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?

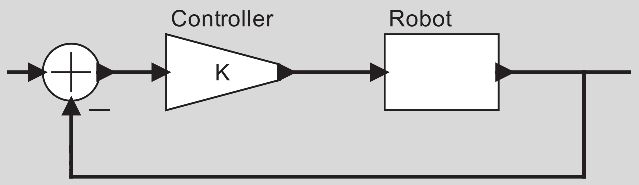
Solution: The model is not BIBO stable. For example, the input

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

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

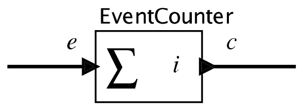
$$\theta(t) = au(t)(1 - e^{-Kt}).$$

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

3

Discrete Dynamics — Exercises

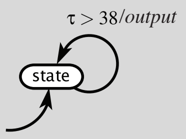
1. Consider an event counter that is a simplified version of the counter in Section 3.1. It has an icon like this:



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.

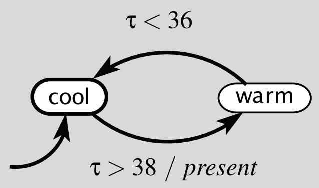
(a) For the first version, your state machine should simply produce a *present* output whenever the input is *present* and greater than 38 degrees. Otherwise, the output should be absent.

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



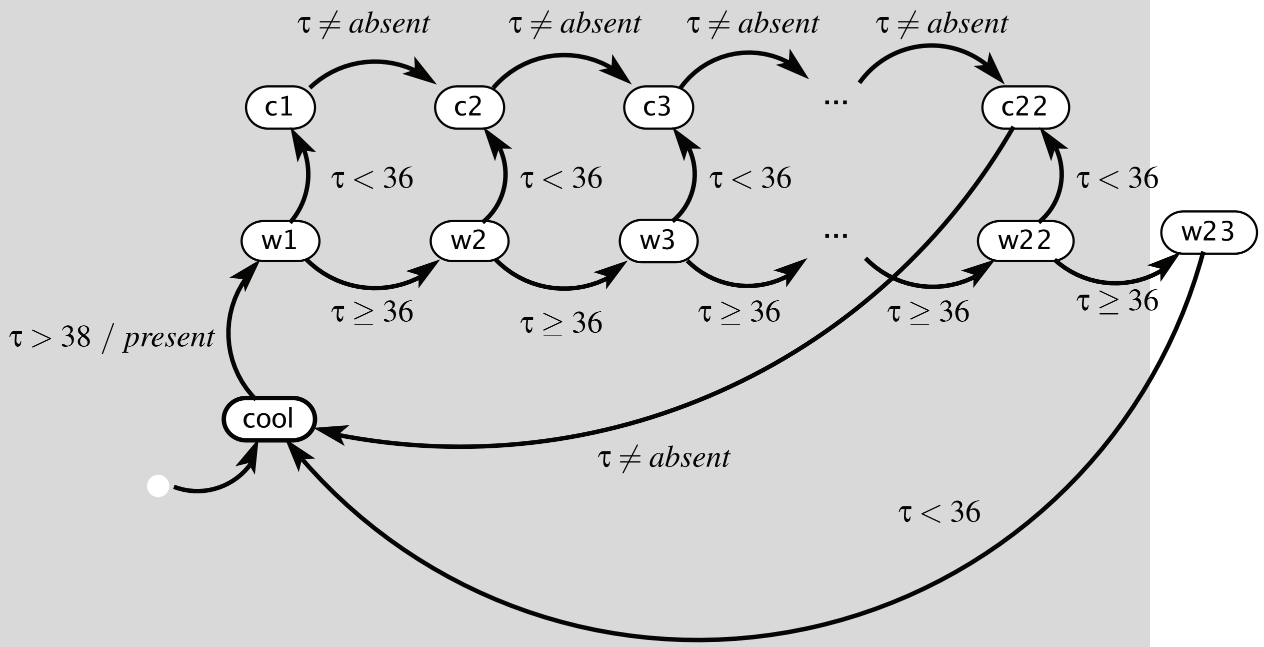
(b) For the second version, your state machine should have hysteresis. Specifically, it should produce a *present* output the first time the input is greater than 38 degrees, and subsequently, it should produce a *present* output anytime the input is greater than 38 degrees but has dropped below 36 degrees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

Solution:



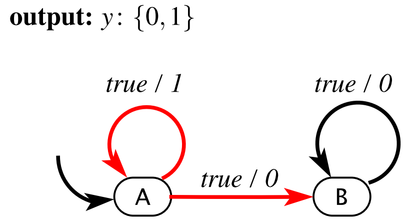
(c) For the third version, your state machine should implement the same hysteresis as in part (b), but also produce a *present* output at most once per day.

