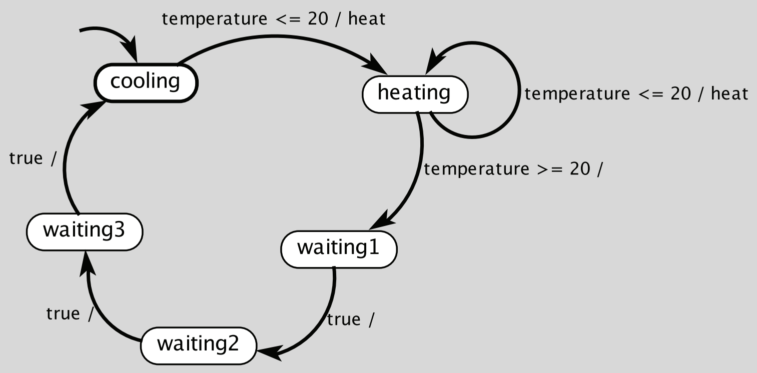
2. Consider a variant of the thermostat of example 3.5. In this variant, there is only one temperature threshold, and to avoid chattering the thermostat simply leaves the heat on or off for at least a fixed amount of time. In the initial state, if the temperature is less than or equal to 20 degrees Celsius, 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?

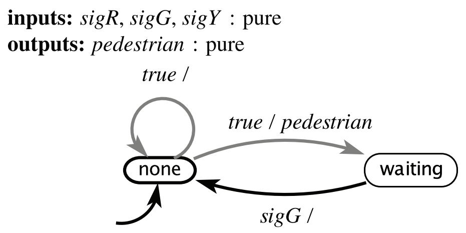
Solution: The FSM has five states. Many alternative designs with more states exist.

   (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.

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:

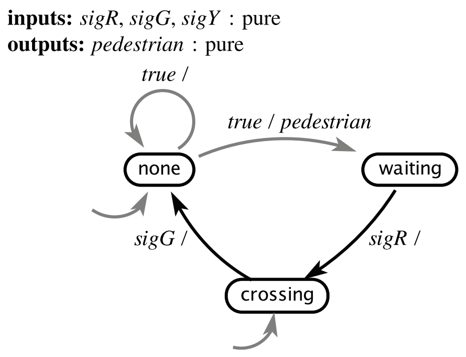

inputs: *sigR*, *sigG*, *sigY* : pure
outputs: *pedestrian* : pure
true /
true / *pedestrian*
none
waiting
sigG /

   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:


inputs: *sigR*, *sigG*, *sigY* : pure
outputs: *pedestrian* : pure
true /
true / *pedestrian*
none
waiting
sigG /
sigR /
crossing

   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.

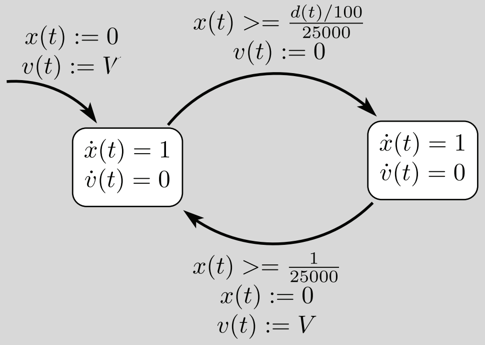
   Solution: Suppose the initial state is chosen to be (red, none) and sometime in the first 60 reactions transitions to (red, waiting). Then eventually the composite machine will transition to (green, waiting), after which all reactions will stutter.

