Principles of Cyber-Physical Systems

Solutions to Exercises

Rajeev Alur

The MIT Press

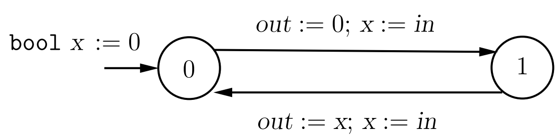
2 Synchronous Model

Solution 2.1: In every odd round, the output of the component `OddDelay` is 0. In every even round, the output equals the value of the input from the previous round. That is, for every $j \geq 1$, if j is an odd number, then the output o_j equals 0, else it equals the input i_{j-1}. Thus the component `OddDelay` alternates between producing the fixed output value 0 and behaving like the component `Delay`. For the given sequence of inputs for the first six rounds, the component has a unique execution shown below, where a state is specified by listing the value of x followed by the value of y:

$$00 \xrightarrow{0/0} 01 \xrightarrow{1/0} 10 \xrightarrow{1/0} 11 \xrightarrow{0/1} 00 \xrightarrow{1/0} 11 \xrightarrow{1/1} 10.$$

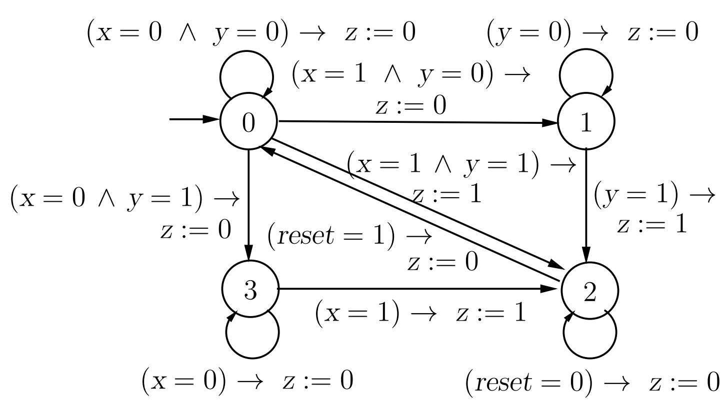
■

Solution 2.2: The extended-state-machine corresponding to the component `OddDelay` is shown below. The modes correspond to the values of the state variable y.



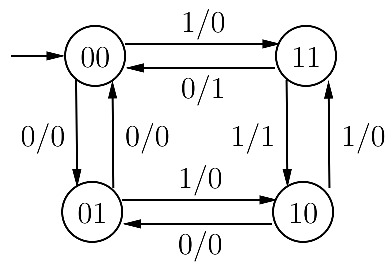
■

Solution 2.3: The extended-state-machine below implements the desired component. The initial mode is 0. When the input x is 1, the component switches to the mode 1, and subsequently when the input y is 1, it switches to the mode 2. Symmetrically, in the initial mode, when the input y is 1, the component switches to the mode 3, and subsequently when the input x is 1, it switches to the mode 2. Note that in the initial mode, if both input variables x and y equal 1, the component directly switches to the mode 2. The transitions to the mode 2 set the output z to 1, and all other transitions set the output to 0. In mode 2, when the condition ($reset = 1$) holds, the component returns to the initial mode.



■

Solution 2.4: The component `OddDelay` is a finite-state component. It has 4 states, and the corresponding Mealy machine is shown below.



■

Solution 2.5: The component `ClockedMax` has three input variables, namely, x of type `nat`, y of type `nat`, and *clock* of type `event`, and a single output variable z of type `event(nat)`. It has no state variables. The reaction description is given by the code

> `if` *clock*? `then` {
>   `if` $(x \geq y)$ `then` $z!\,x$ `else` $z!\,y$
> }.

The component `ClockedMax` is event-triggered and combinational. ■

Solution 2.6: The component `SecondToMinute` has a single input variable *second* of type `event`, a single output variable *minute* of type `event`, and a single state variable x of type `nat`. The initialization is given by $x := 0$, and the reaction description is given by the code

> `if` *second*? `then` {
> $x := x + 1$;
>   `if` $(x = 60)$ `then` { *minute*!; $x := 0$ }
> }.