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.

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.

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) $x = (p, p, p, p, p, \cdots), \quad y = (0, 1, 1, 0, 0, \cdots)$
   (b) $x = (p, p, p, p, p, \cdots), \quad y = (0, 1, 1, 0, a, \cdots)$
   (c) $x = (a, p, a, p, a, \cdots), \quad y = (a, 1, a, 0, a, \cdots)$
   (d) $x = (p, p, p, p, p, \cdots), \quad y = (0, 0, a, a, a, \cdots)$
   (e) $x = (p, p, p, p, p, \cdots), \quad y = (0, a, 0, a, a, \cdots)$

Solution:

(a) no
(b) yes
(c) no
(d) yes
(e) no

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

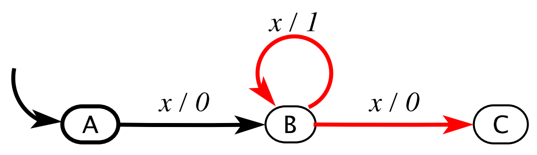


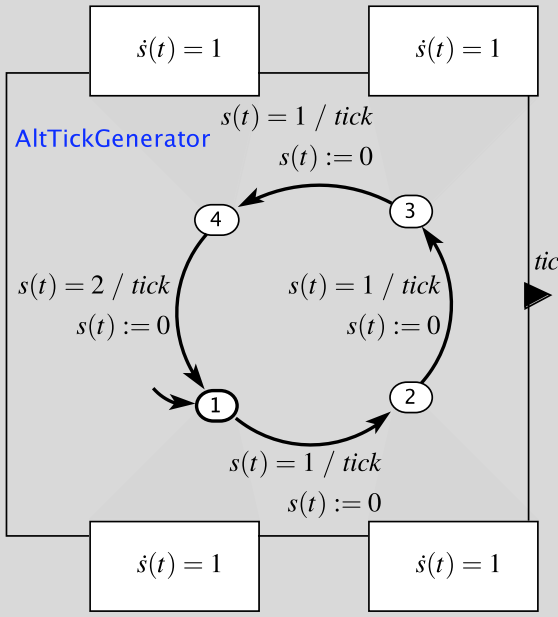
Figure 3.2: State machine for Exercise 7.

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).

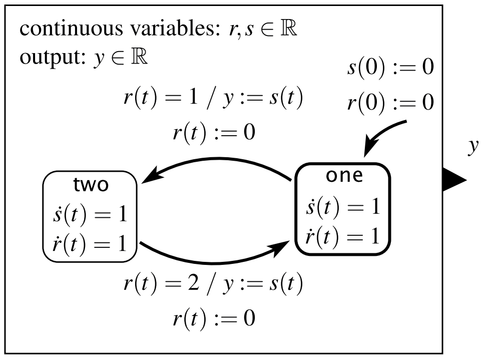
Solution: The following hybrid system will do the job:



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

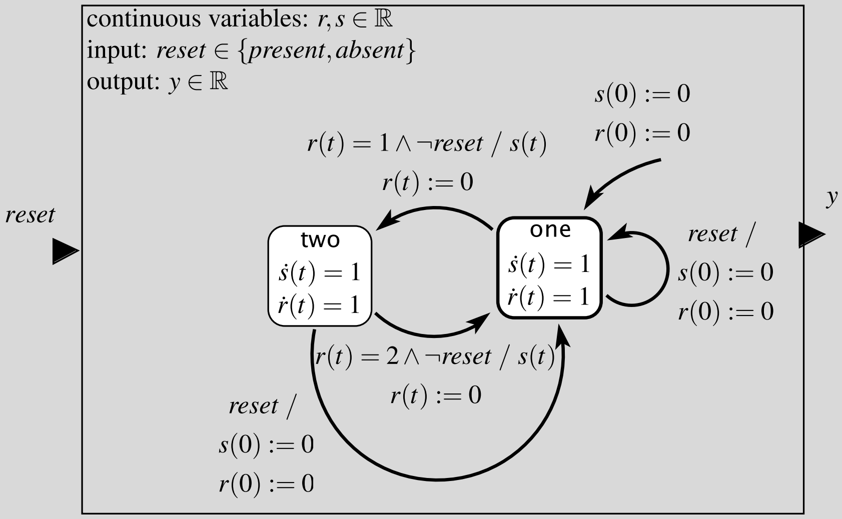
$$1, 3, 4, 6, 7, 9, 10, \cdots .$$

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,

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

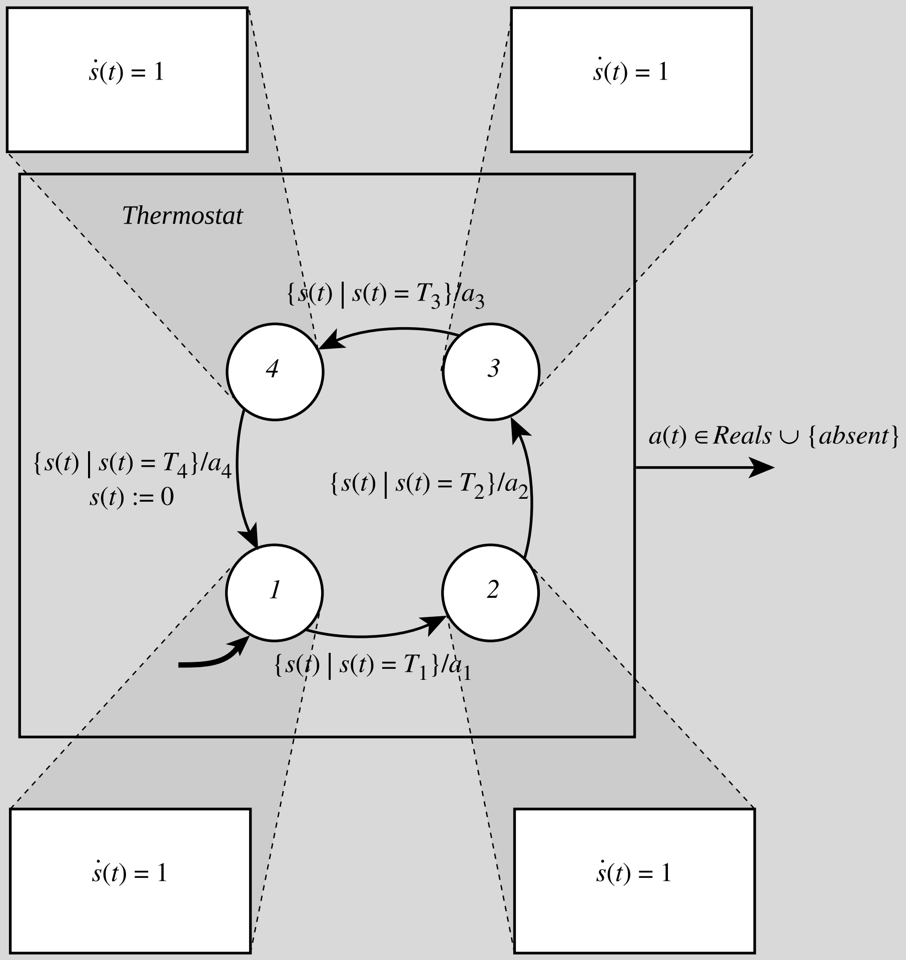
   (b) Assume there is a new pure input *reset*, and that when this input is present, the hybrid system starts over, behaving as if it were starting at time 0 again. Modify the hybrid system from part (a) to do this.