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:

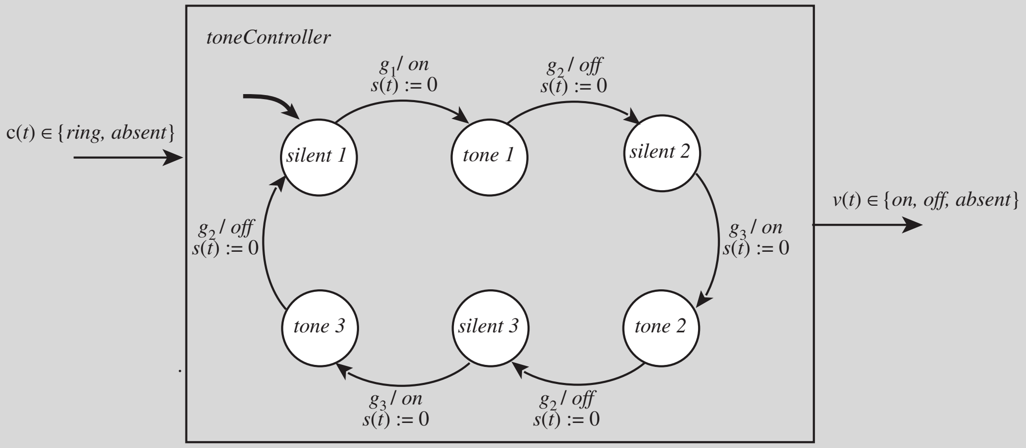
continuous variables: $x(t), y(t) : \mathbb{R}$
input: $d : [0, 100]$
output: $v : \mathbb{R}$



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.

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

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

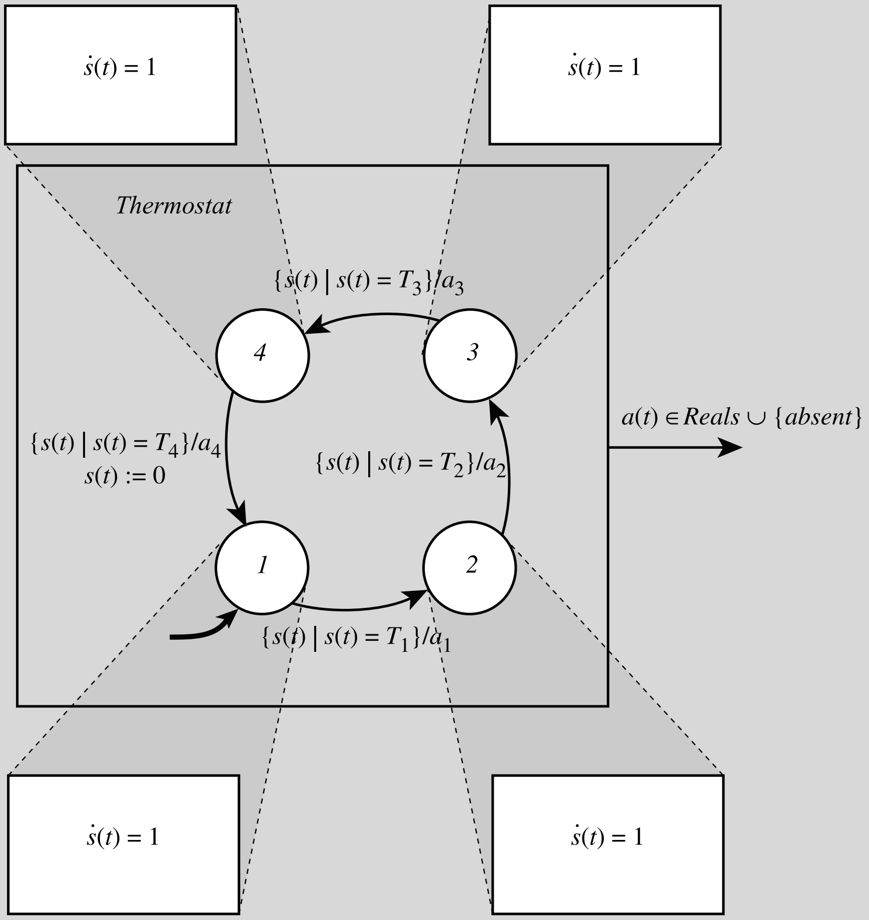
The guards are given by

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

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

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?