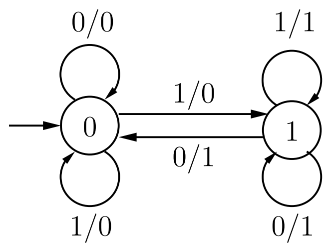
The component `SecondToMinute` is event-triggered. ■

Solution 2.7: The component `ClockedDelay` has two input variables, x of type `bool` and *clock* of type `event`, and a single output variable y of type `event(bool)`. It has a single state variable z of type `bool`. The initialization is given by $z := 0$, and the reaction description is given by the code

`if` *clock*? `then` { $y!\,z$; $z := x$ }

The component `ClockedDelay` is event-triggered. ■

Solution 2.8: The component is nondeterministic. In state 0 (that is, state where the value of x equals 0), the component outputs 0, and if the input is 0, the state stays unchanged, while if the input is 1, the state either stays unchanged or is updated to 1. Symmetrically, in state 1, the component outputs 1, and if the input is 1, the state stays unchanged, while if the input is 0, the state either stays unchanged or is updated to 0. The two-state Mealy machine corresponding to the component is shown below:



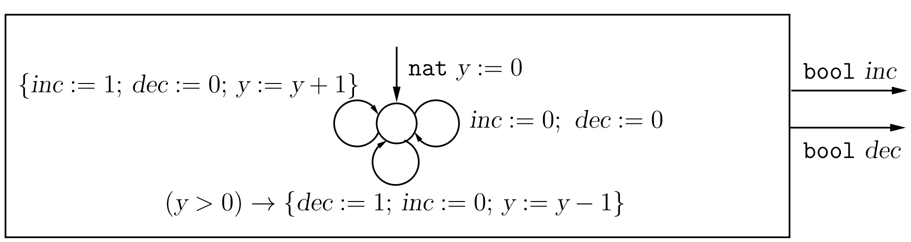
∎

Solution 2.9: The following code can be used as the reaction description of the component `Arbiter`. The value of the local variable x is chosen nondeterministically, and when both the input request events are present, its value is used to decide whether to issue the output event $grant_1$ or to issue the output event $grant_2$.

```
local bool x := choose(0, 1);
  if req1? then
    if req2? then
      if (x = 0) then grant1! else grant2!
    else grant1!
  else if req2? then grant2!.
```

∎

Solution 2.10: The nondeterministic component `CounterEnv` is shown below. Note that when the value of y is zero, the output dec is guaranteed to be 0.



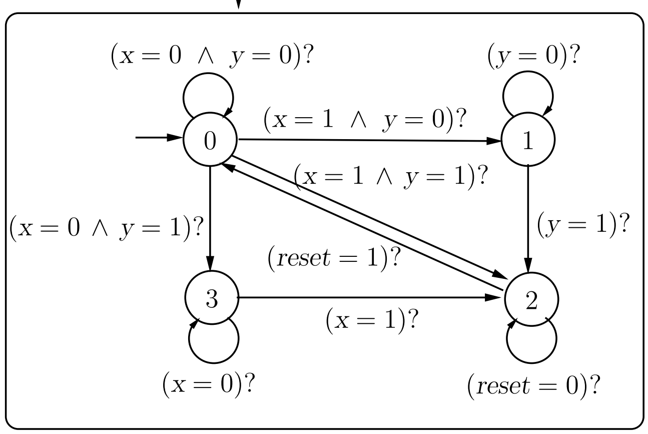
∎

Solution 2.11: The updated reaction description split into two tasks is shown below. The task A_1 computes the value of the output z based on the current state and the inputs x and y. Then, the task A_2 executes to update the state based on all the inputs.

A_1 : $mode, x, y \mapsto z$

```
if [ (mode = 0 ∧ x = 1 ∧ y = 1)
     ∨ (mode = 1 ∧ y = 1)
     ∨ (mode = 3 ∧ x = 1) ]
then z := 1
else z := 0
```

A_2 : $mode, x, y, reset \mapsto mode$



■

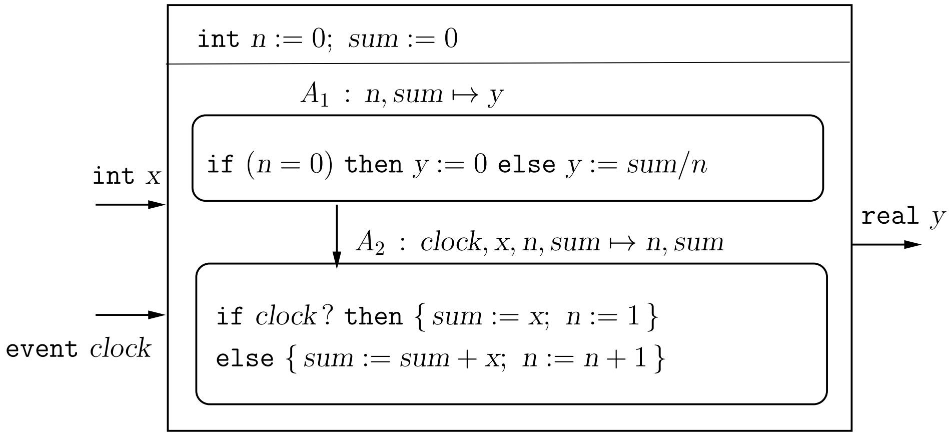
Solution 2.12: Since the task A_2 writes the output z, and z does not await the input x, we can conclude that the task A_2 does not read x and nor does a task that must precede A_2. Since the output y produced by the task A_1 awaits x, it must be the case that A_1 reads x. It follows that there cannot be a precedence edge from the task A_1 to A_2, that is, $A_1 \not\prec A_2$. This means that either there are no precedence constraints (that is, the relation $\prec$ is empty), or the task A_2 precedes A_1 (that is, $A_2 \prec A_1$). ■