Solution:


continuous variables: $r, s \in \mathbb{R}$
input: $\mathit{reset} \in \{\mathit{present}, \mathit{absent}\}$
output: $y \in \mathbb{R}$
$s(0) := 0$
$r(0) := 0$
$r(t) = 1 \wedge \neg \mathit{reset}$ / $s(t)$
$r(t) := 0$
reset
two
$\dot{s}(t) = 1$
$\dot{r}(t) = 1$
one
$\dot{s}(t) = 1$
$\dot{r}(t) = 1$
reset /
$s(0) := 0$
$r(0) := 0$
y
$r(t) = 2 \wedge \neg \mathit{reset}$ / $s(t)$
$r(t) := 0$
reset /
$s(0) := 0$
$r(0) := 0$

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:


$\dot{s}(t) = 1$
$\dot{s}(t) = 1$
Thermostat
$\{s(t) \mid s(t) = T_3\}/a_3$
4
3
$a(t) \in Reals \cup \{absent\}$
$\{s(t) \mid s(t) = T_4\}/a_4$
$s(t) := 0$
$\{s(t) \mid s(t) = T_2\}/a_2$
1
2
$\{s(t) \mid s(t) = T_1\}/a_1$
$\dot{s}(t) = 1$
$\dot{s}(t) = 1$

You need four modes and one timer.

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

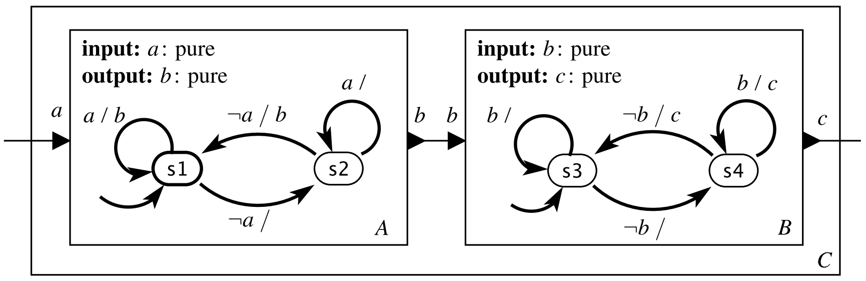
$$(s'_B, o'_B) = update_B(s_B, i_B) \text{ and } s'_A = s_A \text{ and } o'_A = absent$$

or

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

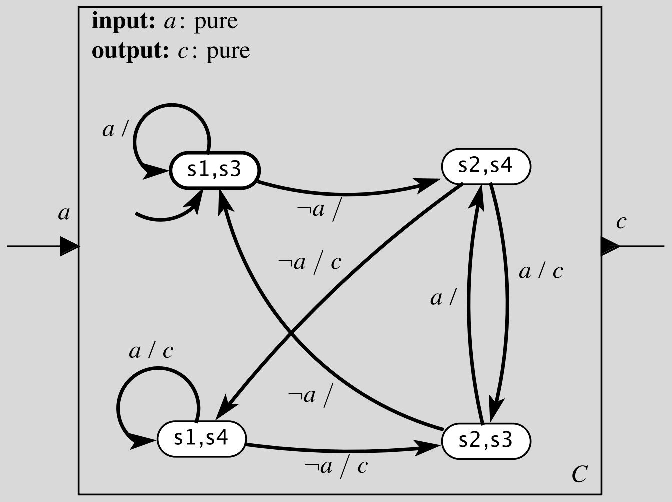
for all $s_A \in States_A$, $s_B \in States_B$, $i_A \in Inputs_A$, and $i_B \in Inputs_B$.

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.

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.

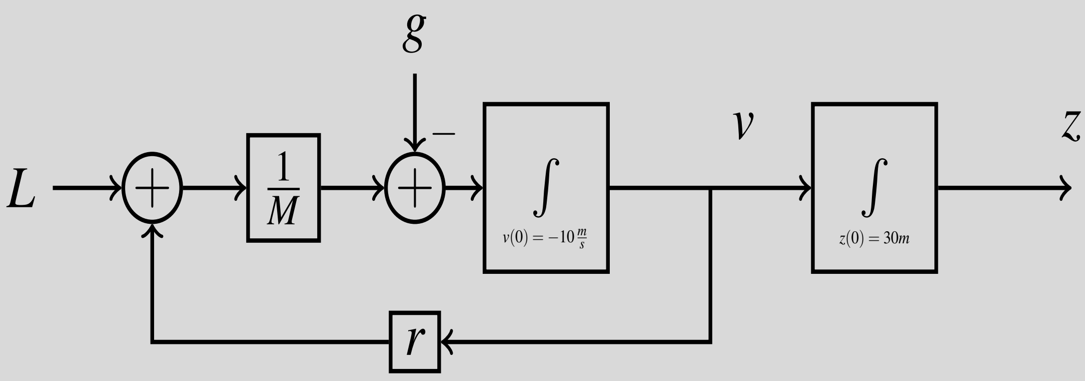
   (b) Given your answer to the previous problem, write down the explicit solution to $z(t)$ and $v(t)$ when the air resistance force is negligible and $r = 0$. At initial time $t = 0$, you are $30m$ above the ground and have an initial velocity of $-10\frac{m}{s}$. *Hint: Write $v(t)$ first then write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