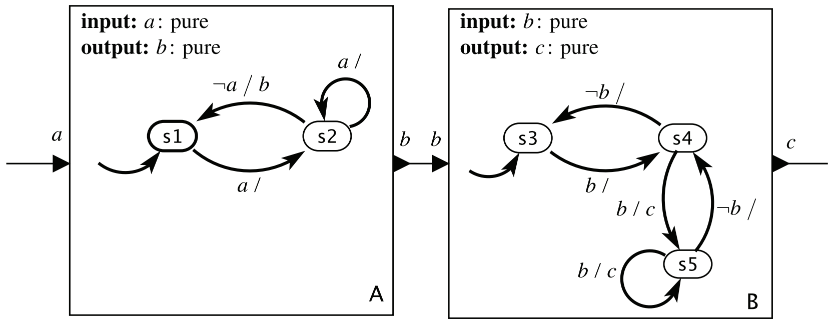
Solution: The following hybrid system will do the job:



$\dot{s}(t) = 1$
$\dot{s}(t) = 1$
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
$\{s(t) \mid s(t) = T_1\}/a_1$
$\dot{s}(t) = 1$
$\dot{s}(t) = 1$

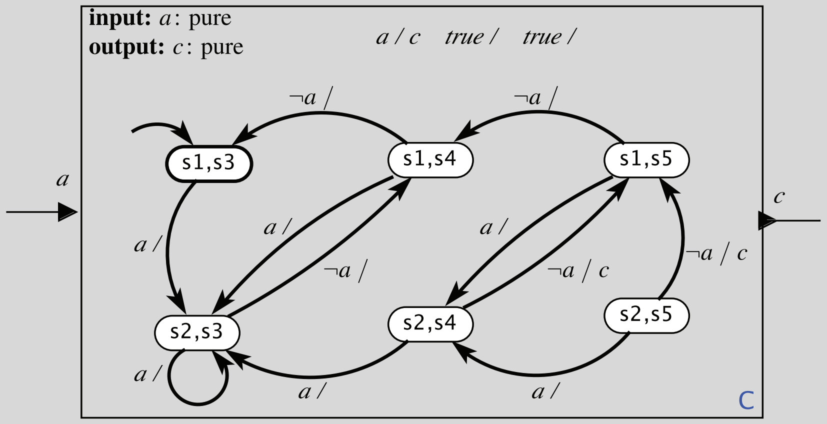
You need four modes and one timer.

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



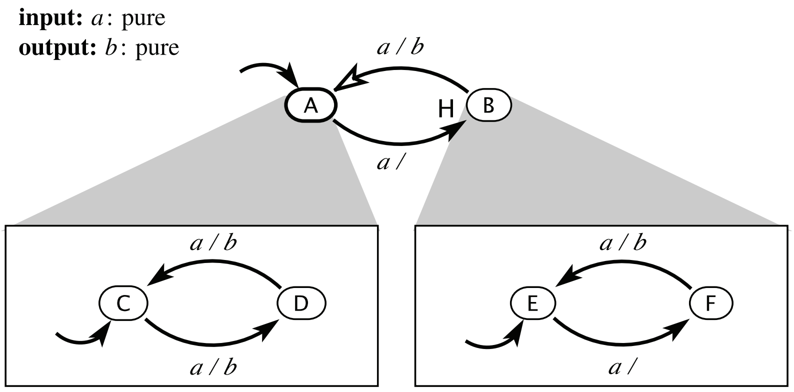
Construct a single state machine *C* representing the composition. Which states of the composition are unreachable?

Solution:



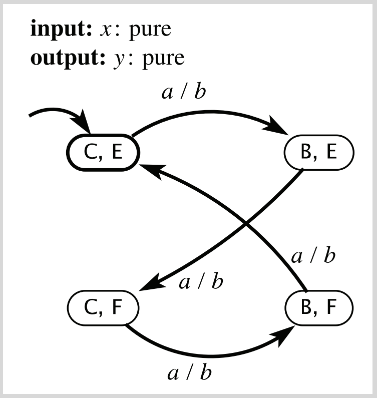
The following states are unreachable: (s1,s5), (s2,s4), and (s2,s5).

