest positive result is 1, which has binary representation:



All other positive results are smaller than this, and hence will have 00 to the left of the binary point. The only way to obtain a result of 1 is to multiply -1 by -1, given the 1.15 format.

The second possibility is that the two high-order bits of the result are 10. However, this would represent a negative number with magnitude strictly greater than 1, which is not a possible result when multiplying two numbers with magnitude ≤ 1. Therefore, this possibility does not occur.

Hence, the only way to get overflow is to multiply -1 by -1.

9 Memory Architectures — Exercises

1. Consider the function `compute_variance` listed below, which computes the variance of integer numbers stored in the array `data`.

```
1  int data[N];
2
3  int compute_variance() {
4    int sum1 = 0, sum2 = 0, result;
5    int i;
6
7    for(i=0; i < N; i++) {
8      sum1 += data[i];
9    }
10   sum1 /= N;
11
12   for(i=0; i < N; i++) {
13     sum2 += data[i] * data[i];
14   }
15   sum2 /= N;
16
17   result = (sum2 - sum1*sum1);
18
19   return result;
20 }
```

   Suppose this program is executing on a 32-bit processor with a direct-mapped cache with parameters $(m,S,E,B) = (32,8,1,8)$. We make the following additional assumptions:

   - An `int` is 4 bytes wide.
   - `sum1`, `sum2`, `result`, and `i` are all stored in registers.
   - `data` is stored in memory starting at address `0x0`.

   Answer the following questions:

   (a) Consider the case where `N` is 16. How many cache misses will there be?

Solution: First, note that each block stored in the cache is of size 8 bytes. Since each `int` is 4 bytes wide, we note that `data[0]` and `data[1]` will lie in the same cache block, so will `data[2]` and `data[3]`, and so on.

If `N` is 16, then we will suffer a cache miss in the first for loop on reading `data[i]` for every even `i` from 0 to 15. At the end of the first for loop, the entire array `data` will be in the cache. Thus, while executing the second for loop, we will never suffer a cache miss on any read.

Thus, the total number of cache misses is 8.

(b) Now suppose that `N` is 32. Recompute the number of cache misses.

Solution: If `N` is 32, `data[i]` and `data[i+16]` will map to the same cache block. Thus, during the first for loop, each read to `data[i+16]` will evict the block containing `data[i]`.

Thus, when we execute the second for loop, we will suffer a cache miss on each even entry in the array all over again, just as in the first for loop.

Therefore, the total number of cache misses in this case will be 16*2 = 32.

(c) Now consider executing for `N` = 16 on a 2-way set-associative cache with parameters $(m,S,E,B) = (32,8,2,4)$. In other words, the block size is halved, while there are two cache lines per set. How many cache misses would the code suffer?

Solution: The code would suffer 16 cache misses in the first for loop – one on reading each array entry. The reason the number of misses doubles is the smaller block size. In part (a), reading `data[2*i]` also moves `data[2*i+1]` into the cache, but this does not occur in the present case.

2. Recall from Section 9.2.3 that caches use the middle range of address bits as the set index and the high order bits as the tag. Why is this done? How might cache performance be affected if the middle bits were used as the tag and the high order bits were used as the set index?

Solution: The interpretation of an address as tag and set index bits shown in Section 9.2.3 is done to improve *spatial locality*.

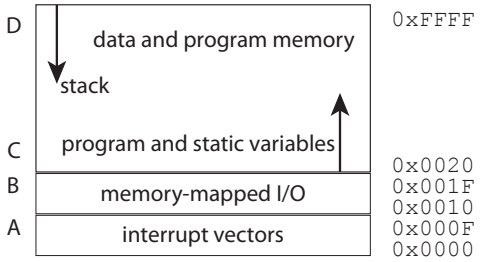
Consider a large array stored in memory.

If the high-order bits are used as the set index, then continguous array elements will map to the *same* cache set. A program reading this array sequentially has good spatial locality, but with this indexing it can only hold a block-sized portion of the array at any given time.

On the other hand, if the middle bits are used as the set index, contiguous array elements will map to different cache sets, and hence if the cache has total size C, a contiguous portion of the array of size C can be stored in the cache, thus greatly improving the number of cache hits.

3. Consider the C program and simplified memory map for a 16-bit microcontroller shown below. Assume that the stack grows from the top (area D) and that the program and static variables are stored in the bottom (area C) of the data and program memory region. Also, assume that the entire address space has physical memory associated with it.

```
1  #include <stdio.h>
2  #define FOO 0x0010
3  int n;
4  int* m;
5  void foo(int a) {
6     if (a > 0) {
7       n = n + 1;
8       foo(n);
9     }
10 }
11 int main() {
12    n = 0;
13    m = (int*)FOO;
14    foo(*m);
15    printf("n = %d\n", n);
16 }
```



You may assume that in this system, an `int` is a 16-bit number, that there is no operating system and no memory protection, and that the program has been compiled and loaded into area C of the memory.

(a) For each of the variables `n`, `m`, and `a`, indicate where in memory (region A, B, C, or D) the variable will be stored.

Solution: `n` and `m` will be in C, and `a` will be in D.

(b) Determine what the program will do if the contents at address 0x0010 is 0 upon entry.

Solution: The program will print 0 and exit.

(c) Determine what the program will do if the contents of memory location 0x0010 is 1 upon entry.

Solution: The program will overflow the stack, overwriting the program and static variables region. This will most likely cause it to start executing arbitrary data as if it were a program, which will produce highly unpredictable results.

4. Consider the following program:

```
1  int a = 2;
2  void foo(int b) {
3    printf("%d", b);
4  }
5  int main(void) {
6    foo(a);
7    a = 1;
8  }
```

Is it true or false that the value of `a` passed to `foo` will always be 2? Explain. Assume that this is the entire program, that this program is stored in persistent memory, and that the program is executed on a bare-iron microcontroller each time a reset button is pushed. p.282

Solution: False. The memory for storing `a` is initialized with the value 2 in the program image created by the compiler. Therefore, this memory location will be assigned value 2 when the program is loaded into the persistent memory. After the first run of the program, the memory location will have 1. If the reset button is pushed again, then `foo` will be passed the value 1.

10 Input and Output — Exercises

1. Similar to Example 10.6, consider a C program for an Atmel AVR that uses a UART to send 8 bytes to an RS-232 serial interface, as follows:

```
1 for(i = 0; i < 8; i++) {
2   while(!(UCSR0A & 0x20));
3   UDR0 = x[i];
4 }
```

   Assume the processor runs at 50 MHz; also assume that initially the UART is idle, so when the code begins executing, `UCSR0A & 0x20 == 0x20` is true; further, assume that the serial port is operating at 19,200 baud. How many cycles are required to execute the above code? You may assume that the `for` statement executes in three cycles (one to increment `i`, one to compare it to 8, and one to perform the conditional branch); the `while` statement executes in 2 cycles (one to compute `!(UCSR0A & 0x20)` and one to perform the conditional branch); and the assigment to `UDR0` executes in one cycle.

   Solution: The first pass through the `for` loop takes $3+2+1=6$ cycles. The second takes $2+2n+1$, where n is the number of times the `while` statement executes. After each write to `UDR0`, the UART needs to send 8 bits before `!(UCSR0A & 0x20)` will become true. At 19,200 baud, it takes $8/19,200$ seconds to do this, which is $50,000,000 \times 8/19,200 = 20,834$ cycles (rounding up). Thus, n must be large enough so that $3+2(n-1) >= 20,834$ (this is $n-1$ not n because the last execution of the `while` occurs after `!(UCSR0A & 0x20)` becomes false. The smallest integer value of n satisfying this is $10,417$. Hence, the second pass through the `for` loop takes $2+2n+1 = 20,837$. The remaining 6 passes through the `for` loop take the same amount of time, so the total time is $6+7\times 20,837 = 145,865$.

   However, RS232 requires a start bit and at least one stop bit, so the UART needs to send 10 bits rather than 8. This makes the time $10/19,200$ sec, giving $26,042$ cycles.

   In practice, these numbers are approximate because the architecture may introduce variability in the timing. For example, instructions may not be in cache, and the cache penalty could be substantial. Moreover, writing to `UDR0` and reading `UCSR0A` may involve transactions over the processor bus, and

there may be other activities competing for the bus (e.g., other I/O activities or DMA). Hence, this number should be interpreted as a lower bound.

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;

// Interrupt service routine.
SIGNAL(SIG_OUTPUT_COMPARE1A) {

  if(timer_count > 0) {
    timer_count--;                    A
  }
}

// Main program.
int main(void) {
  // Set up interrupts to occur
  // once per second.
  ...

  // Start a 3 second timer.
  timer_count = 3;                    B

  // Do something repeatedly
   // for 3 seconds.
  while(timer_count > 0) {
    foo();                            C
  }
}
```

Figure 10.1: Sketch of a C program that performs some function by calling procedure foo() repeatedly for 3 seconds, using a timer interrupt to determine when to stop.

2. Figure 10.1 gives the sketch of a program for an Atmel AVR microcontroller that performs some function repeatedly for three seconds. The function is invoked by calling the procedure foo(). The program begins by setting up a timer interrupt to occur once per second (the code to do this setup is not shown). Each time the interrupt occurs, the specified interrupt service routine is called. That routine decrements a counter until the counter reaches zero. The main() procedure initializes the counter with value 3 and then invokes foo() until the counter reaches zero.

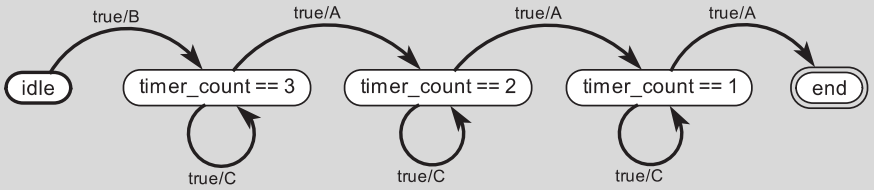
   (a) We wish to assume that the segments of code in the grey boxes, labeled **A**, **B**, and **C**, are atomic. State conditions that make this assumption valid.

   Solution: Assumptions needed for atomicity: (1) Interrupts are disabled while the interrupt service routine is executing. (2) The method foo() does not read or write the variable timer_count. Note that the invocation of the method foo() is most likely not atomic (unless it also disables interrupts, which would be an equally valid assumption). The interrupt could occur during the execution of foo(). It is not constrained to occur between executions of foo(). However, if foo() does not read or write timer_count, then it will behave as if it were atomic.

   (b) Construct a state machine model for this program, assuming as in part (a) that **A**, **B**, and **C**, are atomic. The transitions in your state machine should be labeled with "guard/action", where the

action can be any of **A**, **B**, **C**, or nothing. The actions **A**, **B**, or **C** should correspond to the sections of code in the grey boxes with the corresponding labels. You may assume these actions are atomic.

Solution: One possible model for this program is shown below:



The state with the bold outline is the initial state and the one with a double outline is the final state. The intermediate states represent the three possible values of the timer_count variable. Each of these states has two transitions out of it, both guarded by the expression "true."

(c) Is your state machine deterministic? What does it tell you about how many times foo() may be invoked? Do all the possible behaviors of your model correspond to what the programmer likely intended?

Solution: The state machine is nondeterministic. The state machine shows that this program could result in any number of invocations of foo(), including zero. Zero executions of foo() is almost certainly a behavior that the programmer did not intend.

Note that there are many possible answers. Simple models are preferred over elaborate ones, and complete ones (where everything is defined) over incomplete ones. Feel free to give more than one model.

3. In a manner similar to example 10.8, create a C program for the ARM Cortex™ - M3 to use the SysTick timer to invoke a system-clock ISR with a jiffy interval of 10 ms that records the time since system start in a 32-bit int. How long can this program run before your clock overflows?

Solution:

```
1  volatile int32_t timerCount = 0;
2  void countSince() {
3      timerCount++;
4  }
5  void main() {
6      SysTickPeriodSet(SysCtlClockGet()/100);
7      SysTickIntRegister(&countSince);
8      SysTickEnable();
9      SysTickIntEnable();
10     ...
11 }
```

The largest positive number that fits in an `int32_t` data type is $2^{31} - 1 = 2{,}147{,}483{,}647$. Multiplying this by 10 ms indicates that the clock will overflow after about 249 days.

4. Consider a dashboard display that displays "normal" when brakes in the car operate normally and "emergency" when there is a failure. The intended behavior is that once "emergency" has been displayed, "normal" will not again be displayed. That is, "emergency" remains on the display until the system is reset.

   In the following code, assume that the variable `display` defines what is displayed. Whatever its value, that is what appears on the dashboard.