Solution: The accelerometer would read a positive g value.

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.

```
#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.       B
  timer_count = 3;

  // 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.

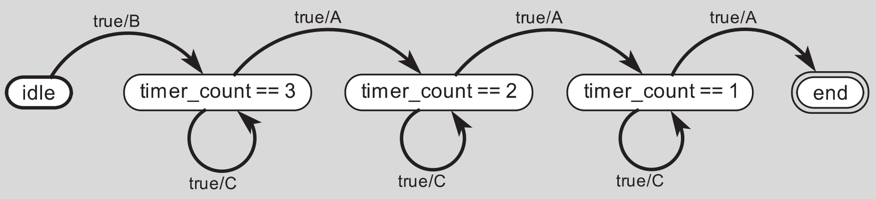
2. Figure 10.1 gives the sketch of a program for an Atmel AVR microcontroller that performs some function repeatedly for three seconds. The function is invoked by calling the procedure foo(). The program begins by setting up a timer interrupt to occur once per second (the code to do this setup is not 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.

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.

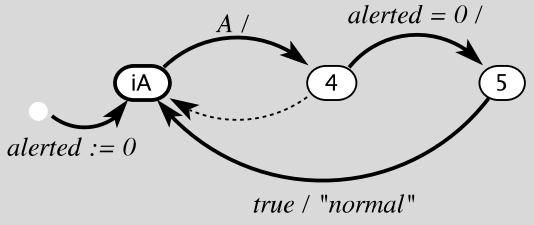
   Solution: No, it does not. If `ISRA` is interrupted after executing line 4 and before executing line 5 by `ISRB`, then the “emergency” display will be quickly overwritten by “normal.”

   (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*`.