Solution 2.13: The reactions of the component are listed below (the output lists the values of y and z in that order):

$$0 \xrightarrow{0/00} 0; \quad 0 \xrightarrow{1/00} 1; \quad 0 \xrightarrow{1/01} 1; \quad 1 \xrightarrow{0/10} 0; \quad 1 \xrightarrow{0/11} 0; \quad 1 \xrightarrow{1/11} 1.$$

The output y *does not* await the input x. The output z awaits the input x. ■

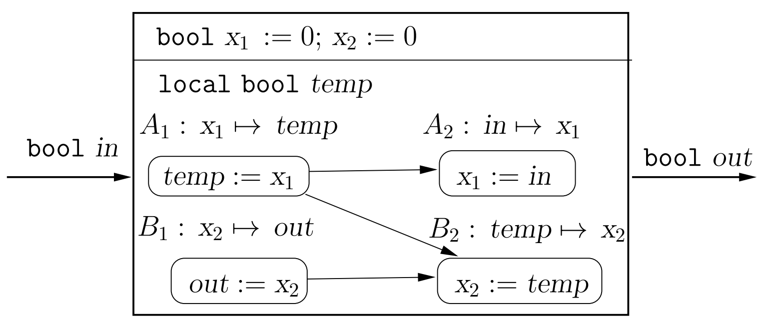
Solution 2.14: The component `ComputeAverage` is shown below. It maintains an integer state variable n that tracks the number of rounds elapsed since the presence of the input event *clock*, and an integer state variable *sum* that maintains the sum of the values of the input variable x since the presence of the input event *clock*. The task A_1 computes the value of the output y based on the current state, and the task A_2 then updates the state variables based on the inputs.



■

Solution 2.15: The component `ClockedDelayComparator` has input variables in_1 and in_2 of type `nat`, an input event variable *clock*, and an output variable y of type `event(bool)`. Suppose the input *clock* is present during rounds, say, $n_1 < n_2 < n_3 < \cdots$. Then, in round n_1, the output y is 0; and in round n_{j+1}, for each j, the output equals 1 if the value of the input variable in_1 in the round n_j is greater than or equal to the value of the input variable in_2 in the round n_j, and equals 0 otherwise; and in the remaining rounds (that is, rounds during which the input event *clock* is absent), output is absent. ■

Solution 2.16: The component `DoubleSplitDelay` has input variable *in*, output variable *out*, state variables x_1 and x_2, and local variable *temp*, all of type `bool`. Its reaction description consists of 4 tasks as shown below.



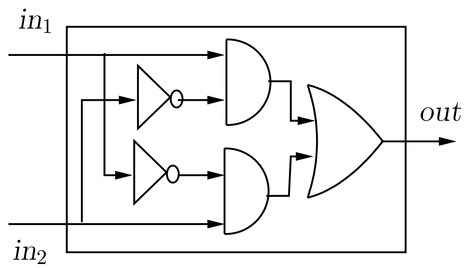
The output variable *out* does not await the input variable *in*. ■

Solution 2.17: The desired component `SecondToHour` is defined as

$$(\texttt{SecondToMinute} \parallel \texttt{SecondToMinute}[\,minute \mapsto hour\,][\,second \mapsto minute\,])\backslash minute.$$

■

Solution 2.18 : For the component `SyncXor`, its output *out* should be 1 exactly when only one of the inputs in_1 and in_2 is 1. Thus, the output *out* corresponds to the Boolean expression $(in_1 \wedge \neg\, in_2) \vee (\neg\, in_1 \wedge in_2)$. The desired output is computed by the following combinational circuit that uses 2 instances of `SyncAnd`, 2 instances of `SyncNot`, and one instance of `SyncOr`.