```
1  volatile static uint8_t alerted;
2  volatile static char* display;
3  void ISRA() {
4      if (alerted == 0) {
5          display = "normal";
6      }
7  }
8  void ISRB() {
9      display = "emergency";
10     alerted = 1;
11 }
12 void main() {
13     alerted = 0;
14     ...set up interrupts...
15     ...enable interrupts...
16     ...
17 }
```

   Assume that `ISRA` is an interrupt service routine that is invoked when the brakes are applied by the driver. Assume that `ISRB` is invoked if a sensor indicates that the brakes are being applied at the same time that the accelerator pedal is depressed. Assume that neither ISR can interrupt itself, but that `ISRB` has higher priority than `ISRA`, and hence `ISRB` can interrupt `ISRA`, but `ISRA` cannot interrupt `ISRB`. Assume further (unrealistically) that each line of code is atomic.

   (a) Does this program always exhibit the intended behavior? Explain. In the remaining parts of this problem, you will construct various models that will either demonstrate that the behavior is correct or will illustrate how it can be incorrect.

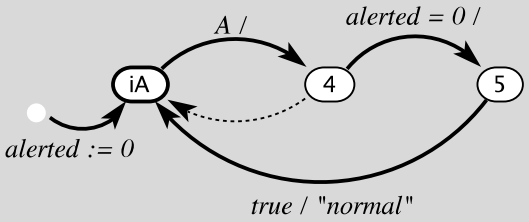
Solution: No, it does not. If `ISRA` is interrupted after executing line 4 and before executing line 5 by `ISRB`, then the "emergency" display will be quickly overwritten by "normal."

   (b) Construct a determinate extended state machine modeling `ISRA`. Assume that:

   - `alerted` is a variable of type $\{0, 1\} \subset$ `uint8_t`,
   - there is a pure input A that when present indicates an interrupt request for `ISRA`, and
   - `display` is an output of type `char*`.

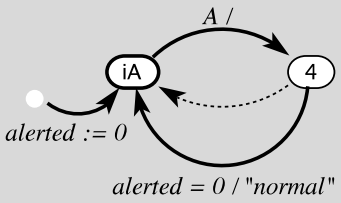
Solution: In the following, iA indicates that the ISR is idle, 4 indicates that the program is about to execute line 4, and 5 indicates that it is about to execute line 5.

variable: *alerted*: $\{0,1\}$
input: A: pure
output: *display*: `char*`



It may be tempting to use a simpler model like that shown below:

variable: *alerted*: $\{0,1\}$
input: A: pure
output: *display*: `char*`



but this model will not exhibit the bug identified in part (a).

(c) Give the size of the state space for your solution.

Solution: There are three bubbles and the variable `alerted` has two possible values, so the state space has size 6. Note however that `alerted` is never changed to 1 by this state machine, so considering this state machine in isolation, only three of the six states are reachable.

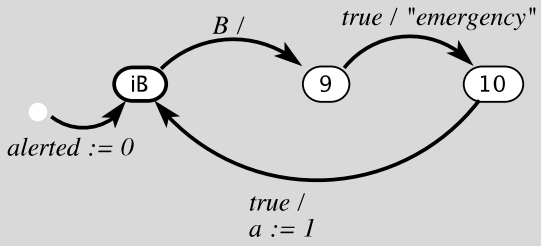
(d) Explain your assumptions about when the state machine in (b) reacts. Is this time triggered, event triggered, or neither?

Solution: The machine reacts each time the input A is present, and the subsequently when the processor executes one line of code. This could be time triggered if each line of code executes in a fixed time unit, but this is unlikely. Hence, it is neither event triggered nor time triggered.

(e) Construct a determinate extended state machine modeling `ISRB`. This one has a pure input B that when present indicates an interrupt request for `ISRB`.

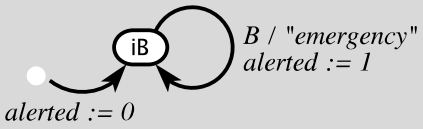
Solution: The following state machine models `ISRB` in a manner similar to the model we constructed for `ISRA`:

variable: *alerted*: $\{0,1\}$
input: B: pure
output: *display*: `char*`



However, in this case, the model really does not need to be this detailed. Since `ISRB` cannot be interrupted, we can model its execution as a single atomic operation, arriving at the very simple model shown below:

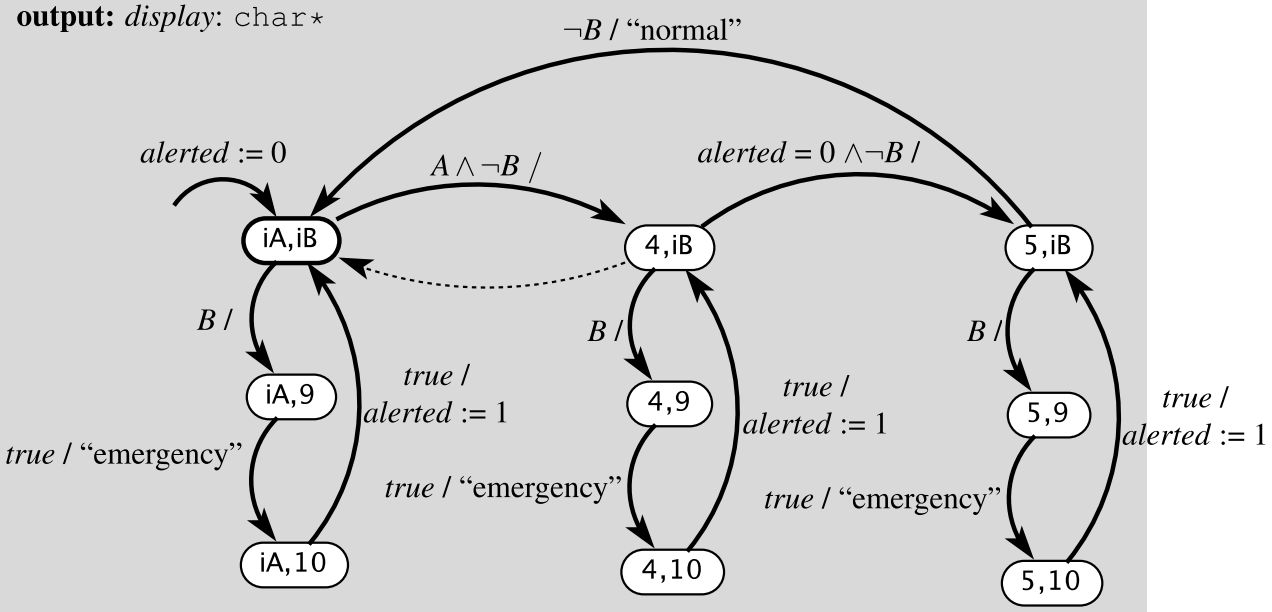
variable: *alerted*: $\{0,1\}$
input: B: pure
output: *display*: `char*`



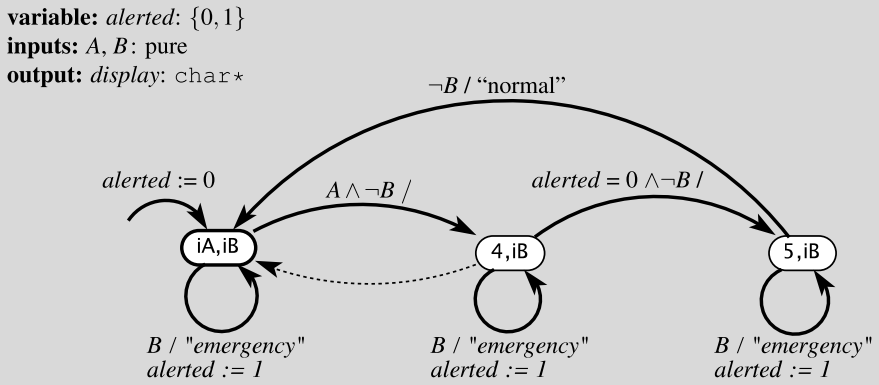
(f) Construct a flat (non-hierarchical) determinate extended state machine describing the joint operation of the these two ISRs. Use your model to argue the correctness of your answer to part (a).

Solution: Using the more complex model of `ISRB` above, we arrive at the following:

variable: *alerted*: $\{0,1\}$
inputs: A, B: pure
output: *display*: `char*`



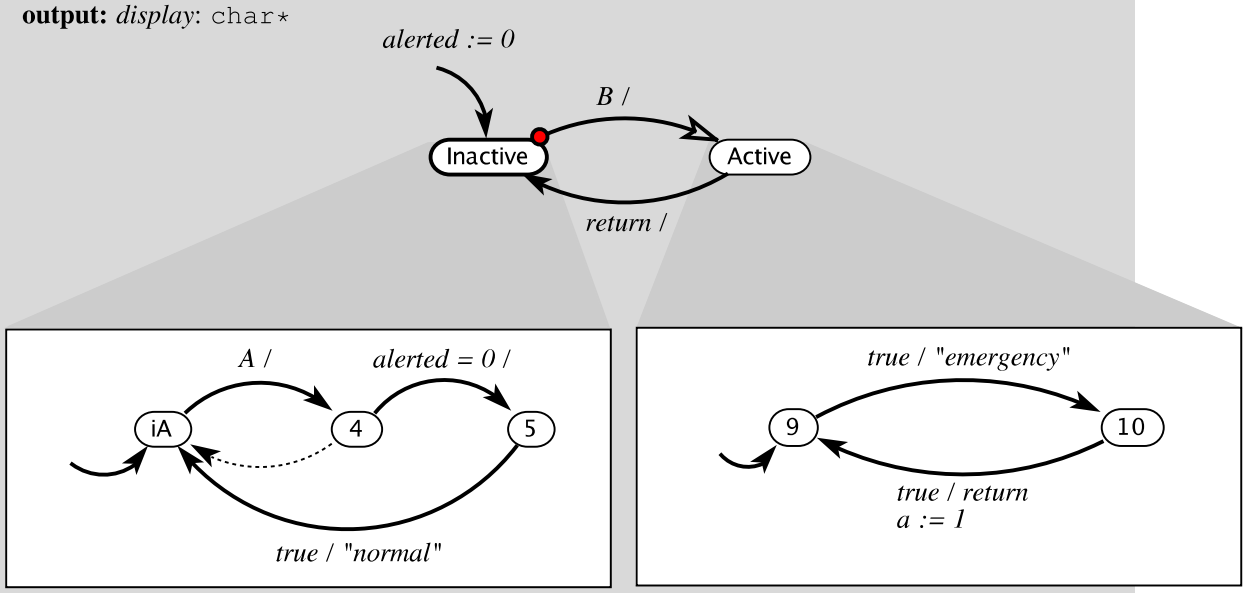
Using the simpler model of ISRB above, we arrive at the following:



This model shows clearly the bug in part (a). Any time that the self-loop on the right-most state is taken, the bug will occur.

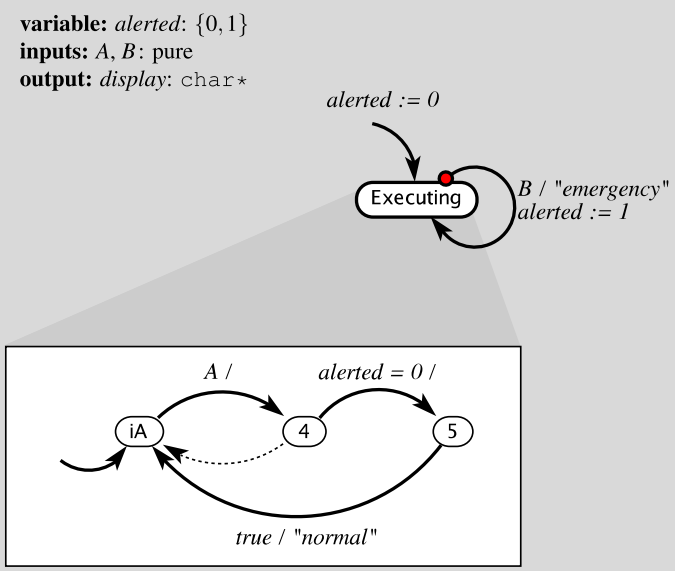
(g) Give an equivalent hierarchical state machine. Use your model to argue the correctness of your answer to part (a).

Solution: Using the more complex model of ISRB above, we arrive at the following:



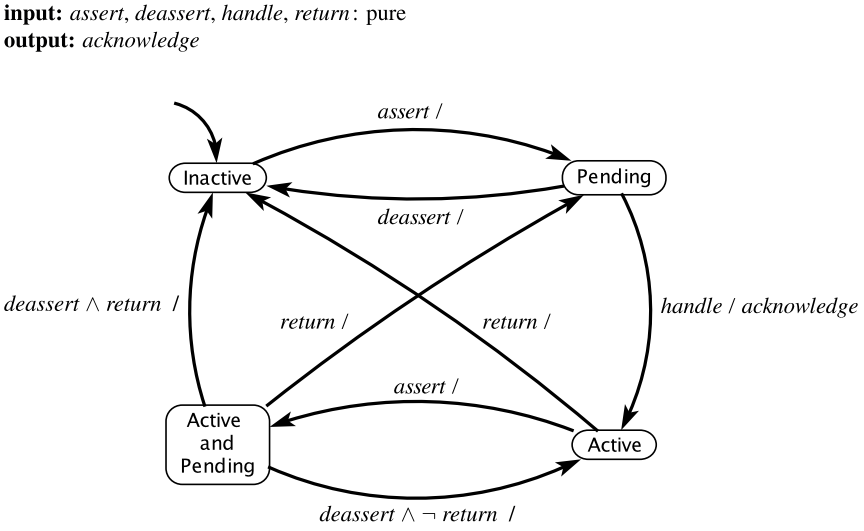
Here, we assume that the *return* output of the Active refinement is visible *in the same reaction* to the upper state machine. Note that the transition to the right is a reset transition, and the one to the left is a history transition.

Using the simpler model of ISRB above, we arrive at the following:



If the preemptive transition is taken while the refinement is in state 5, then the bug from part (a) will occur.

5. Suppose a processor handles interrupts as specified by the following FSM:



Here, we assume a more complicated interrupt controller than that considered in Example 10.12, where there are several possible interrupts and an arbiter that decides which interrupt to service. The above state machine shows the state of one interrupt. When the interrupt is asserted, the FSM transitions to the Pending state, and remains there until the arbiter provides a *handle* input. At that time, the FSM transitions to the Active state and produces an *acknowledge* output. If another interrupt is asserted while in the Active state, then it transitions to Active and Pending. When the ISR returns, the input *return* causes a transition to either Inactive or Pending, depending on the starting point. The *deassert* input allows external hardware to cancel an interrupt request before it gets serviced.

Answer the following questions.

(a) If the state is Pending and the input is *return*, what is the reaction?

Solution: The FSM remains in state Pending.

(b) If the state is Active and the input is *assert* $\wedge$ *deassert*, what is the reaction?

Solution: The machine moves to Active and Pending.

(c) Suppose the state is Inactive and the input sequence in three successive reactions is:

i. *assert* ,

ii. *deassert* $\wedge$ *handle* ,

iii. *return* .

What are all the possible states after reacting to these inputs? Was the interrupt handled or not?

Solution: The only possible state is Inactive. The ISR may or may not have executed. The fact that the input sequence includes *return* suggests that it was, but if the instruction set permits an erroneous program to issue a "return from interrupt" instruction even if not in an ISR, then the ISR may not have executed. We do not have enough information to be sure.

(d) Suppose that an input sequence never includes *deassert*. Is it true that every *assert* input causes an *acknowledge* output? In other words, is every interrupt request serviced? If yes, give a proof. If no, give a counterexample.

Solution: No. If the state is Active and Pending, then any *assert* input is ignored.

6. Suppose you are designing a processor that will support two interrupts whose logic is given by the FSM in Exercise 5. Design an FSM giving the logic of an arbiter that assigns one of these two interrupts higher priority than the other. The inputs should be the following pure signals:

$$assert1, return1, assert2, return2$$

to indicate requests and return from interrupt for interrupts 1 and 2, respectively. The outputs should be pure signals *handle1* and *handle2*. Assuming the *assert* inputs are generated by two state machines like that in Exercise 5, can you be sure that this arbiter will handle every request that is made? Justify your answer.

Solution:

7. Consider the following program that monitors two sensors. Here `sensor1` and `sensor2` denote the variables storing the readouts from two sensors. The actual read is performed by the functions `readSensor1()` and `readSensor2()`, respectively, which are called in the interrupt service routine `ISR`.