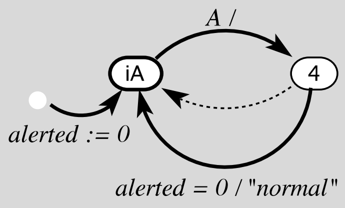
   Solution: In the following, iA indicates that the ISR is idle, 4 indicates that the program is about to execute line 4, and 5 indicates that it is about to execute line 5.

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.

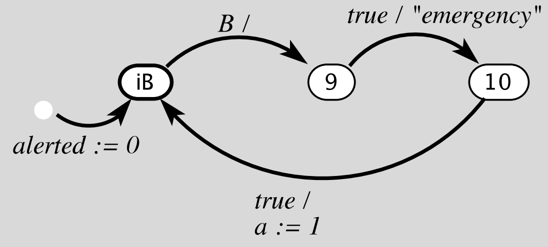
(d) Explain your assumptions about when the state machine in (b) reacts. Is this time triggered, event triggered, or neither?

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`.

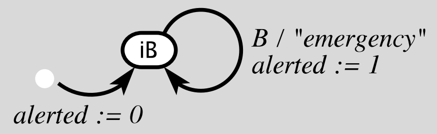
Solution: The following state machine models `ISRB` in a manner similar to the model we constructed for `ISRA`:

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:

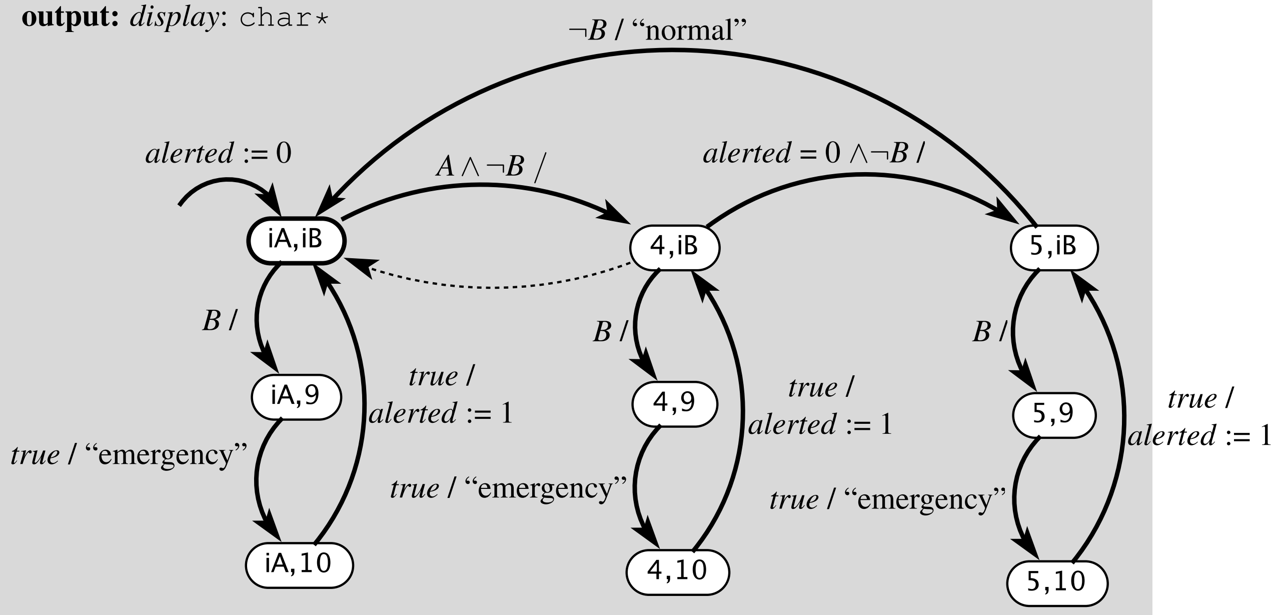
variable: *alerted*: $\{0,1\}$
input: B: pure
output: *display*: `char*`



(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).

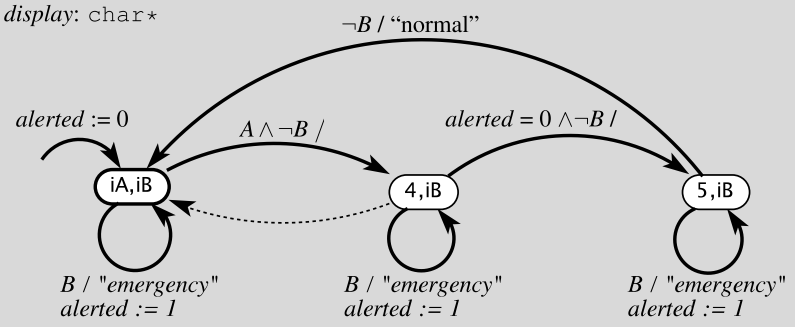
Solution: Using the more complex model of `ISRB` above, we arrive at the following:

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