5. Consider the following hierarchical state machine:

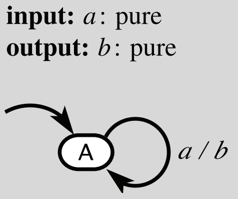


Construct an equivalent flat FSM giving the semantics of the hierarchy. Describe in words the input/output behavior of this machine. Is there a simpler machine that exhibits the same behavior? (Note that equivalence relations between state machines are considered in Chapter 14, but here, you can use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



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.

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

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

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 $b \in \{0,\dots,2^B-1\}$ 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 $a \in \{0,\dots,2^B-1\}$ 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 $\{0,\dots,2^B-1\}$, 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

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

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:

```
0 1 . 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

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.

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*.

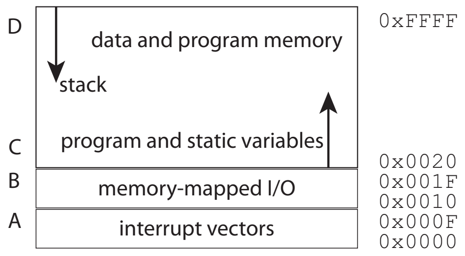
Consider a large array stored in memory.

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.

```c
#include <stdio.h>
#define FOO 0x0010
int n;
int* m;
void foo(int a) {
   if (a > 0) {
     n = n + 1;
     foo(n);
   }
}
int main() {
   n = 0;
   m = (int*)FOO;
   foo(*m);
   printf("n = %d\n", n);
}
```



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 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.

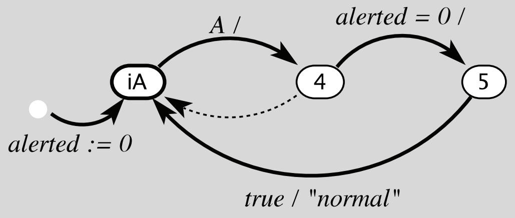
   Solution: No, it does not. If `ISRA` is interrupted after executing line 4 and before executing line 5 by `ISRB`, then the "emergency" display will be quickly overwritten by "normal."

   (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*`.

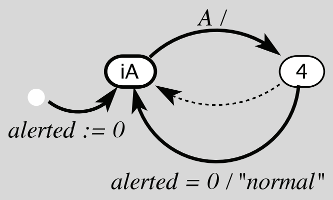
   Solution: In the following, ¡A indicates that the ISR is idle, 4 indicates that the program is about to execute line 4, and 5 indicates that it is about to execute line 5.

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.

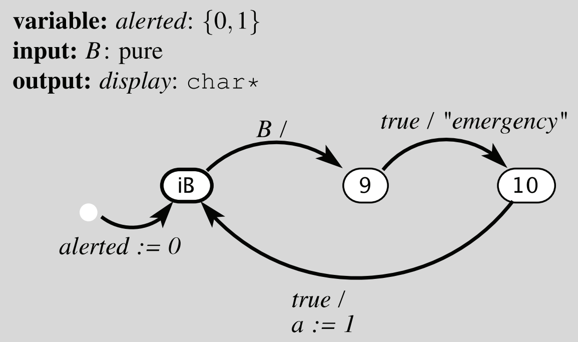