```
1  char flag = 0;
2  volatile char* display;
3  volatile short sensor1, sensor2;
4
5  void ISR() {
6    if (flag) {
7       sensor1 = readSensor1();
8    } else {
9       sensor2 = readSensor2();
10   }
11 }
12
13 int main() {
14   // ... set up interrupts ...
15   // ... enable interrupts ...
16   while(1) {
17      if (flag) {
18         if isFaulty2(sensor2) {
19            display = "Sensor2 Faulty";
20         }
21      } else {
22         if isFaulty1(sensor1) {
23            display = "Sensor1 Faulty";
24         }
25      }
26      flag = !flag;
27   }
28 }
```

Functions `isFaulty1()` and `isFaulty2()` check the sensor readings for any discrepancies, returning 1 if there is a fault and 0 otherwise. Assume that the variable `display` defines what is shown on the monitor to alert a human operator about faults. Also, you may assume that `flag` is modified only in the body of `main`.

Answer the following questions:

(a) Is it possible for the `ISR` to update the value of `sensor1` while the main function is checking whether `sensor1` is faulty? Why or why not?

Solution: No, because of `flag`. During the entire time that the main function is checking whether `sensor1` is faulty the value of `flag` must be non-zero. Hence, if an interrupt occurs during that time, `ISR` will update `sensor2`, not `sensor1`.

(b) Suppose a spurious error occurs that causes `sensor1` or `sensor2` to be a faulty value for one measurement. Is it possible for that this code would not report "Sensor1 faulty" or "Sensor2 faulty"?

Solution: Yes. It is possible for `ISR` to be invoked twice in a row before the main function gets a chance to check the value of a sampled sensor value, thus overwriting that value before `main` has checked it.

(c) Assuming the interrupt source for `ISR()` is timer-driven, what conditions would cause this code to never check whether the sensors are faulty?

Solution: If the interrupts occur so frequently that the main thread never gets to execute, and infinitely many interrupts occur, then the program will never check whether the sensors are faulty.

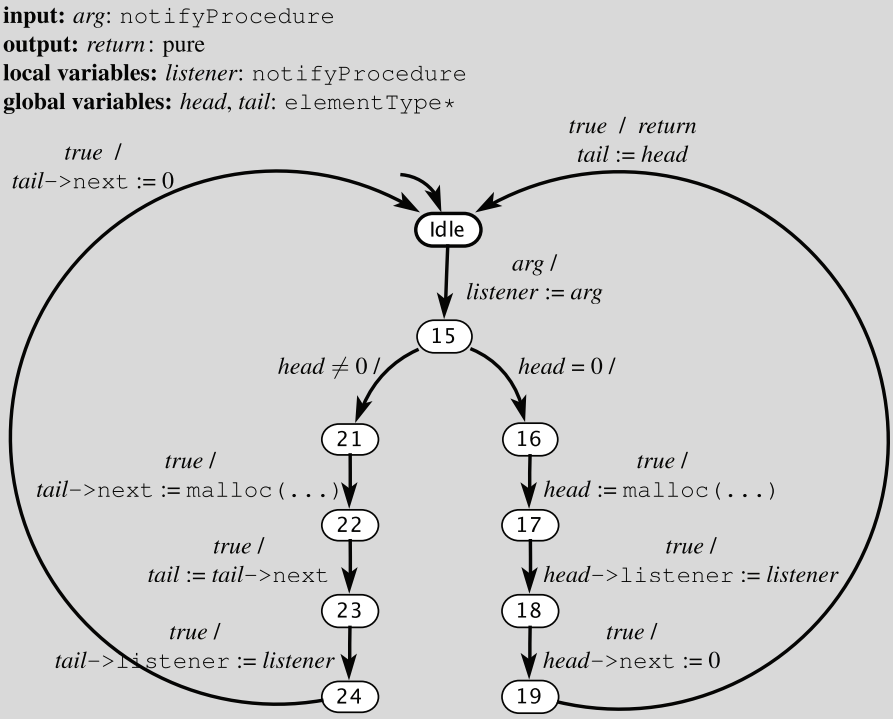
(d) Suppose that instead being interrupt driven, `ISR` and `main` are executed concurrently, each in its own thread. Assume a microkernel that can interrupt any thread at any time and switch contexts to execute another thread. In this scenario, is it possible for the `ISR` to update the value of `sensor1` while the main function is checking whether `sensor1` is faulty? Why or why not?

Solution: Yes. Suppose that `main` is checking `sensor2`, so `flag` is non-zero, and the thread scheduler interrupts it and begins executing `ISR`. Suppose that `ISR` checks the value of `flag`, reaching line 7, and then gets interrupted before executing line 7. Suppose that `main` continues executing, changing the value of `flag` to zero, and then begins checking `sensor1`. While checking `sensor1`, `main` could be interrupted again, at which point `ISR` could resume by executing line 7, updating the value of `sensor1`.

11 Multitasking

1. Give an extended state-machine model of the `addListener` procedure in Figure 11.2 similar to that in Figure 11.3,

Solution:



2. Suppose that two `int` global variables `a` and `b` are shared among several threads. Suppose that `lock_a` and `lock_b` are two mutex locks that guard access to `a` and `b`. Suppose you cannot assume that reads and writes of `int` global variables are atomic. Consider the following code:

```
1   int a, b;
2   pthread_mutex_t lock_a
3       = PTHREAD_MUTEX_INITIALIZER;
4   pthread_mutex_t lock_b
5       = PTHREAD_MUTEX_INITIALIZER;
6
7   void procedure1(int arg) {
8     pthread_mutex_lock(&lock_a);
9     if (a == arg) {
10       procedure2(arg);
11    }
12    pthread_mutex_unlock(&lock_a);
13  }
14
15  void procedure2(int arg) {
16    pthread_mutex_lock(&lock_b);
17    b = arg;
18    pthread_mutex_unlock(&lock_b);
19  }
```

Suppose that to ensure that deadlocks do not occur, the development team has agreed that `lock_b` should always be acquired before `lock_a` by any thread that acquires both locks. Note that the code listed above is not the only code in the program. Moreover, for performance reasons, the team insists that no lock be acquired unnecessarily. Consequently, it would not be acceptable to modify `procedure1` as follows:

```
1   void procedure1(int arg) {
2     pthread_mutex_lock(&lock_b);
3     pthread_mutex_lock(&lock_a);
4     if (a == arg) {
5        procedure2(arg);
6     }
7     pthread_mutex_unlock(&lock_a);
8     pthread_mutex_unlock(&lock_b);
9   }
```

A thread calling `procedure1` will acquire `lock_b` unnecessarily when `a` is not equal to `arg`. [1] Give a design for `procedure1` that minimizes unnecessary acquisitions of `lock_b`. Does your solution eliminate unnecessary acquisitions of `lock_b`? Is there any solution that does this?

Solution: Here is a candidate solution:

```
1   void procedure1(int arg) {
2     int result;
3     pthread_mutex_lock(&lock_a);
4     result = (a == arg);
5     pthread_mutex_unlock(&lock_a);
6
7     if (result) {
8       pthread_mutex_lock(&lock_b);
9       pthread_mutex_lock(&lock_a);
10      if (a == arg) {
```

[1] In some thread libraries, such code is actually incorrect, in that a thread will block trying to acquire a lock it already holds. But we assume for this problem that if a thread attempts to acquire a lock it already holds, then it is immediately granted the lock.

```
11          procedure2(arg);
12        }
13        pthread_mutex_unlock(&lock_a);
14        pthread_mutex_unlock(&lock_b);
15      }
16    }
```

This code avoids acquiring `lock_b` if `a != arg` at the time of the first test. Notice that the first test for `a == arg` needs to be performed while holding `lock_a` because reading `a` is not assured of being atomic. Moreover, there is no need to acquire `lock_b` at that point. The second test for `a == arg` is necessary because the value of `a` may have changed after the first test. Also notice that this solution may acquire `lock_b` unnecessarily. In particular, a thread could acquire `lock_b`, then get suspended, and while suspended, the value of `a` may change so that the second test for `a == arg` yields false.

3. The implementation of `get` in Figure 11.6 permits there to be more than one thread calling `get`.

   However, if we change the code on lines 30-32 to:

```
1       if (size == 0) {
2         pthread_cond_wait(&sent, &mutex);
3       }
```

   then this code would only work if two conditions are satisfied:

   - `pthread_cond_wait` returns *only* if there is a matching call to `pthread_cond_signal`, and
   - there is only one consumer thread.

   Explain why the second condition is required.

Solution: Because of the first condition, we would expect that after line 2 above, it must be true that `size != 0`. However, this is not necessarily true if there are two or more consumer threads. Recall that `pthread_cond_wait` temporarily releases the lock on `mutex`, so it could be possible for line 1 above to return true in thread *A*, which will then block on line 2, and while it is blocked, another consumer thread *B* manages to consume the next message sent without executing `pthread_cond_wait`. This could happen if *B* happens to execute line 1 above after a producer has inserted one message into the queue, but before *A* has been scheduled to respond to the signal. When *A* responds to the signal, it will wake up at line 2 above and proceed to consume a message, but there is no message on the queue! Disaster.

4. The producer/consumer pattern implementation in Example 11.13 has the drawback that the size of the queue used to buffer messages is unbounded. A program could fail by exhausting all available memory (which will cause `malloc` to fail). Construct a variant of the `send` and `get` procedures of Figure 11.6 that limits the buffer size to 5 messages.

Solution: The following definitions solve the problem with an additional condition variable.

```
1  int size = 0;
2  pthread_cond_t sent = PTHREAD_COND_INITIALIZER;
3  pthread_cond_t received = PTHREAD_COND_INITIALIZER;
4
5  // Procedure to send a message.
6  void send(int message) {
7      pthread_mutex_lock(&mutex);
8      while (size >= 5) {
9          pthread_cond_wait(&received, &mutex);
10     }
11     if (head == 0) {
12         head = malloc(sizeof(element_t));
13         head->payload = message;
14         head->next = 0;
15         tail = head;
16     } else {
17         tail->next = malloc(sizeof(element_t));
18         tail = tail->next;
19         tail->payload = message;
20         tail->next = 0;
21     }
22     size++;
23     pthread_cond_signal(&sent);
24     pthread_mutex_unlock(&mutex);
25 }
26
27 // Procedure to get a message.
28 int get() {
29     element_t* element;
30     int result;
31     pthread_mutex_lock(&mutex);
32     // Wait until the size is non-zero.
33     while (size == 0) {
34         pthread_cond_wait(&sent, &mutex);
35     }
36     result = head->payload;
37     element = head;
38     head = head->next;
39     free(element);
40     size--;
41     if (size < 5) {
42         pthread_cond_signal(&received);
43     }
44     pthread_mutex_unlock(&mutex);
45     return result;
46 }
```

5. An alternative form of message passing called rendezvous is similar to the producer/consumer pattern of Example 11.13, but it synchronizes the producer and consumer more tightly. In particular, in Example 11.13, the `send` procedure returns immediately, regardless of whether there is any consumer thread ready to receive the message. In a rendezvous-style communication, the `send` procedure will not return until a consumer thread has reached a corresponding call to `get`. Consequently, no buffering of the messages is needed. Construct implementations of `send` and `get` that implement such a rendezvous.

Solution:

```
1  // Condition variables to signal ready and complete.
2  int send_ready = 0;
3  pthread_cond_t send_ready_c = PTHREAD_COND_INITIALIZER;
4  int receive_ready = 0;
5  pthread_cond_t receive_ready_c = PTHREAD_COND_INITIALIZER;
6  int complete = 1;
7  pthread_cond_t complete_c = PTHREAD_COND_INITIALIZER;
8
9  // Procedure to send a message.
10 void send(int message) {
11     pthread_mutex_lock(&mutex);
12     complete = 0;
13     // Wait until the receiver is ready.
14     while (receive_ready == 0) {
15         pthread_cond_wait(&receive_ready_c, &mutex);
16     }
17     buffer = message;
18     send_ready = 1;
19     pthread_cond_signal(&send_ready_c);
20     // Wait until communication completes.
21     while (complete == 0) {
22         pthread_cond_wait(&complete_c, &mutex);
23     }
24     pthread_mutex_unlock(&mutex);
25 }
26
27 // Procedure to get a message.
28 int get() {
29     int result;
30     pthread_mutex_lock(&mutex);
31     // Signal that the receiver is ready.
32     receive_ready = 1;
33     pthread_cond_signal(&receive_ready_c);
34     // Wait until the sender is ready.
35     while (send_ready == 0) {
36         pthread_cond_wait(&send_ready_c, &mutex);
37     }
38     receive_ready = 0;
39     result = buffer;
40     send_ready = 0;
41     complete = 1;
42     pthread_cond_signal(&complete_c);
43     pthread_mutex_unlock(&mutex);
44     return result;
45 }
```

6. Consider the following code.

```
1  int x = 0;
2  int a;
3  pthread_mutex_t lock_a = PTHREAD_MUTEX_INITIALIZER;
4  pthread_cond_t go = PTHREAD_COND_INITIALIZER; // used in part c
5
6  void proc1(){
7      pthread_mutex_lock(&lock_a);
8      a = 1;
9      pthread_mutex_unlock(&lock_a);
10     <proc3>(); // call to either proc3a or proc3b
11                // depending on the question
12 }
13
14 void proc2(){
15     pthread_mutex_lock(&lock_a);
16     a = 0;
17     pthread_mutex_unlock(&lock_a);
18     <proc3>();
19 }
20
21 void proc3a(){
22     if(a == 0){
23         x = x + 1;
24     } else {
25         x = x - 1;
26     }
27 }
28
29 void proc3b(){
30     pthread_mutex_lock(&lock_a);
31     if(a == 0){
32         x = x + 1;
33     } else {
34         x = x - 1;
35     }
36     pthread_mutex_unlock(&lock_a);
37 }
```

Suppose `proc1` and `proc2` run in two separate threads and that each procedure is called in its respective thread exactly once. Variables `x` and `a` are global and shared between threads and `x` is initialized to 0. Further, assume the increment and decrement operations are atomic.

The calls to `proc3` in `proc1` and `proc2` should be replaced with calls to `proc3a` and `proc3b` depending on the part of the question.

(a) If `proc1` and `proc2` call `proc3a` in lines 10 and 18, is the final value of global variable `x` guaranteed to be 0? Justify your answer.

Solution: No. As a counterexample let proc1 execute first until it is preempted between lines 9 and 10. proc2 executes next to completion and modifies the value of x to 1 and the value of a to 0. Now resume execution for proc1 *but with a still set to 0.* x is again incremented and finishes execution with value 2.

(b) What if `proc1` and `proc2` call `proc3b`? Justify your answer.

Solution: No. The same counterexample from part (a) applies here.

(c) With `proc1` and `proc2` still calling `proc3b`, modify `proc1` and `proc2` with condition variable `go` to guarantee the final value of `x` is 2. Specifically, give the lines where `pthread_cond_wait` and `pthread_cond_signal` should be inserted into the code listing. Justify your answer briefly. Make the assumption that `proc1` acquires lock_a before `proc2`.

Also recall that

```
pthread_cond_wait(&go, &lock_a);
```

will temporarily release `lock_a` and block the calling thread until