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

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

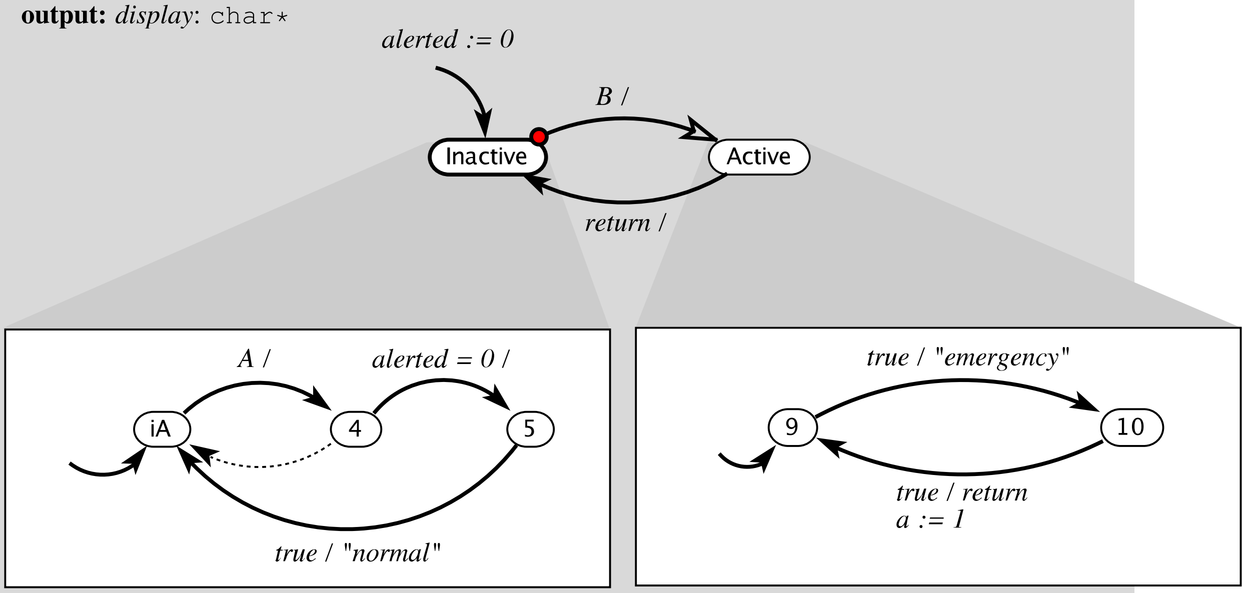


This model shows clearly the bug in part (a). Any time that the self-loop on the right-most state is taken, the bug will occur.

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

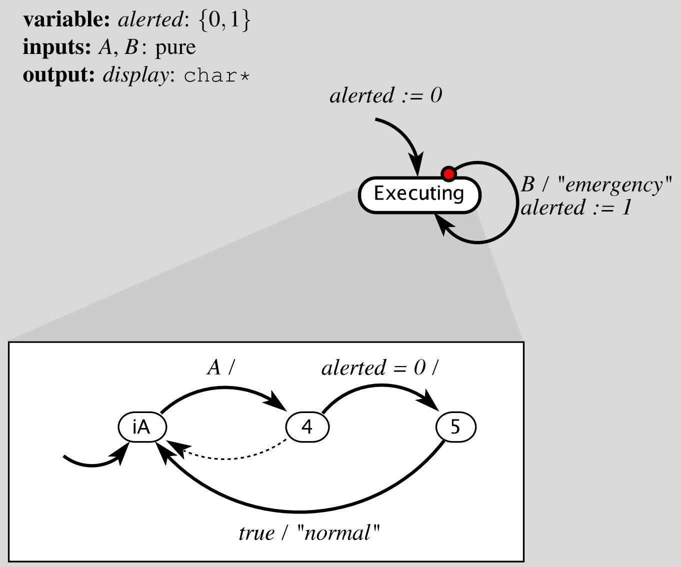
Solution: Using the more complex model of `ISRB` above, we arrive at the following:

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



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.

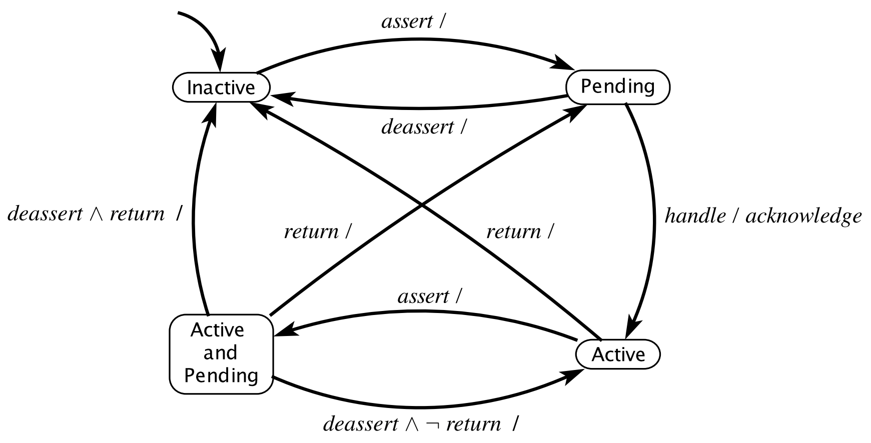
Using the simpler model of `ISRB` above, we arrive at the following:



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:

input: *assert*, *deassert*, *handle*, *return*: pure
output: *acknowledge*



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* $\land$ *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* $\land$ *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.