Solution: There are three bubbles and the variable `alerted` has two possible values, so the state space has size 6. Note however that `alerted` is never changed to 1 by this state machine, so considering this state machine in isolation, only three of the six states are reachable.

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

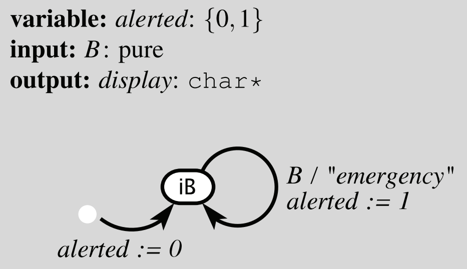
Solution: The machine reacts each time the input *A* is present, and the subsequently when the processor executes one line of code. This could be time triggered if each line of code executes in a fixed time unit, but this is unlikely. Hence, it is neither event triggered nor time triggered.

(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`:

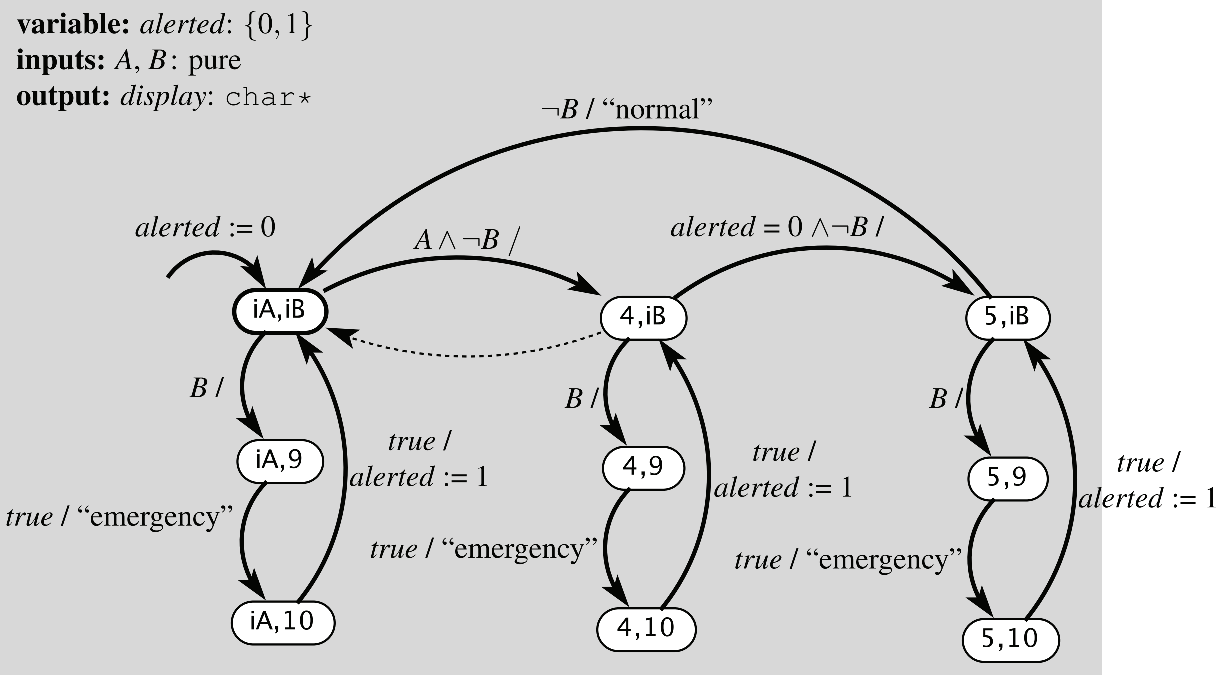


However, in this case, the model really does not need to be this detailed. Since `ISRB` cannot be interrupted, we can model its execution as a single atomic operation, arriving at the very simple model shown below:

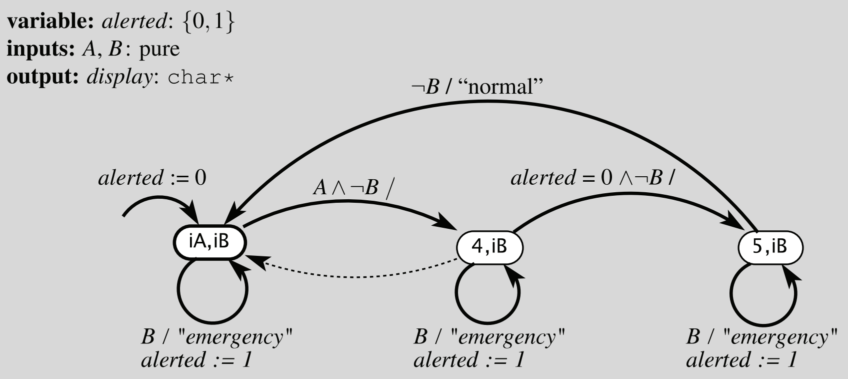

variable: *alerted*: $\{0,1\}$
input: *B*: pure
output: *display*: `char*`
iB
B / "emergency"
alerted := 1
alerted := 0