```
pthread_cond_signal(&go);
```

is called in another thread, at which point the waiting thread will be unblocked and reacquire `lock_a`.

Solution: The solution is to use the condition variable to force the execution of proc1 and proc2 to follow the counterexample from part (a).

`pthread_cond_wait(&go, &lock_a);` should be inserted between lines 8 and 9 and `pthread_cond_signal(&go);` should be inserted between lines 16 and 17. There are other valid placements as well.

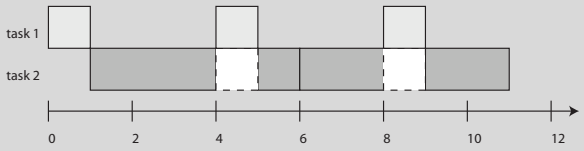
(This problem is due to Matt Weber.)

12 Scheduling — Exercises

1. This problem studies fixed-priority scheduling. Consider two tasks to be executed periodically on a single processor, where task 1 has period $p_1 = 4$ and task 2 has period $p_2 = 6$. p.314

   (a) Let the execution time of task 1 be $e_1 = 1$. Find the maximum value for the execution time e_2 of task 2 such that the RM schedule is feasible.

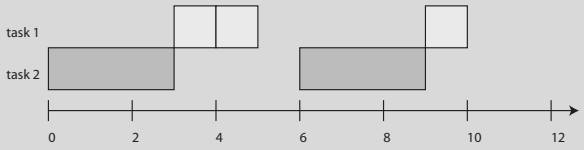
   Solution: The largest execution time for task 2 is $e_2 = 4$. The following figure shows the resulting schedule:

   The schedule repeats every 12 time units.

   (b) Again let the execution time of task 1 be $e_1 = 1$. Let non-RMS be a fixed-priority schedule that is not an RM schedule. Find the maximum value for the execution time e_2 of task 2 such that non-RMS is feasible.

   Solution: The largest execution time for task 2 is $e_2 = 3$. The following figure shows the resulting schedule:

   The schedule repeats every 12 time units.

   (c) For both your solutions to (a) and (b) above, find the processor utilization. Which is better?

Solution: From the figures above, we see that the RM schedule results in the machine being idle for 1 out of 12 time units, so the utilization is 11/12. The non-RM schedule results in the machine being idle for 3 out of 12 time units, so the utilization is 9/12 or 3/4. The RM schedule is better.

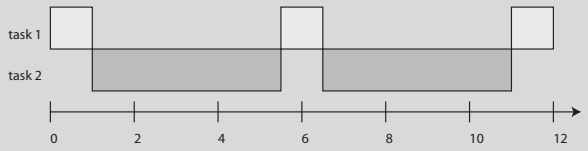
(d) For RM scheduling, are there any values for e_1 and e_2 that yield 100% utilization? If so, give an example.

Solution: Yes. For example, $e_1 = 4$ and $e_2 = 0$.

2. This problem studies dynamic-priority scheduling. Consider two tasks to be executed periodically on a single processor, where task 1 has period $p_1 = 4$ and task 2 has period $p_2 = 6$. Let the deadlines for each invocation of the tasks be the end of their period. That is, the first invocation of task 1 has deadline 4, the second invocation of task 1 has deadline 8, and so on. p.314

   (a) Let the execution time of task 1 be $e_1 = 1$. Find the maximum value for the execution time e_2 of task 2 such that EDF is feasible.

Solution: The largest execution time for task 2 is $e_2 = 4.5$. The following figure shows the resulting schedule:



The schedule repeats every 12 time units.

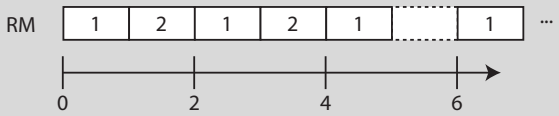
   (b) For the value of e_2 that you found in part (a), compare the EDF schedule against the RM schedule from Exercise 1 (a). Which schedule has less preemption? Which schedule has better utilization?

Solution: Comparing the schedule in (a) with the schedule in Exercise 1(a), we see that EDF has no preemption at all, while RM performs two preemptions every 12 time units. Moreover, EDF has 100% utilization, whereas RM has less.

3. This problem compares RM and EDF schedules. Consider two tasks with periods $p_1 = 2$ and $p_2 = 3$ and execution times $e_1 = e_2 = 1$. Assume that the deadline for each execution is the end of the period.

   (a) Give the RM schedule for this task set and find the processor utilization. How does this utilization compare to the Liu and Layland utilization bound of (12.2)?

Solution: The RM schedule is shown below:



The utilization is given by

$$U = 1 - 1/6 \approx 833\%$$

The utilization bound if $n = 2$ is $n(2^{1/n} - 1) \approx 0.828$. Thus, utilization is larger than the utilization bound, so we have no assurance that the RM schedule is feasible.

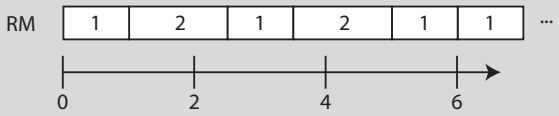
   (b) Show that any increase in e_1 or e_2 makes the RM schedule infeasible. If you hold $e_1 = e_2 = 1$ and $p_2 = 3$ constant, is it possible to reduce p_1 below 2 and still get a feasible schedule? By how much? If you hold $e_1 = e_2 = 1$ and $p_1 = 2$ constant, is it possible to reduce p_2 below 3 and still get a feasible schedule? By how much?

Solution: In the first three time units, the RM schedule must execute task 1 twice, because under the RM principle, it has highest priority and it has become enabled twice in this time period. With $e_1 = 1$, this leaves exactly one time unit to execute task 2 in its first period. Thus, any increase in e_2 will result in task 2 missing its deadline at time 3. Any increase in e_1 will leave less than one time unit for task 2 in its first period, resulting again in a missed deadline.

Holding e_1, e_2, and p_2 constant, we can reduce p_1 to 1.5 and still get a feasible schedule. Holding e_1, e_2, and p_1 constant, we can reduce p_2 to 2 and still get a feasible schedule. In both cases, no further reduction is possible because at this point we have 100% utilization.

   (c) Increase the execution time of task 2 to be $e_2 = 1.5$, and give an EDF schedule. Is it feasible? What is the processor utilization?

Solution: The EDF schedule is:

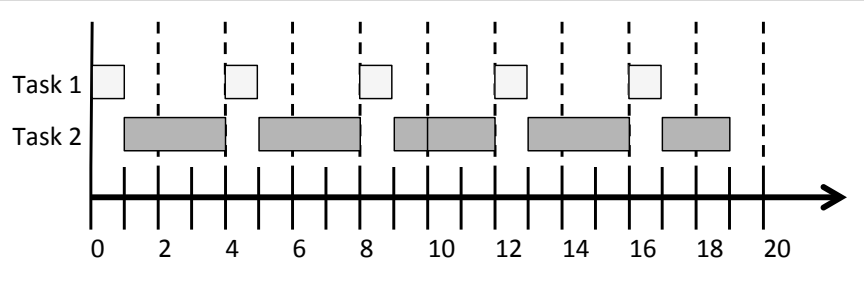


The schedule is feasible and the utilization is 100%.

4. This problem, formulated by Hokeun Kim, also compares RM and EDF schedules. Consider two tasks to be executed periodically on a single processor, where task 1 has period $p_1 = 4$ and task 2 has period $p_2 = 10$. Assume task 1 has execution time $e_1 = 1$, and task 2 has execution time $e_2 = 7$.

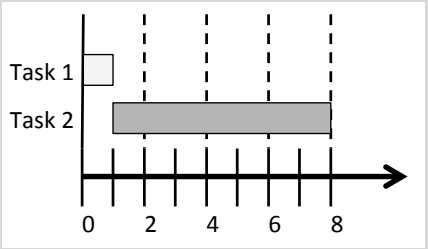
   (a) Sketch a rate-monotonic schedule (for 20 time units, the least common multiple of 4 and 10). Is the schedule feasible?

Solution: The rate monotonic schedule is feasible. The figure below shows the schedule.



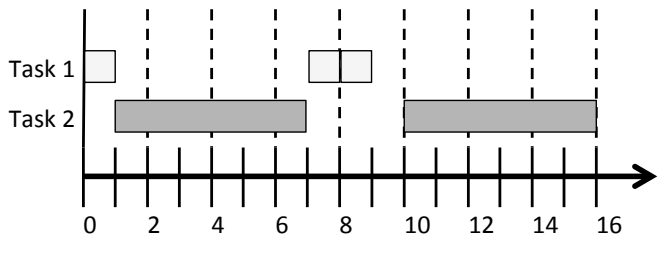
   (b) Now suppose task 1 and 2 contend for a mutex lock, assuming that the lock is acquired at the beginning of each execution and released at the end of each execution. Also, suppose that acquiring or releasing locks takes zero time and the priority inheritance protocol is used. Is the rate-monotonic schedule feasible?

Solution: No. Task 1 misses its deadline at time point 8 (the first deadline, which is met, is at time 4; the second deadline, which is not met, is at time 8).

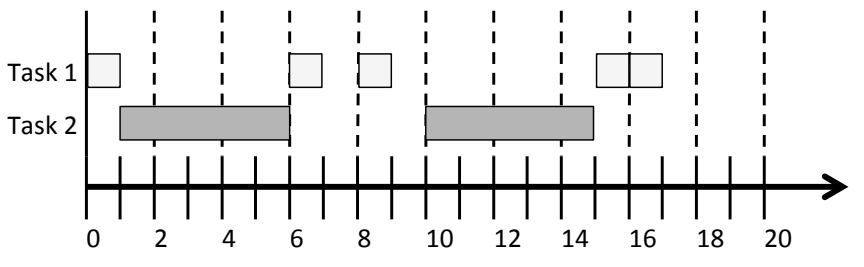


   (c) Assume still that tasks 1 and 2 contend for a mutex lock, as in part (b). Suppose that task 2 is running an **anytime algorithm**, which is an algorithm that can be terminated early and still deliver useful results. For example, it might be an image processing algorithm that will deliver a lower quality image when terminated early. Find the maximum value for the execution time e_2 of task 2 such that the rate-monotonic schedule is feasible. Construct the resulting schedule, with the reduced execution time for task 2, and sketch the schedule for 20 time units. You may assume that execution times are always positive integers.

Solution: If we reduce the execution time of task 2 to $e_2 = 6$, task 1 still misses its deadline at time point 16, as shown below:



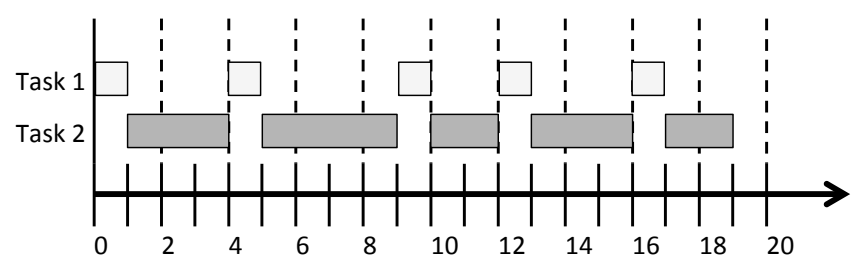
Reducing further to $e_2 = 5$ makes the schedule feasible, as shown below:



Therefore, the maximum value for the execution time of task 2 that makes the rate monotonic schedule feasible is $e_2 = 5$.

(d) For the original problem, where $e_1 = 1$ and $e_2 = 7$, and there is no mutex lock, sketch an EDF schedule for 20 time units. For tie-breaking among task executions with the same deadline, assume the execution of task 1 has higher priority than the execution of task 2. Is the schedule feasible?

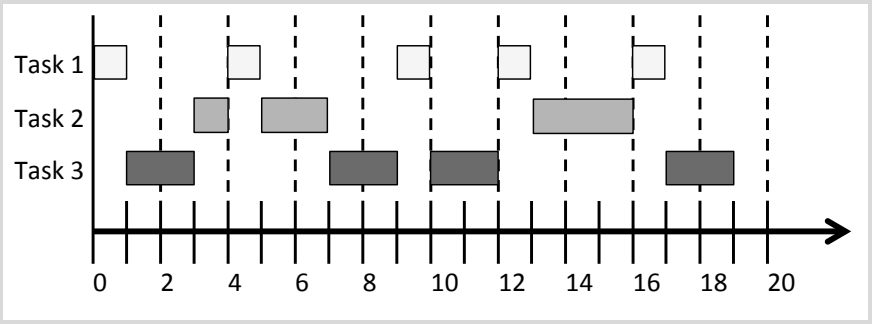
Solution: The EDF schedule is feasible. The figure below shows the schedule.



(e) Now consider adding a third task, task 3, which has period $p_3 = 5$ and execution time $e_3 = 2$. In addition, assume as in part (c) that we can adjust execution time of task 2.

Find the maximum value for the execution time e_2 of task 2 such that the EDF schedule is feasible and sketch the schedule for 20 time units. Again, you may assume that the execution times are always positive integers. For tie-breaking among task executions with the same deadline, assume task i has higher priority than task j if $i < j$.)

Solution: The total execution times of task 1 and 2 within the 20 time units window are 5 and 8, respectively. Thus, the total execution time of task 3 should be less than $20 - (5 + 8) = 7$ within the 20 time units window. Therefore, the EDF schedule becomes feasible when $e_2 = 3$. The figure below shows the schedule.


Task 1
Task 2
Task 3
0
2
4
6
8
10
12
14
16
18
20

5. This problem compares fixed vs. dynamic priorities, and is based on an example by **?**. Consider two periodic tasks, where task τ_1 has period $p_1 = 2$, and task τ_2 has period $p_2 = 3$. Assume that the execution times are $e_1 = 1$ and $e_2 = 1.5$. Suppose that the release time of execution i of task τ_1 is given by

$$r_{1,i} = 0.5 + 2(i-1)$$

for $i = 1, 2, \cdots$. Suppose that the deadline of execution i of task τ_1 is given by

$$d_{1,i} = 2i.$$

Correspondingly, assume that the release times and deadlines for task τ_2 are

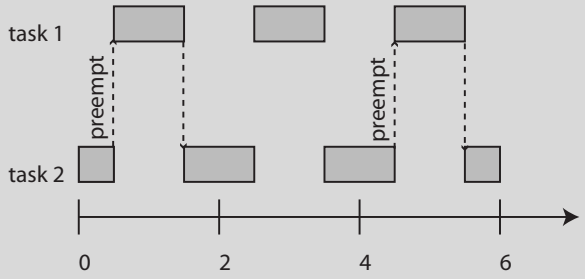
$$r_{2,i} = 3(i-1)$$

and

$$d_{2,i} = 3i.$$

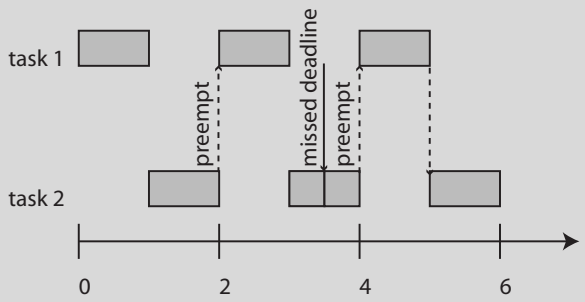
p.314 (a) Give a feasible fixed-priority schedule.

Solution: A feasible schedule gives higher priority to task τ_1 and is shown below:

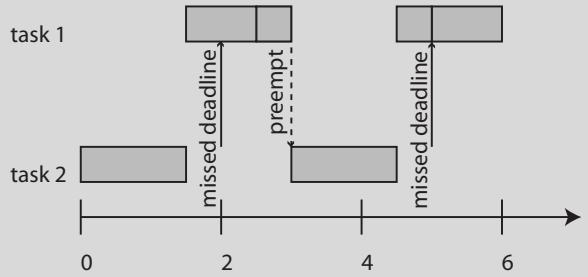


(b) Show that if the release times of all executions of task τ_1 are reduced by 0.5, then no fixed-priority schedule is feasible.

Solution: If we give higher priority to task τ_1, the resulting schedule, shown below, is infeasible:



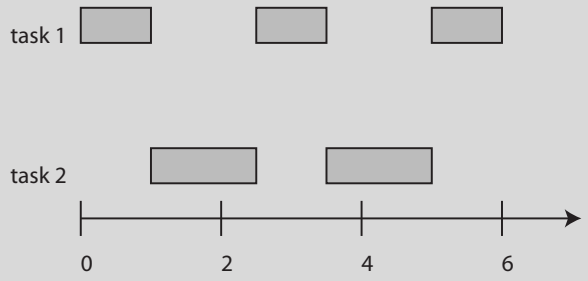
If we give higher priority to task τ_2, the resulting schedule, shown below, is also infeasible:



(c) Give a feasible dynamic-priority schedule with the release times of task τ_1 reduced to p.314

$$r_{1,i} = 2(i-1).$$

Solution: A feasible schedule is an EDF schedule, shown below, which alternates priorities:



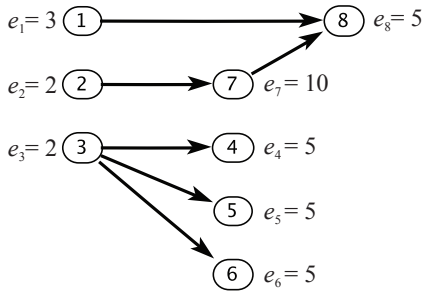
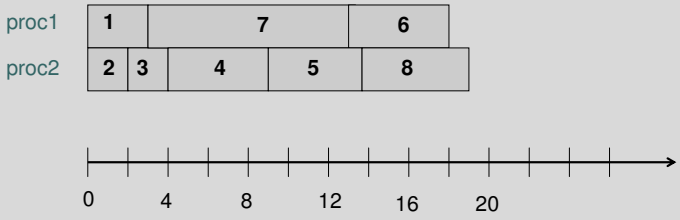


Figure 12.1: Precedence Graph for Exercise 6.

6. This problem studies scheduling anomalies. Consider the task precedence graph depicted in Figure 12.1 with eight tasks. In the figure, e_i denotes the execution time of task i. Assume task i has higher priority than task j if $i < j$. There is no preemption. The tasks must be scheduled respecting all precedence constraints and priorities. We assume that all tasks arrive at time $t = 0$.

   (a) Consider scheduling these tasks on two processors. Draw the schedule for these tasks and report the makespan.

Solution: The schedule is as shown below with a makespan of 19 units:

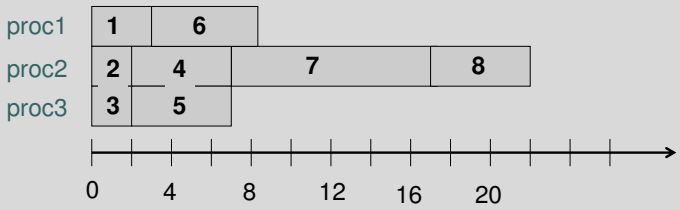


Schedule length = 19

   (b) Now consider scheduling these tasks on three processors. Draw the schedule for these tasks and report the makespan. Is the makespan bigger or smaller than that in part (a) above?

Solution: The schedule for 3 processors is as shown below with the makespan increasing to 22 units:

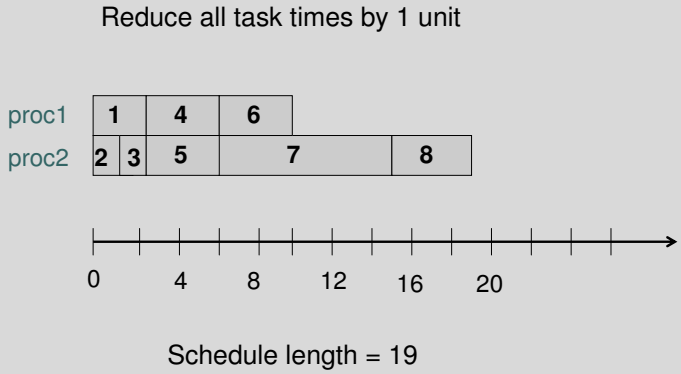
Add a processor



Schedule length = 22

(c) Now consider the case when the execution time of each task is reduced by 1 time unit. Consider scheduling these tasks on two processors. Draw the schedule for these tasks and report the makespan. Is the makespan bigger or smaller than that in part (a) above?

Solution: If each task time is reduced by 1 unit, the schedule is as shown with the makespan staying at 19 units below:



7. This problem studies the interaction between real-time scheduling and mutual exclusion, and was formulated by Kevin Weekly.

   Consider the following excerpt of code:

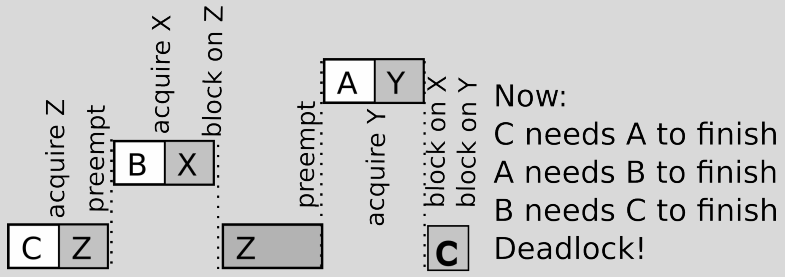
```
1  pthread_mutex_t X; // Resource X: Radio communication
2  pthread_mutex_t Y; // Resource Y: LCD Screen
3  pthread_mutex_t Z; // Resource Z: External Memory (slow)
4
5  void ISR_A() { // Safety sensor Interrupt Service Routine
6    pthread_mutex_lock(&Y);
7    pthread_mutex_lock(&X);
8    display_alert(); // Uses resource Y
9    send_radio_alert(); // Uses resource X
10   pthread_mutex_unlock(&X);
11   pthread_mutex_unlock(&Y);
12 }
13
14 void taskB() { // Status recorder task
15   while (1) {
16     static time_t starttime = time();
17     pthread_mutex_lock(&X);
18     pthread_mutex_lock(&Z);
19     stats_t stat = get_stats();
20     radio_report( stat ); // uses resource X
21     record_report( stat ); // uses resource Z
22     pthread_mutex_unlock(&Z);
23     pthread_mutex_unlock(&X);
24     sleep(100-(time()-starttime)); // schedule next excecution
25   }
26 }
27
28 void taskC() { // UI Updater task
29   while(1) {
30     pthread_mutex_lock(&Z);
31     pthread_mutex_lock(&Y);
32     read_log_and_display(); // uses resources Y and Z
33     pthread_mutex_unlock(&Y);
34     pthread_mutex_unlock(&Z);
35   }
36 }
```

   You may assume that the comments fully disclose the resource usage of the procedures. That is, if a comment says "uses resource X", then the relevant procedure uses only resource X. The scheduler running aboard the system is a priority-based preemptive scheduler, where taskB is higher priority than taskC. In this problem, ISR_A can be thought of as an asynchronous task with the highest priority.

   The intended behavior is for the system to send out a radio report every 100ms and for the UI to update constantly. Additionally, if there is a safety interrupt, a radio report is sent immediately and the UI alerts the user.

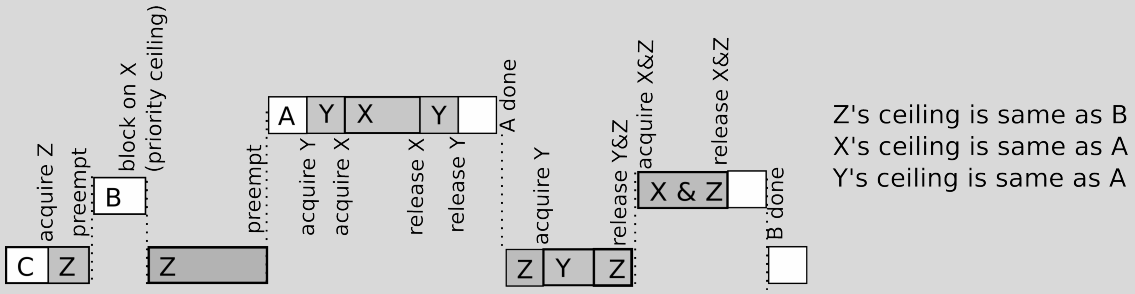
   (a) Occasionally, when there is a safety interrupt, the system completely stops working. In a scheduling diagram (like Figure 12.11 in the text), using the tasks {A,B,C}, and resources {X,Y,Z}, explain the cause of this behavior. Execution times do not have to be to scale in your diagram. Label your diagram clearly. You will be graded in part on the clarity of your answer, not just on its correctness.

Solution: The scheduling diagram is shown below:



(b) Using the priority ceiling protocol, show the scheduling diagram for the same sequence of events that you gave in part (a). Be sure to show all resource locks and unlocks until all tasks are finished or reached the end of an iteration. Does execution stop as before?

Solution: Execution does not stop, as illustrated with the following diagram:



(c) Without changing the scheduler, how could the code in taskB be reordered to fix the issue? Using an exhaustive search of all task/resource locking scenarios, prove that this system will not encounter deadlock. (Hint: There exists a proof enumerating 6 cases, based on reasoning that the 3 tasks each have 2 possible resources they could block on.)

Solution: The following could replace the body of the while loop in TaskB:

```
stats_t stat = get_stats();
pthread_mutex_lock(&X);
radio_report( stat ); // uses resource X
pthread_mutex_unlock(&X);
pthread_mutex_lock(&Z);
record_report( stat ); // uses resource Z
pthread_mutex_unlock(&Z);
```

1: If A blocks on X, it is dependent on B. While it has X, B will not block on any other resource and is guaranteed to release X.
2: If A blocks on Y, it is dependent on C. While holding Y, C already has both the locks it needs and will release both.
1+2: Thus, A will always finish.

3: If C blocks on Z, it depends on B. While it holds Z, B will not block any other resource and is guaranteed to release Z.
4: If C blocks on Y, it is dependent on A. We have shown that A always finishes.

3+4: Thus, C will always finish.

5: If B blocks on X, it is dependent on A finishing. We have shown that A always finishes.
6: If B blocks on Z, it depends on C. We have shown that C always finishes.
5+6: Thus, B always finishes.

1-6: Since, A, B, and C, always finish execution, this does not encounter deadlocks.
Note that the change of code allows us to guarantee that B releases it's locks. Otherwise, we would have a circular dependency and could not prove this.

Another solution is to swap the order that B requests its resources in (but still nested). The proof is similar to above, except line 3, in which you must show that B always finishes.

Part III

Analysis and Verification

13 Invariants and Temporal Logic — Exercises

1. For each of the following questions, give a short answer and justification.

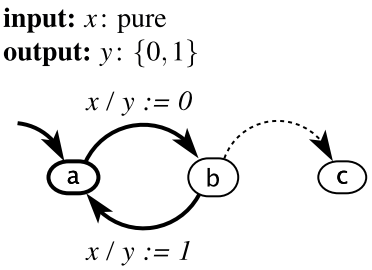
   (a) TRUE or FALSE: If $\mathbf{GF}p$ holds for a state machine A, then so does $\mathbf{FG}p$.

   Solution: FALSE. Consider a trace where p holds for every second reaction. This satisfies $\mathbf{GF}p$ but not $\mathbf{FG}p$.

   (b) TRUE or FALSE: $\mathbf{G}(\mathbf{G}p)$ holds for a trace if and only if $\mathbf{G}p$ holds.

   Solution: TRUE. "if" part: If $\mathbf{G}p$ holds, then p holds for every element of the trace, so $\mathbf{G}(\mathbf{G}p)$ holds; "only if" part: If $\mathbf{G}(\mathbf{G}p)$ holds, then for every suffix of the trace, $\mathbf{G}p$ holds, which means that p holds for every element of every suffix of the trace. Hence, it holds for every element of the trace.

2. Consider the following state machine:



(Recall that the dashed line represents a default transition.) For each of the following LTL formulas, determine whether it is true or false, and if it is false, give a counterexample:

(a) $x \implies \mathbf{F}\mathsf{b}$

Solution: *true*

(b) $\mathbf{G}(x \implies \mathbf{F}(y = 1))$

Solution: *false*. Counterexample: If the input sequence begins with $(x, absent, \cdots)$, then the machine will be in state c. From that point on, even if x is *present*, it is not true that eventually $y = 1$ will appear on the output.

(c) $(\mathbf{G}x) \implies \mathbf{F}(y = 1)$

Solution: *true*. In this case, the input x is always present, so $y = 1$ will be produced on every second reaction.

(d) $(\mathbf{G}x) \implies \mathbf{GF}(y = 1)$

Solution: *true*. In this case, the input x is always present, so $y = 1$ will be produced on every second reaction, which is infinitely often.

(e) $\mathbf{G}((\mathsf{b} \wedge \neg x) \implies \mathbf{FG}\mathsf{c})$

Solution: *true*

(f) $\mathbf{G}((\mathsf{b} \wedge \neg x) \implies \mathbf{G}\mathsf{c})$

Solution: *false*. Unlike the previous example, for this to be true, it requires that the state machine be in c in the *same* reaction in which it is in b, which cannot happen.

(g) $(\mathbf{GF}\neg x) \implies \mathbf{FG}\mathsf{c}$

Solution: *false*. A counterexample is the reaction to the input sequence $(x, x, \neg x, x, x, \neg x, \cdots)$, where the pattern repeats. In this case, x is absent infinitely often, so the left side is true. However, the right side not true because state c is never reached.

3. Consider the synchronous feedback composition studied in Exercise 6 of Chapter 6. Determine whether the following statement is true or false:

   The following temporal logic formula is satisfied by the sequence w for every possible behavior of the composition and is not satisfied by any sequence that is not a behavior of the composition:

   $$(\mathbf{G}w) \vee (w\mathbf{U}(\mathbf{G}\neg w)) \ .$$

   Justify your answer. If you decide it is false, then provide a temporal logic formula for which the assertion is true.

Solution: It is false. The sequence where w is absent everywhere satisfies the temporal logic formula, but it cannot be produced by the state machine composition. The first output of the state machine is required to be present. The following modification of the temporal logic formula works:

$$w \wedge ((\mathbf{G}w) \vee (w\mathbf{U}(\mathbf{G}\neg w)))$$

The first clause is required because otherwise there would be no assurance that the first output be present.

4. This problem is concerned with specifying in linear temporal logic tasks to be performed by a robot. Suppose the robot must visit a set of n locations $l_1, l_2, \ldots, l_n$. Let p_i be an atomic formula that is *true* if and only if the robot visits location l_i.

   Give LTL formulas specifying the following tasks:

   (a) The robot must eventually visit at least one of the n locations.

Solution:

$$\bigvee_{i=1}^{n} \mathbf{F}\, p_i$$

   (b) The robot must eventually visit all n locations, but in any order.

Solution:

$$\bigwedge_{i=1}^{n} \mathbf{F}\, p_i$$

   (c) The robot must eventually visit all n locations, in the order $l_1, l_2, \ldots, l_n$.

Solution:

$$\mathbf{F}\,(p_1 \wedge \mathbf{F}\,(p_2 \wedge \mathbf{F}\,(p_3 \wedge \ldots \mathbf{F}\, p_n)))$$

variables: *timerCount*: `uint`
input: *assert*: pure, *return*: pure
output: *return*: pure

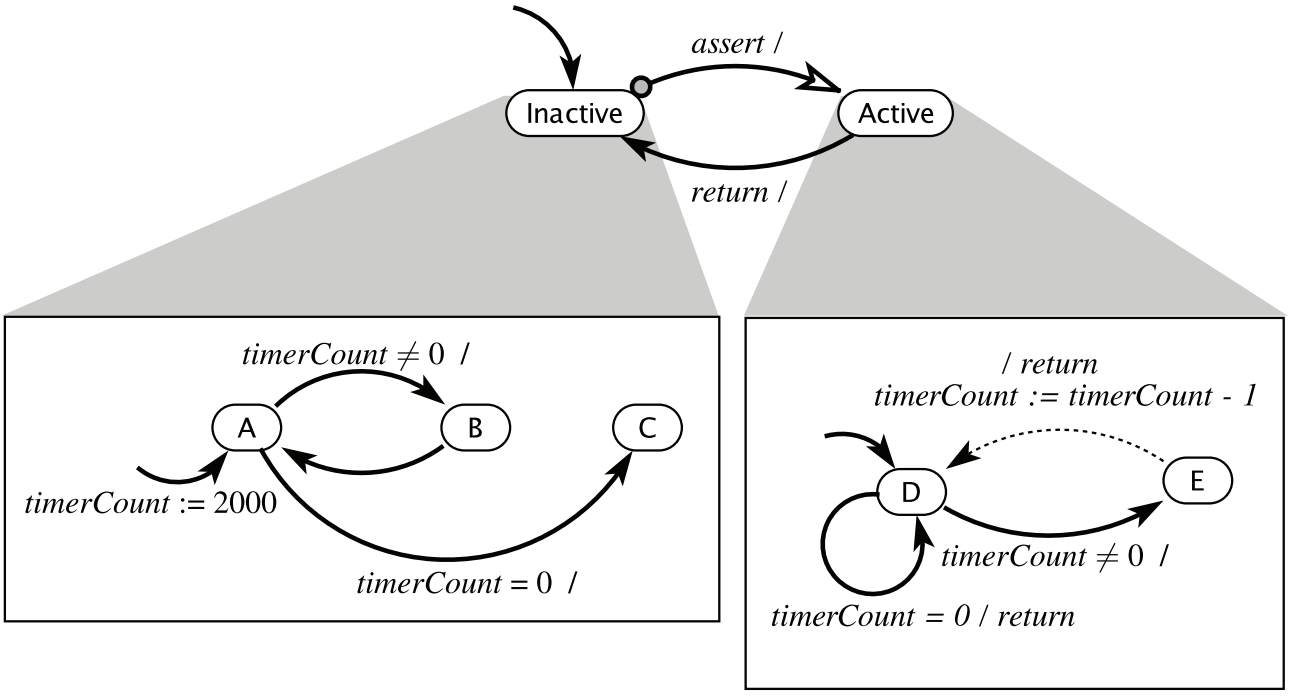


Figure 13.1: Hierarchical state machine modeling a program and its interrupt service routine.

5. Consider a system M modeled by the hierarchical state machine of Figure 13.1, which models an interrupt-driven program. M has two modes: Inactive, in which the main program executes, and Active, in which the interrupt service routine (ISR) executes. The main program and ISR read and update a common variable *timerCount*.

   Answer the following questions:

   (a) Specify the following property ϕ in linear temporal logic, choosing suitable atomic propositions:

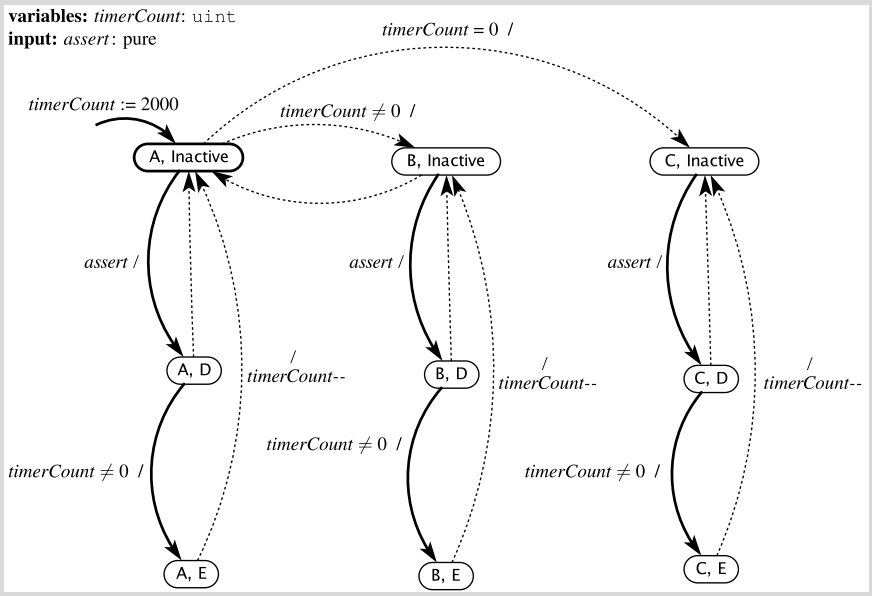
   ϕ: The main program eventually reaches program location C.

   Solution: Let C denote that atomic proposition that the machine is in state C. Then the following LTL formula expresses the property:

   $$\mathbf{F}\, C\,.$$

   (b) Does M satisfy the above LTL property? Justify your answer by constructing the product FSM. If M does not satisfy the property, under what conditions would it do so? Assume that the environment of M can assert the interrupt at any time.

Solution: No, M does not satisfy the LTL property, since, for example, it could stay in the Inactive mode indefinitely. Below is a flattened composite FSM obtained from hierarchical FSM of Figure 13.1:



If the interrupt is asserted exactly two thousand times, and then the conditional at program location A is checked, the LTL property will be satisfied, since *timerCount* will count down to 0 and the transition from A to C will be taken. Note that merely requiring the interrupt to be asserted at least two thousand times is not sufficient, since, if *timerCount* falls below 0, the transition from A to C will never be taken.

6. Express the postcondition of Example 13.3 as an LTL formula. State your assumptions clearly.

Solution: Let p_1 denote the condition that `altitude_control_task` is called. Let p_2 denote the condition that the `desired_climb` is in the range [-`CLIMB_MAX`, `CLIMB_MAX`]. Then the post condition can be written

$$\mathbf{G}(p_1 \implies \mathbf{X}p_2) ,$$

where we have assumed a state machine model where `altitude_control_task` executes in one reaction.

7. Consider the program fragment shown in Figure 11.6, which provides procedures for threads to communicate asynchronously by sending messages to one another. Please answer the following questions about this code. Assume the code is running on a single processor (not a multicore machine). You may also assume that only the code shown accesses the static variables that are shown.

   (a) Let s be an atomic proposition asserting that `send` releases the mutex (i.e. executes line 24). Let g be an atomic proposition asserting that `get` releases the mutex (i.e. executes line 38). Write an LTL formula asserting that g cannot occur before s in an execution of the program. Does this formula hold for the first execution of any program that uses these procedures?

Solution: The problem statement, as usual for statements in natural language, is ambiguous. One interpretation is that the first occurrence of g, if there is any occurrence of g, must occur after at least one occurrence of s. This can be expressed as

$$(\neg g \mathbf{U} s) \vee (\mathbf{G} \neg s).$$

The left part alone is not sufficient because it holds only if s eventually occurs. The second part allows for the possibility that s never occurs. An alternative expression is

$$\mathbf{G}(\mathbf{F} g \implies \neg g \mathbf{U} s).$$

A second possible interpretation is that for every occurrence of s, there is no occurrence of g before it. This can be expressed as

$$\mathbf{G}(g \implies \neg(\mathbf{F} s)),$$

or

$$\neg \mathbf{G}(g \implies \mathbf{F} s).$$

Under the first interpretation, the assertion holds for any first execution of the program. (We have to assume the first execution so that we can assume that the static initializers in the first part of the program do in fact define the initial values of the static variables.) The `get` procedure will not return until `size` is non-zero, and `size` will only be incremented if `send` has successfully acquired the mutex and executed line 22.

Under the second interpretations, the assertion does not hold for all executions of the program. In particular, the second occurrence of s could certainly be preceded by the first occurrence of g.

   (b) Suppose that a program that uses the `send` and `get` procedures in Figure 11.6 is aborted at an arbitrary point in its execution and then restarted at the beginning. In the new execution, it is possible for a call to `get` to return before any call to `send` has been made. Describe how this could come about. What value will `get` return?