(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:



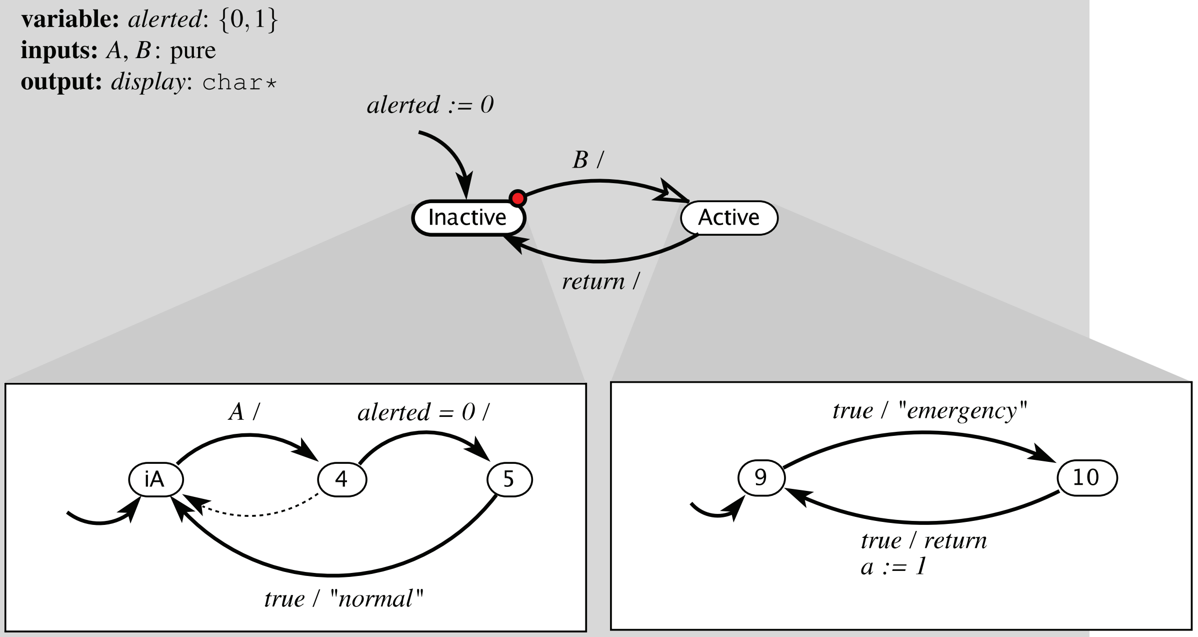
Using the simpler model of `ISRB` above, we arrive at the following:



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.

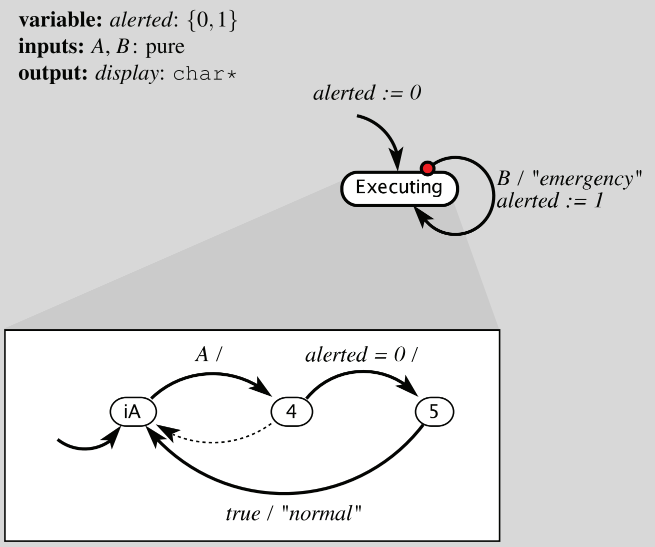
(g) Give an equivalent hierarchical state machine. Use your model to argue the correctness of your answer to part (a).

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:



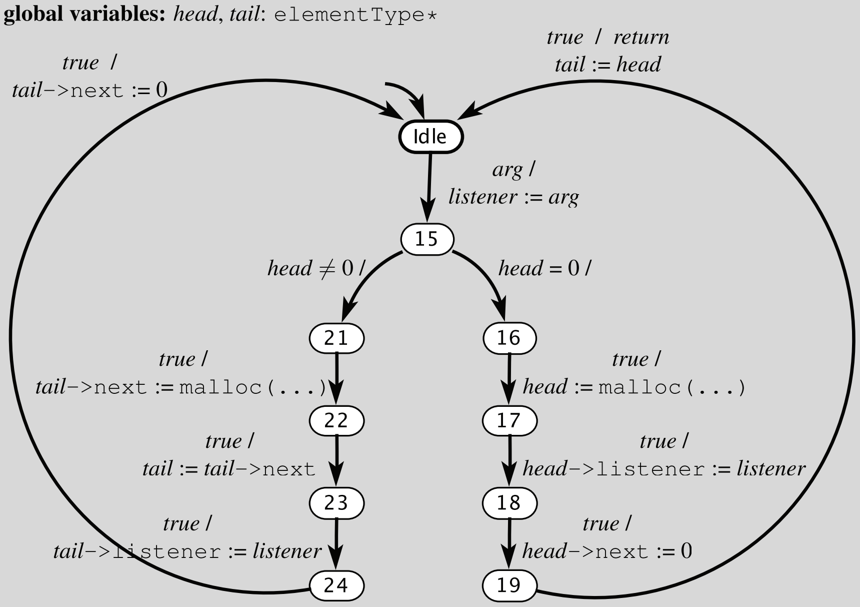
If the preemptive transition is taken while the refinement is in state 5, then the bug from part (a) will occur.

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:

input: *arg*: `notifyProcedure`
output: *return*: pure
local variables: *listener*: `notifyProcedure`
global variables: *head*, *tail*: `elementType*`


true / tail->next := 0
true / return tail := head
Idle
arg / listener := arg
15
head ≠ 0 /
head = 0 /
21
16
true / tail->next := malloc(...)
true / head := malloc(...)
22
17
true / tail := tail->next
true / head->listener := listener
23
18
true / tail->listener := listener
true / head->next := 0
24
19

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:

```c
int a, b;
pthread_mutex_t lock_a
    = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock_b
    = PTHREAD_MUTEX_INITIALIZER;

void procedure1(int arg) {
  pthread_mutex_lock(&lock_a);
  if (a == arg) {
     procedure2(arg);
  }
  pthread_mutex_unlock(&lock_a);
}

void procedure2(int arg) {
  pthread_mutex_lock(&lock_b);
  b = arg;
  pthread_mutex_unlock(&lock_b);
}
```

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:

```c
void procedure1(int arg) {
  pthread_mutex_lock(&lock_b);
  pthread_mutex_lock(&lock_a);
  if (a == arg) {
     procedure2(arg);
  }
  pthread_mutex_unlock(&lock_a);
  pthread_mutex_unlock(&lock_b);
}
```

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:

```c
void procedure1(int arg) {
  int result;
  pthread_mutex_lock(&lock_a);
  result = (a == arg);
  pthread_mutex_unlock(&lock_a);

  if (result) {
    pthread_mutex_lock(&lock_b);
    pthread_mutex_lock(&lock_a);
    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.

```
            procedure2(arg);
          }
          pthread_mutex_unlock(&lock_a);
          pthread_mutex_unlock(&lock_b);
        }
      }
```

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.

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.

```
int size = 0;
pthread_cond_t sent = PTHREAD_COND_INITIALIZER;
pthread_cond_t received = PTHREAD_COND_INITIALIZER;

// Procedure to send a message.
void send(int message) {
    pthread_mutex_lock(&mutex);
    while (size >= 5) {
        pthread_cond_wait(&received, &mutex);
    }
    if (head == 0) {
        head = malloc(sizeof(element_t));
        head->payload = message;
        head->next = 0;
        tail = head;
    } else {
        tail->next = malloc(sizeof(element_t));
        tail = tail->next;
        tail->payload = message;
        tail->next = 0;
    }
    size++;
    pthread_cond_signal(&sent);
    pthread_mutex_unlock(&mutex);
}

// Procedure to get a message.
int get() {
    element_t* element;
    int result;
    pthread_mutex_lock(&mutex);
    // Wait until the size is non-zero.
    while (size == 0) {
        pthread_cond_wait(&sent, &mutex);
    }
    result = head->payload;
    element = head;
    head = head->next;
    free(element);
    size--;
    if (size < 5) {
        pthread_cond_signal(&received);
    }
    pthread_mutex_unlock(&mutex);
    return result;
}
```

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:

```
// Condition variables to signal ready and complete.
int send_ready = 0;
pthread_cond_t send_ready_c = PTHREAD_COND_INITIALIZER;
int receive_ready = 0;
pthread_cond_t receive_ready_c = PTHREAD_COND_INITIALIZER;
int complete = 1;
pthread_cond_t complete_c = PTHREAD_COND_INITIALIZER;

// Procedure to send a message.
void send(int message) {
    pthread_mutex_lock(&mutex);
    complete = 0;
    // Wait until the receiver is ready.
    while (receive_ready == 0) {
        pthread_cond_wait(&receive_ready_c, &mutex);
    }
    buffer = message;
    send_ready = 1;
    pthread_cond_signal(&send_ready_c);
    // Wait until communication completes.
    while (complete == 0) {
        pthread_cond_wait(&complete_c, &mutex);
    }
    pthread_mutex_unlock(&mutex);
}

// Procedure to get a message.
int get() {
    int result;
    pthread_mutex_lock(&mutex);
    // Signal that the receiver is ready.
    receive_ready = 1;
    pthread_cond_signal(&receive_ready_c);
    // Wait until the sender is ready.
    while (send_ready == 0) {
        pthread_cond_wait(&send_ready_c, &mutex);
    }
    receive_ready = 0;
    result = buffer;
    send_ready = 0;
    complete = 1;
    pthread_cond_signal(&complete_c);
    pthread_mutex_unlock(&mutex);
    return result;
}
```

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 counter-example 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.)

13

Invariants and Temporal Logic — Exercises

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

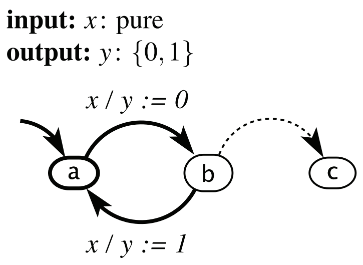
   (a) TRUE or FALSE: If $\mathbf{GF}p$ holds for a state machine A, then so does $\mathbf{FG}p$.

   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.

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:

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

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

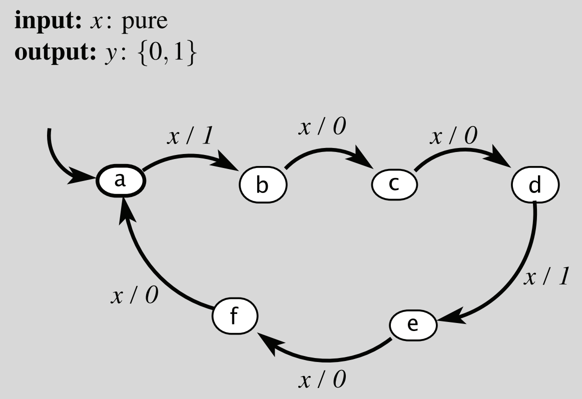
$$States = \{a, b, c, d, e, f\},$$

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:



(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

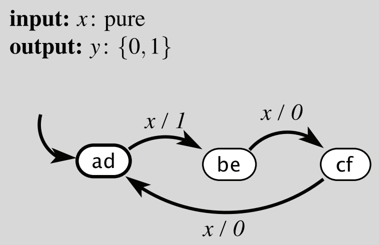
$$x = (p, p, p, \cdots),$$

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:



The bisimulation relation is

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