Solution: The variables declared on lines 2 through 7 are static variables initialized with static initializers. The initial values given on those lines will be set when the program is loaded, not when the program begins executing. If the program is running in a context where the abort and restart does not reload the program, then the values of these variables will the ones they had when the first program execution was aborted. If the first execution is aborted after `send` had been called, but before `get` had been called, then the Pthreads condition variables will be in the

state indicating that a `send` has occurred. In this case, the first call to `get` will be able to acquire the mutex and will return the value written to the buffer by the last `send` in the first execution.

(c) Suppose again that a program that uses the `send` and `get` procedures above is aborted at an arbitrary point in its execution and then restarted at the beginning. In the new execution, is it possible for deadlock to occur, where neither a call to `get` nor a call to `send` can return? If so, describe how this could come about and suggest a fix. If not, give an argument.

Solution: There are many ways this could come about. If the program is aborted during a call to `send` but before line 24 executes, then the `mutex` variable will indicate that a thread holds the mutual exclusion lock. This assumes that the mutex variable is initialized only on line 6, using a static initializer. When the program resumes, it is possible that some thread in the new execution will acquire the same thread ID that appears to hold the lock, but it is also possible that there will be no thread with that ID. In that case, any thread that calls `send` or `get` will block attempting to acquire the lock.

The problem can be fixed by initializing the static variables in the `main` part of the program.

A more difficult solution would be to ensure an orderly abortion of the program that releases all muteness that are held by the program. This is more difficult because such abortions usually occur due to some fault, and an orderly abort may be difficult to guarantee. In that event, it might be possible to detect a disorderly abortion and reload the program; this will reset the mutex variable and all other static variables using their static initializers.

14 Equivalence and Refinement — Exercises

1. In Figure 14.1 are four pairs of actors. For each pair, determine whether
   - A and B are type equivalent,
   - A is a type refinement of B,
   - B is a type refinement of A, or
   - none of the above.

Solution:

(a) A is a type refinement of B.

(b) None of the above.

(c) A is a type refinement of B.

(d) A is a type refinement of B.

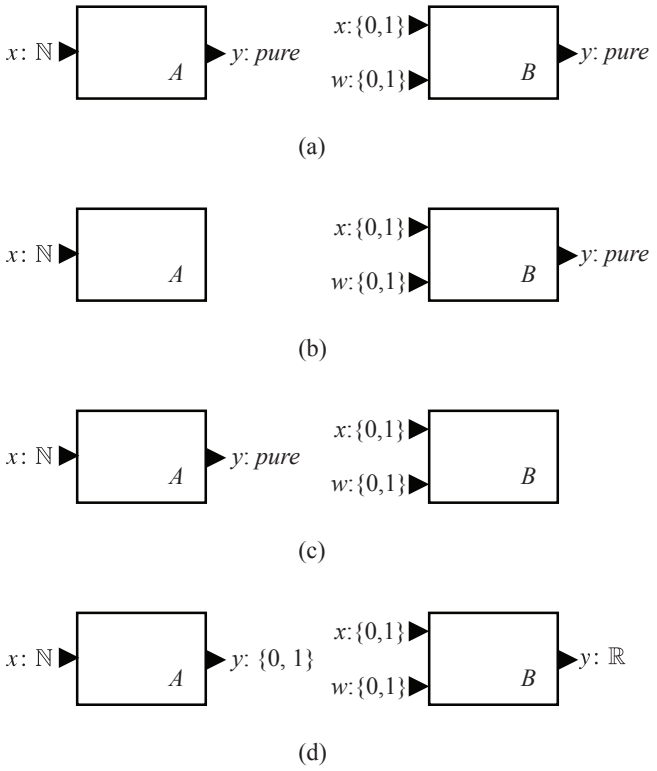


Figure 14.1: Four pairs of actors whose type refinement relationships are explored in Exercise 1.

2. In the box on page 367, a state machine M is given that accepts finite inputs x of the form (1), $(1,0,1)$, $(1,0,1,0,1)$, etc.

   (a) Write a regular expression that describes these inputs. You may ignore stuttering reactions.

   Solution: $1(01)^*$

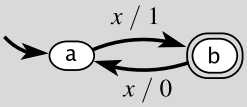
   (b) Describe the output sequences in $L_a(M)$ in words, and give a regular expression for those output sequences. You may again ignore stuttering reactions.

   Solution: Ignoring stuttering reactions, $L_a(M)$ includes as outputs all sequences that start with an odd number of *present* outputs, followed by an infinite tail of *absent* outputs. A regular expression for this is

   $$present(present\ present)^*$$

   (c) Create a state machine that accepts *output* sequences of the form (1), $(1,0,1)$, $(1,0,1,0,1)$, etc. (see box on page 367). Assume the input x is pure and that whenever the input is present, a present output is produced. Give a deterministic solution if there is one, or explain why there is no deterministic solution. What *input* sequences does your machine accept?

   Solution:
   input: x: pure
   output: y: $\{0,1\}$

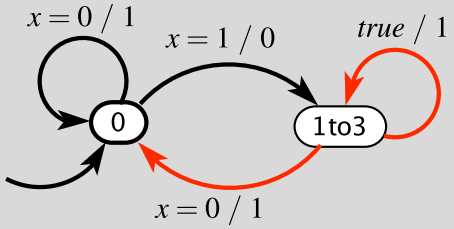
   

   This machine is deterministic and accepts input sequences consisting of a finite and odd number of *present* valuations.

3. The state machine in Figure 14.2 has the property that it outputs at least one 1 between any two 0's. Construct a two-state nondeterministic state machine that simulates this one and preserves that property. Give the simulation relation. Are the machines bisimilar?

Solution: The following machine does the job:

inputs: x: $\{0,1\}$
outputs: y: $\{0,1\}$



As suggested by the state names, 1to3 matches states 1, 2, and 3, while 0 matches 0. Hence, the simulation relation is

$$\{(0,0),(1,\text{1to3}),(2,\text{1to3}),(3,\text{1to3})\}\ .$$

The machines are not bisimilar. The above machine has more observable traces than the one in Figure 14.2.

inputs: x: $\{0,1\}$
outputs: y: $\{0,1\}$

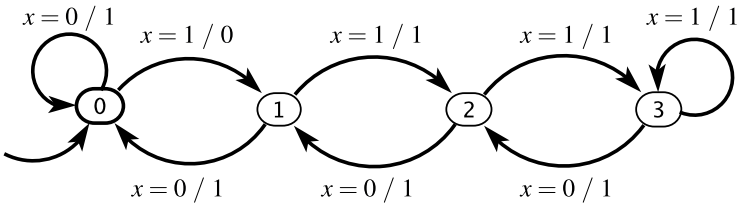


Figure 14.2: Machine that outputs at least one 1 between any two 0's.

4. Consider the FSM in Figure 14.3, which recognizes an input code. The state machine in Figure 14.4 also recognizes the same code, but has more states than the one in Figure 14.3. Show that it is equivalent by giving a bisimulation relation with the machine in Figure 14.3.

Solution: The bisimulation relation is

$$\{(start, start), (1,1), (11,11), (110,110), (start, 1100)\}.$$

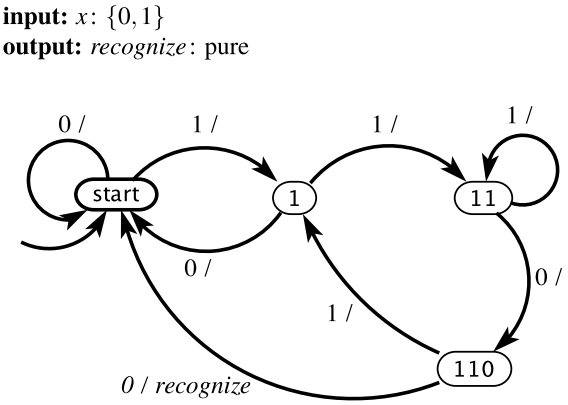


Figure 14.3: A machine that implements a code recognizer. It outputs *recognize* at the end of every input subsequence 1100; otherwise it outputs *absent*.

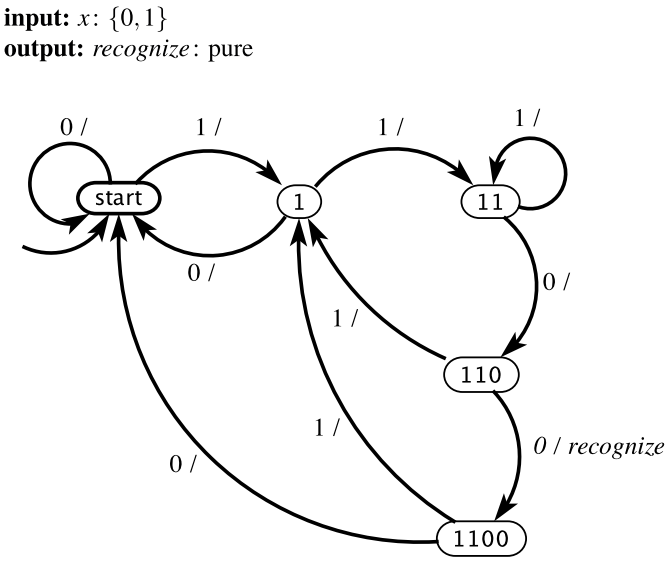
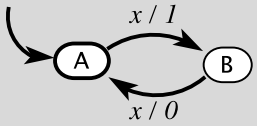


Figure 14.4: A machine that implements a recognizer for the same code as in Figure 14.3, but has more states.

5. Consider the state machine in Figure 14.5. Find a bisimilar state machine with only two states, and give the bisimulation relation.

Solution: A two-state bisimilar machine is shown below:

input: x: pure
output: y: $\{0,1\}$



The bisimulation relation is

$$S = \{(A,A),(B,B),(C,A),(D,B)\},$$

or equivalently,

$$S' = \{(A,A),(B,B),(A,C),(B,D)\},$$

input: x: pure
output: y: $\{0,1\}$

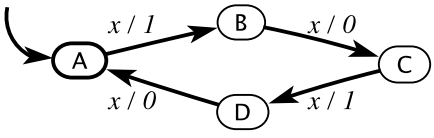


Figure 14.5: A machine that has more states than it needs.

6. You are told that state machine A has one input x, and one output y, both with type $\{1,2\}$, and that it has states $\{a,b,c,d\}$. You are told nothing further. Do you have enough information to construct a state machine B that simulates A? If so, give such a state machine, and the simulation relation.

Solution: Yes, we can give such a machine B. It has one state; let's call it e, with two self loops labeled:

$$true/1$$

$$true/2$$

The simulation relation is

$$S = \{(a,e),(b,e),(c,e),(d,e)\}.$$

7. Consider a state machine with a pure input x, and output y of type $\{0,1\}$. Assume the states are

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

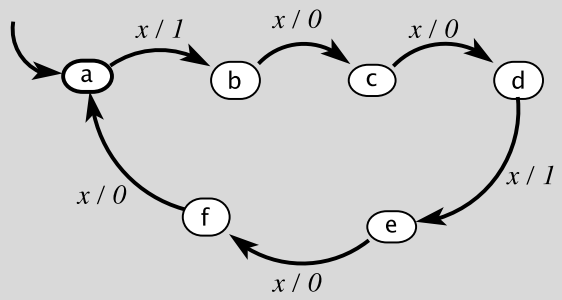
and the initial state is a. The *update* function is given by the following table (ignoring stuttering):

$(currentState, input)$	$(nextState, output)$
(a,x)	$(b,1)$
(b,x)	$(c,0)$
(c,x)	$(d,0)$
(d,x)	$(e,1)$
(e,x)	$(f,0)$
(f,x)	$(a,0)$

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

Solution: The state transition diagram:

input: x: pure
output: y: $\{0,1\}$



(b) Ignoring stuttering, give all possible behaviors for this machine.

Solution: Since the machine stutters exactly on reactions where the input is absent, we only need to consider inputs of the form

$$x = (p,p,p,\cdots),$$

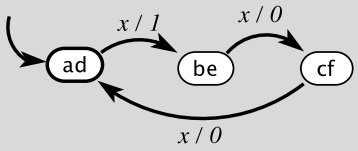
where p denotes *present*. Since the machine is deterministic, there is only one possible output sequence,

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

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

Solution: A bisimilar machine:

input: x: pure
output: y: $\{0,1\}$



The bisimulation relation is

$$\{(a,ad),(b,be),(c,cf),(d,ad),(e,be),(f,cf)\}.$$

8. For each of the following questions, give a short answer and justification.

   (a) TRUE or FALSE: Consider a state machine A that has one input x, and one output y, both with type $\{1,2\}$ and a single state s, with two self loops labeled $true/1$ and $true/2$. Then for any state machine B which has exactly the same inputs and outputs (along with types), A simulates B.

   Solution: FALSE. If the initial state of B has a transition $true/absent$, then for any simulation relation, since the initial states have to be related and A can never output absent, it cannot simulate this transition.

   (b) TRUE or FALSE: Suppose that f is an arbitrary LTL formula that holds for state machine A, and that A simulates another state machine B. Then we can safely assert that f holds for B.

   Solution: FALSE. An LTL formula may refer to states of a state machine in its atomic propositions, and there is no assurance that B has the same states. The LTL formula may need to be rewritten to replace each state s_A of A with a state s_B of B, where $(s_B, s_A) \in S$, where S is the simulation relation.

   (c) TRUE or FALSE: Suppose that A are B are two type-equivalent state machines, and that f is an LTL formula where the atomic propositions refer only to the inputs and outputs of A and B, not to their states. If the LTL formula f holds for state machine A, and A simulates state machine B, then f holds for B.

   Solution: TRUE. Because the atomic propositions refer only to the inputs and outputs, we need only examine the behaviors of the state machines, not their traces. By theorem 13.1, if A simulates B, then every behavior of B is also a behavior of A. Therefore, an LTL formula that holds for every behavior of A also holds for every behavior of B.

15

Reachability Analysis and Model Checking — Exercises

1. Consider the system M modeled by the hierarchical state machine of Figure 13.1, which models an interrupt-driven program.

   Model M in the modeling language of a verification tool (such as SPIN). You will have to construct an environment model that asserts the interrupt. Use the verification tool to check whether M satisfies ϕ, the property stated in Exercise 5:

   ϕ: The main program eventually reaches program location C.

   Explain the output you obtain from the verification tool.

Solution: This solution uses the SPIN model checker.

The following is the Promela model of the system M:

```
mtype = {returnType,assertType};

int timerCount = 2000;

chan returnChan = [1] of {mtype};
chan assertChan = [1] of {mtype};

byte mode = 1;
byte mainState = 1;
byte ISRState = 1;

/*
 env() is the environment model, which generates interrupts
*/
active proctype env()
{
  do
```

```
    :: if
       /* env, when scheduled, asserts an interrupt */
      :: assertChan!assertType
      fi
    od
}


/*
 sys() is the System Model.
 mode encodes the mode of the top-level FSM:
   Inactive or Active.
 mainState encodes the state of the main program:
   A=1, B=2, C=3.
 ISRState encodes the state of the ISR:
   D=1, E=2.
*/

active proctype sys()
{
  do
  :: if
     :: (mode == 1) ->  /* Inactive mode */
        if
        :: assertChan?assertType ->
        /* when interrupt asserted, move to Active mode */
           atomic {
             mode = 2;
             ISRState = 1;
           }
        ::
           if
           /* 1 is A, 2 is B, 3 is C */
           :: ((mainState == 1) && (timerCount != 0)) ->
              mainState = 2;
           :: ((mainState == 1) && (timerCount == 0)) ->
              mainState = 3;
           :: mainState == 2 ->
              mainState = 1;
           :: else -> skip /* mainState unchanged when = 3 (C) */
           fi
        fi

     :: (mode == 2) ->  /* Active mode */
       if
       :: returnChan?returnType -> mode = 1
       ::
         if
         :: ((ISRState == 1) && (timerCount != 0)) ->
   ISRState = 2;
         :: ISRState == 2 ->
            atomic {
             ISRState = 1;
             timerCount = timerCount-1;
             returnChan!returnType;
            }
         :: else -> skip
         fi
```

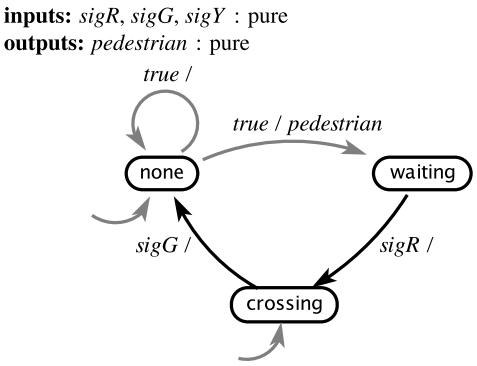
```
        fi
      :: else -> skip
      fi
  od
}
```

The LTL formula is simply `<> (mainState == 3)`.

On running SPIN we find that M does NOT satisfy ϕ. In our run of SPIN (your results might vary), it generates the counterexample where the main program stays at location A while timerCount is decremented more than 2000 times.

2. Figure 15.3 shows the synchronous-reactive composition of the traffic light controller of Figure 3.10 and the pedestrian model of Figure 3.11.

   Consider replacing the pedestrian model in Figure 15.3 with the alternative model given below where the initial state is nondeterministically chosen to be one of none or crossing:



   (a) Model the composite system in the modeling language of a verification tool (such as SPIN). How many reachable states does the combined system have? How many of these are initial states?

Solution: FIXME: create model.

   (b) Formulate an LTL property stating that every time a pedestrian arrives, eventually the pedestrian is allowed to cross (i.e., the traffic light enters state red).

Solution: The LTL property is

$$\mathbf{G}\left[(\mathsf{none} \wedge \mathbf{X}\mathsf{waiting}) \Rightarrow \mathbf{F}\,\mathsf{red}\right]$$

Note that the inputs and outputs of the state machines that are combined by synchronous composition (such as *pedestrian*) cannot be referenced in this property as they do not appear in the composite model.

   (c) Use the verification tool to check whether the model constructed in part (a) satisfies the LTL property specified in part (b). Explain the output of the tool.

Solution: FIXME

3. The notion of reachability has a nice symmetry. Instead of describing all states that are reachable from some initial state, it is just as easy to describe all states from which some state can be reached. Given a finite-state system M, the **backward reachable states** of a set F of states is the set B of all states from which some state in F can be reached. The following algorithm computes the set of backward reachable states for a given set of states F:

Input : A set F of states and transition relation δ for closed finite-state system M
Output: Set B of backward reachable states from F in M

1 **Initialize:** $B := F$
2 $B_{\text{new}} := B$
3 **while** $B_{\text{new}} \neq \emptyset$ **do**
4 $\quad B_{\text{new}} := \{s \mid \exists s' \in B \text{ s.t. } s' \in \delta(s) \wedge s \notin B\}$
5 $\quad B := B \cup B_{\text{new}}$
6 **end**

Explain how this algorithm can check the property $\mathbf{G}p$ on M, where p is some property that is easily checked for each state s in M. You may assume that M has exactly one initial state s_0.

Solution: Let the input F to the algorithm be the set of all states where p does not hold. Then $\mathbf{G}p$ is true if and only if the output B of the algorithm does not contain the initial state s_0.

16 Quantitative Analysis — Exercises

1. This problem studies execution time analysis. Consider the C program listed below:

```
1  int arr[100];
2
3  int foo(int flag) {
4    int i;
5    int sum = 0;
6
7    if (flag) {
8      for(i=0;i<100;i++)
9        arr[i] = i;
10   }
11
12   for(i=0;i<100;i++)
13     sum += arr[i];
14
15   return sum;
16 }
```

Assume that this program is run on a processor with data cache of size big enough that the entire array `arr` can fit in the cache.

(a) How many paths does the function `foo` of this program have? Describe what they are.

Solution: This program has two paths, one with `flag` $== 0$ and the other with `flag` $!= 0$. For the latter path, the array `arr` is initialized before computing the sum, while for the former it is not.

(b) Let T denote the execution time of the second `for` loop in the program. How does executing the first `for` loop affect the value of T? Justify your answer.

Solution: If the first `for` loop is executed before the second `for` loop (i.e., `flag` $!= 0$), then the array `arr` will be in the cache when the second `for` loop executes, so each access to `arr` inside

the second `for` loop will be a cache hit, and so T will be smaller if the first `for` loop executes before the second one.

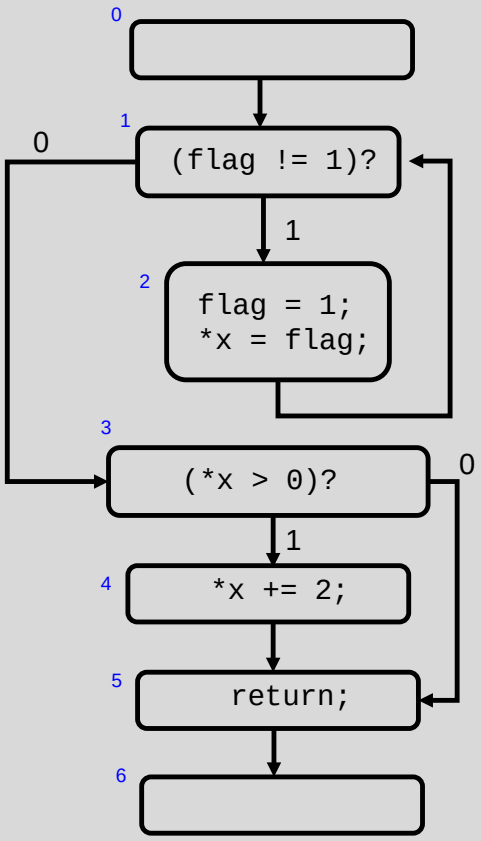
2. Consider the program given below:

```
1 void testFn(int *x, int flag) {
2   while (flag != 1) {
3     flag = 1;
4     *x = flag;
5   }
6   if (*x > 0)
7     *x += 2;
8 }
```

In answering the questions below, assume that `x` is not `NULL`.

(a) Draw the control-flow graph of this program. Identify the basic blocks with unique IDs starting with 1.

Solution:



Note that we have added a dummy source node, numbered 0, to represent the entry to the function. For convenience, we have also introduced a dummy sink node, although this is not strictly required.

(b) Is there a bound on the number of iterations of the while loop? Justify your answer.

Solution: The bound on the number of iterations is one. If `flag` is not equal to 1 initially, the loop gets executed and `flag` gets set to 1, so the loop must exit the next time the condition is evaluated.

(c) How many total paths does this program have? How many of them are feasible, and why?

Solution: This program has a total of 4 paths, corresponding to 2 choices of the conditional in the while loop and 2 choices for the conditional in the if-statement. 3 of these paths are feasible — the only infeasible path is the path 0-1-2-1-3-5-6 corresponding to executing one iteration of the while loop (in which `*x` is set to 1) and the else branch of the conditional `*x > 0`. This cannot be executed since `*x` is greater than 0 after executing one iteration of the loop.

(d) Write down the system of flow constraints, including any logical flow constraints, for the control-flow graph of this program.

Solution: The system of flow constraints is as follows: (the logical flow constraint is given later)

$$\begin{aligned} x_0 &= 1 \\ x_1 &= 2 \\ x_1 &= d_{12} + d_{13} \\ x_2 &= 1 \\ x_2 &= d_{12} = d_{21} \\ x_3 &= d_{13} = d_{34} + d_{35} \\ x_4 &= d_{34} = d_{45} \\ x_5 &= d_{35} + d_{45} \\ x_6 &= 1 \end{aligned}$$

Even though this program has a loop, the loop body is executed at most once. Moreover, the two edges leaving Node 3 are mutually exclusive. Thus, we have the following two logical flow constraints:

$$\begin{aligned} d_{12} + d_{35} &\leq 1 \\ d_{34} + d_{35} &= 1 \end{aligned}$$

(e) Consider running this program uninterrupted on a platform with a data cache. Assume that the data pointed to by `x` is not present in the cache at the start of this function.

For each read/write access to `*x`, argue whether it will be a cache hit or miss.

Now, assume that `*x` is present in the cache at the start of this function. Identify the basic blocks whose execution time will be impacted by this modified assumption.

Solution: The answer to this question depends on the path executed.

If the path is 0-1-3-4-5-6, then the access in Basic Block 3 is a miss, but the accesses in Block 4 will be a hit. The miss in Block 3 will occur even in path 0-1-3-5-6.

If the path is 0-1-2-1-3-4-5-6, then the access in Block 2 will be a miss, but subsequent acesses will hit.

For the second part, if `*x` is present in the cache, then every access is likely to be a hit. The accesses in Blocks 2 and 3 are affected by this change.

3. Consider the function `check_password` given below that takes two arguments: a user ID `uid` and candidate password `pwd` (both modeled as `int`s for simplicity). This function checks that password against a list of user IDs and passwords stored in an array, returning 1 if the password matches and 0 otherwise.

```
1  struct entry {
2    int user;
3    int pass;
4  };
5  typedef struct entry entry_t;
6
7  entry_t all_pwds[1000];
8
9  int check_password(int uid, int pwd) {
10   int i = 0;
11   int retval = 0;
12
13   while(i < 1000) {
14     if (all_pwds[i].user == uid && all_pwds[i].pass == pwd) {
15       retval = 1;
16       break;
17     }
18     i++;
19   }
20
21   return retval;
22 }
```

(a) Draw the control-flow graph of the function `check_password`. State the number of nodes (basic blocks) in the CFG. (Remember that each conditional statement is considered a single basic block by itself.)

Also state the number of paths from entry point to exit point (ignore path feasibility).

Solution: FIXME: draw the CFG

6 nodes.

2^{1000} is an upper bound on the number of paths. Actual number is 1001.

(b) Suppose the array `all_pwds` is sorted based on passwords (either increasing or decreasing order). In this question, we explore if an external client that calls `check_password` can *infer anything about the passwords* stored in `all_pwds` by repeatedly calling it and *recording the execution time* of `check_password`. Figuring out secret data from "physical" information, such as running time, is known as a *side-channel attack*.

In each of the following two cases, what, if anything, can the client infer about the passwords in `all_pwds`?

(i) The client has exactly one (uid, password) pair present in `all_pwds`

(ii) The client has NO (uid, password) pairs present in in `all_pwds`

Assume that the client knows the program but not the contents of the array `all_pwds`.

Solution: For (i), the client can infer the fraction of users that have passwords alphabetically smaller than his/hers. First call `check_password` with the correct (uid, password) pair, and then call it again with an incorrect pair (e.g., same uid, different password).

For (ii), the client can infer nothing (as the program runs through the entire array and takes the maximum amount of time).

4. Consider the code below that implements the logic of a highly simplified vehicle automatic transmission system. The code aims to set the value of `current_gear` based on a sensor input `rpm`. `LO_VAL` and `HI_VAL` are constants whose exact values are irrelevant to this problem (you can assume that `LO_VAL` is strictly smaller than `HI_VAL`).

```
1  volatile float rpm;
2
3  int current_gear; // values range from 1 to 6
4
5  void change_gear() {
6    if (rpm < LO_VAL)
7      set_gear(-1);
8    else {
9      if (rpm > HI_VAL)
10       set_gear(1);
11   }
12
13   return;
14 }
15
16 void set_gear(int update) {
17   int new_gear = current_gear + update;
18   if (new_gear > 6)
19     new_gear = 6;
20   if (new_gear < 1)
21     new_gear = 1;
22
23   current_gear = new_gear;
24
25   return;
26 }
```

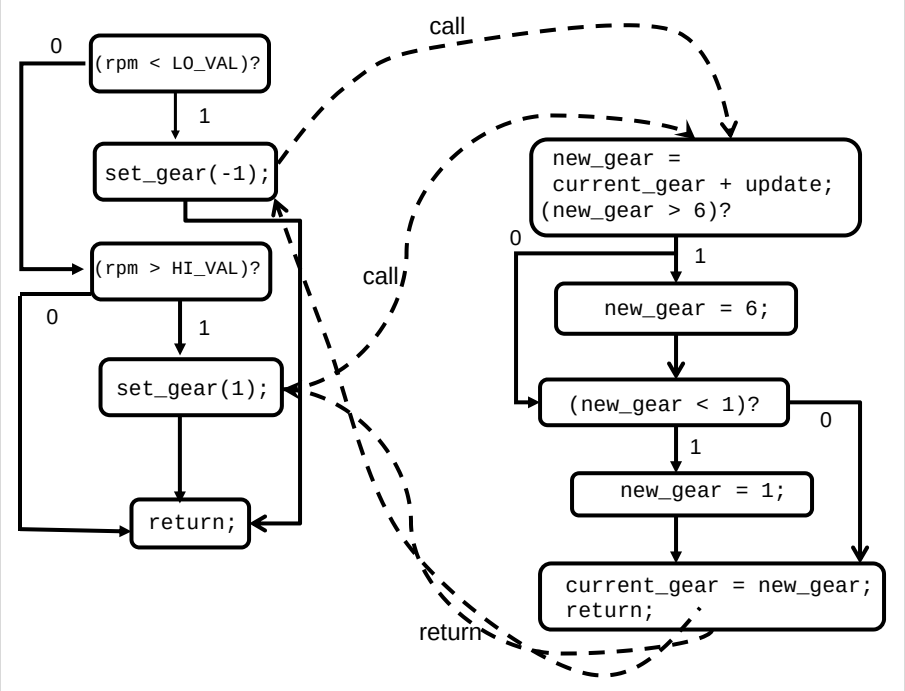
This is a 6-speed automatic transmission system, and thus the value of `current_gear` ranges between 1 and 6.

Answer the following questions based on the above code:

(a) Draw the control-flow graph (CFG) of the program starting from `change_gear`, *without inlining* function `set_gear`. In other words, you should draw the CFG using call and return edges.

For brevity, you need not write the code for the basic blocks inside the nodes of the CFG. Just indicate which statements go in which node by using the line numbers in the code listing above.

Solution: Our solution includes the code in each node.



(b) Count the number of execution paths from the entry point in `set_gear` to its exit point (the return statement). Ignore feasibility issues for this question. Also count the number of paths from the entry point in `change_gear` to its exit point (the return statement), including the paths through `set_gear`. State the number of paths in each case.

Solution: For `set_gear`: 4 paths. For `change_gear`: 9 paths (4+4+1).

(c) Now consider path feasibility. Recalling that `current_gear` ranges between 1 and 6, how many feasible paths does `change_gear` have? Justify your answer.

Solution: There are 5 feasible paths. Recall that the value of `current_gear` is always between 1 and 6.

When `rpm` is less than `LO_VAL`, then since the gear can only be decremented, `new_gear` cannot exceed 6. So there are only two cases: either it is less than 1, or it is not. Similarly, when `rpm` is greater than `HI_VAL`, the gear can only be incremented, and so `new_gear` cannot fall below 1. So there are only two cases: either it is greater than 6, or it is not.

Thus, in each case (calling `set_gear` with +1 or -1), there are only 2 feasible paths through `set_gear`. Thus, the total number of paths is 2+2+1=5.

Common error: analyzing `set_gear` independent of the context in which it was called, and thus coming up with 3 feasible paths through it, so 3+3+1=7 through `change_gear`.

(d) Give an example of a feasible path and of an infeasible path through the function `change_gear`. Describe each path as a sequence of line numbers, ignoring the line numbers corresponding to function definitions and return statements.

Solution: There are many possible answers. Here is one.
Feasible: 6; 7; 17; 18; 20; 21; 23.
Infeasible: 6; 7; 17; 18; 19; 20; 21; 23

17

Security and Privacy — Exercises

1. Consider the buffer overflow vulnerability in Example 17.2. Modify the code so as to prevent the buffer overflow.

Solution: There are several ways to modify the code depending on the specific setting in which it is used. One approach is to take only the first 16 bytes from the sensor stream, and this is implemented in the code below.

```
1  char sensor_data[16];
2  int secret_key;
3
4  void read_sensor_data() {
5    int i = 0;
6
7    for(i=0; i<16; i++) {
8      if(more_data())
9        sensor_data[i] = get_next_byte();
10     else
11       break; // exit if no more data available
12   }
13
14   return;
15 }
```

This code will write at most 16 bytes of data to `sensor_data`, thus eliminating the buffer overflow.

2. Suppose a system M has a secret input s, a public input x, and a public output z. Let all three variables be Boolean. Answer the following TRUE/FALSE questions with justification:

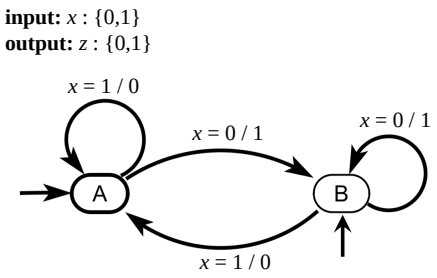
(a) Suppose M satisfies the linear temporal logic (LTL) property $\mathbf{G}\neg z$. Then M must also satisfy observational determinism.

Solution: TRUE. The public output of M is always 0 no matter what the values the secret input takes.

(b) Suppose M satisfies the linear temporal logic (LTL) property $\mathbf{G}[(s \wedge x) \Rightarrow z]$. Then M must also satisfy observational determinism.

Solution: FALSE. Consider an M where z is 1 whenever s and x are 1, but is 0 otherwise. Then, the input sequence where s takes values only 1 can be distinguished from the input sequence where s takes values only 0.

3. Consider the finite-state machine below with one input x and one output z, both taking values in $\{0,1\}$. Both x and z are considered public ("low") signals from a security viewpoint. However, the state of the FSM (i.e., "A" or "B") is considered secret ("high").



TRUE or FALSE: There is an input sequence an attacker can supply that tells him whether the state machine begins execution in A or in B.

Solution: FALSE. On input 0, the FSM always generates 1, and vice-versa, whether the FSM begins execution in A or B. In other words, it satisfies observational determinism — the output is a deterministic function of the input and nothing else.

Part IV

Appendices

This part of this text covers some background in mathematics and computer science that is useful to more deeply understand the formal and algorithmic aspects of the main text. Appendix A reviews basic notations in logic, with particular emphasis on sets and functions. Appendix B reviews notions of complexity and computability, which can help a system designer understand the cost of implementing a system and fundamental limits that make certain systems not implementable.

A

Sets and Functions — Exercises

1. This problem explores properties of onto and one-to-one functions.

   (a) Show that if $f: A \to B$ is onto and $g: B \to C$ is onto, then $(g \circ f): A \to C$ is onto.

   Solution: The image of $(g \circ f)$ is $\hat{g}(\hat{f}(A))$. Since f is onto, $\hat{f}(A) = B$. Since g is onto, $\hat{g}(\hat{f}(A)) = C$. Hence, $(g \circ f)$ is onto.

   (b) Show that if $f: A \to B$ is one-to-one and $g: B \to C$ is one-to-one, then $(g \circ f): A \to C$ is one-to-one.

   Solution: Since f is one-to-one, $a \neq a' \Rightarrow f(a) \neq f(a')$. Since g is one-to-one, $f(a) \neq f(a') \Rightarrow g(f(a)) \neq g(f(a'))$. Hence, $a \neq a' \Rightarrow (g \circ f)(a) \neq (g \circ f)(a')$.

2. Let $\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \cdots\}$ be the von Neumann numbers as defined in the box on page 477. This problem explores the relationship between this set and $\mathbb{N}$, the set of natural numbers.

   (a) Let $f: \omega \to \mathbb{N}$ be defined by
   $$f(x) = |x|, \quad \forall\, x \in \omega\,.$$
   That is, $f(x)$ is the size of the set x. Show that f is bijective.

Solution: f is onto because the set ω contains at least one set with any size $n \in \mathbb{N}$. f is one-to-one because ω contains no more than one set with any size $n \in \mathbb{N}$. Since f is onto and one-to-one, it is bijective.

   (b) The lifted version of the function f in part (a) is written $\hat{f}$. What is the value of $\hat{f}(\{0, \{0\}\})$? What is the value of $f(\{0, \{0\}\})$? Note that on page 473 it is noted that when there is no ambiguity, $\hat{f}$ may be written simply f. For this function, is there such ambiguity?

Solution: $\hat{f}(\{0, \{0\}\}) = \{0, 1\}$. $f(\{0, \{0\}\}) = 2$. Hence, in this case, there is definitely ambiguity, and the shorthand notation should not be used.

Complexity and Computability — Exercises

1. Complete the formal definition of the tape machine T by giving the initial state of T and the mathematical description of its transition function $update_T$.

   Solution: The initial state of the tape machine is $(0, d_0)$ where the 0 indicates that the head is initially in the left-most position on the tape, and d_0 is the function from $\mathbb{N}$ to $\Sigma \cup \{\square\}$ such that $d_0(n) = \square$ for all $n \in \mathbb{N}$; i.e., in the initial state, all tape cells are empty.

   Suppose the current state $s = (l, d)$, where l is the current location of the head and d is the function representing the current contents of the tape. Similarly, let the input at any step be denoted by $i = (w, m)$ where w denotes the symbol to be written and m the next move of the head.

   Then, $update_T(s, i) = (s', o)$, where $s' = (l', d')$ is the next state and o is the output, with values as follows:

   - $d'(l) = w$ and $d'(n) = d(n)$ for all $n \neq l$.
   - l is updated as follows:
     - if $m = L$, $l' = l - 1$ if $l > 0$ and $l' = l$ otherwise (cannot move off the left end of the tape).
     - if $m = R$, $l' = l + 1$.
   - $o = d(l)$.

2. *Directed, acyclic graphs* (DAGs) have several uses in modeling, design, and analysis of embedded systems; e.g., they are used to represent precedence graphs of tasks (see Chapter 12) and control-flow graphs of loop-free programs (see Chapter 16).

   A common operation on DAGs is to topologically sort the nodes of the graph. Formally, consider a DAG $G = (V, E)$ where V is the set of vertices $\{v_1, v_2, \ldots, v_n\}$ and E is the set of edges. A **topological sort** of G is a linear ordering of vertices $\{v_1, v_2, \ldots, v_n\}$ such that if $(v_i, v_j) \in E$ (i.e., there is a directed edge from v_i to v_j), then vertex v_i appears before vertex v_j in this ordering. p.165

   The following algorithm due to **?** topologically sorts the vertices of a DAG:

input : A DAG $G = (V, E)$ with n vertices and m edges.
output: A list L of vertices in V in topologically-sorted order.

```
1  L ← empty list
2  S ← {v | v is a vertex with no incoming edges}
3  while S is non-empty do
4  |   Remove vertex v from S
5  |   Insert v at end of list L
6  |   for each vertex u such that edge (v,u) is in E do
7  |   |   Mark edge (v,u)
8  |   |   if all incoming edges to u are marked then
9  |   |   |   Add u to set S
10 |   |   end
11 |   end
12 end
   L contains all vertices of G in topologically sorted order.
13
```

Algorithm B.1: Topological sorting of vertices in a DAG

State the asymptotic time complexity of Algorithm B.1 using Big O notation. Prove the correctness of your answer.

Solution: $O(m+n)$

The algorithm has two loops: the while-loop starting on Line 3 and the for-loop starting on Line 6. Note that the while-loop can be executed exactly once for each node in the graph, for at most n times. Given a fixed node in the graph, v, the for-loop is executed once for every outgoing edge from v to some other node. Thus, summed over all iterations of the while-loop, the number of iterations of the for-loop is at most the number of edges in the graph m.

From this, we conclude that Lines 3,4, and 5 are executed a total of at most n times and Lines 6-11 are executed at most m times in total. Since each execution of each of those lines takes time $O(1)$, the overall runtime of the algorithm is $O(n+m)$.