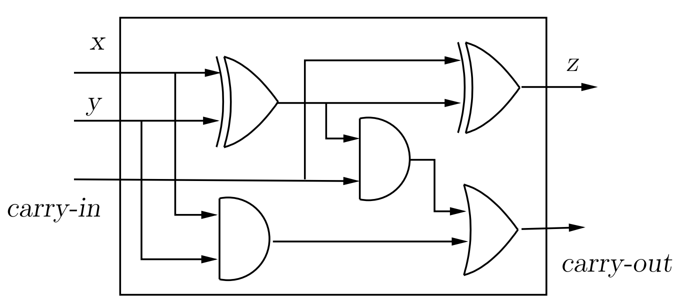
■

Solution 2.19 : The parity circuit with n inputs is defined inductively. For $n = 2$, the desired functionality coincides with that of the XOR gate. Thus, the component $\texttt{Parity}_2$ is same as the component `SyncXor` from exercise 2.18. Having defined the circuit $\texttt{Parity}_{n-1}$ that computes the parity of $n-1$ input variables, now we wish to construct the circuit $\texttt{Parity}_n$ with input variables $in_1, \ldots in_n$ and output *out*. Observe that the output should be 1 exactly when either the input in_n is 1 and the parity of the first $n-1$ input variables is even, or the input in_n is 0 and the parity of the first $n-1$ input variables is odd. Thus, the desired circuit is defined as:

$$\texttt{Parity}_n \;=\; (\texttt{Parity}_{n-1}[\,out \mapsto temp\,] \parallel \texttt{SyncXor}[\,in_1 \mapsto temp\,][\,in_2 \mapsto in_n\,])\backslash temp.$$

Note that the circuit $\texttt{Parity}_n$ uses one more instance of `SyncXor` than the component $\texttt{Parity}_{n-1}$, and thus, $n-1$ total instances of `SyncXor`. ■

Solution 2.20 : The combinational circuit `1BitAdder`, shown below, uses 2 instances of `SyncAnd`, one instance of `SyncOr`, and 2 instances of `SyncXor`. Verify that the output z is 1 when an odd number of the input variables equal 1, and the output *carry-out* is 1 when two or more of the input variables equal 1.



The 3-bit synchronous adder is obtained by composing 3 instances of `1BitAdder`, and is defined as the following component:

$$\begin{array}{rcl}
\texttt{Bit0} & = & \texttt{1BitAdder}[x \mapsto x_0][y \mapsto y_0][z \mapsto z_0][\mathit{carry\text{-}out} \mapsto c_0] \\
\texttt{Bit1} & = & \texttt{1BitAdder}[x \mapsto x_1][y \mapsto y_1][z \mapsto z_1][\mathit{carry\text{-}in} \mapsto c_0][\mathit{carry\text{-}out} \mapsto c_1] \\
\texttt{Bit2} & = & \texttt{1BitAdder}[x \mapsto x_2][y \mapsto y_2][z \mapsto z_2][\mathit{carry\text{-}in} \mapsto c_1] \\
\texttt{3BitAdder} & = & (\texttt{Bit0} \parallel \texttt{Bit1} \parallel \texttt{Bit2}) \setminus \{c_0, c_1\}.
\end{array}$$

■

Solution 2.21: To implement the desired functionality, the component `SetSpeed` now maintains a state variable *mode* that can be either `on`, `paused`, or `off`. The initialization is given by:

$$\texttt{nat}\ x := \texttt{minSpeed};\ \{\texttt{on}, \texttt{paused}, \texttt{off}\}\ \mathit{mode} := \texttt{off}$$

The reaction description is revised to:

```
if cruise? then
  if (mode = off) then {
    mode := on;
    if (speed < minSpeed) then s := minSpeed
    else if (speed > maxSpeed) then s := maxSpeed
      else s := speed
    }
   else mode := off
else if (mode = on ∧ pause?) then mode := paused
   else if (mode = on ∧ dec? ∧ s > minSpeed) then s := s − 1
     else if (mode = on ∧ inc? ∧ s < maxSpeed) then s := s + 1
       else if (mode = paused ∧ pause?) then mode := on
if (mode = on) then cruiseSpeed := s
```

■

Solution 2.22: Consider a node with identifier n, and suppose at the beginning of a round, say k, the value of its *id* variable is m, and suppose this value stays unchanged during this round. According to the reaction description of figure 2.35, the node n transmits the same value m to all its neighbors during rounds k as well as $k+1$. For every node ℓ such that there is an edge from the node n to node ℓ, the updated value of the variable *id* for the node ℓ is guaranteed to be $\geq m$ at the end of the round k. As a result, the output value of the node n during round $k+1$ does not cause the node ℓ to change its value in round $k+1$, and hence, we can modify the protocol so that the node n leaves its output absent in such a case.

The modified component `SyncLENode`, maintains, in addition to the state variables *id* and *r*, a Boolean variable *new*. The component should transmit an output value only when the variable *new* equals 1, and in each round, if the state variable *id* is modified, the variable *new* is set to 1. The initialization is given by

$$\texttt{nat}\ \mathit{id} := \texttt{myID};\ r := 1;\ \texttt{bool}\ \mathit{new} := 1$$

The update code the task A_1 is changed to:

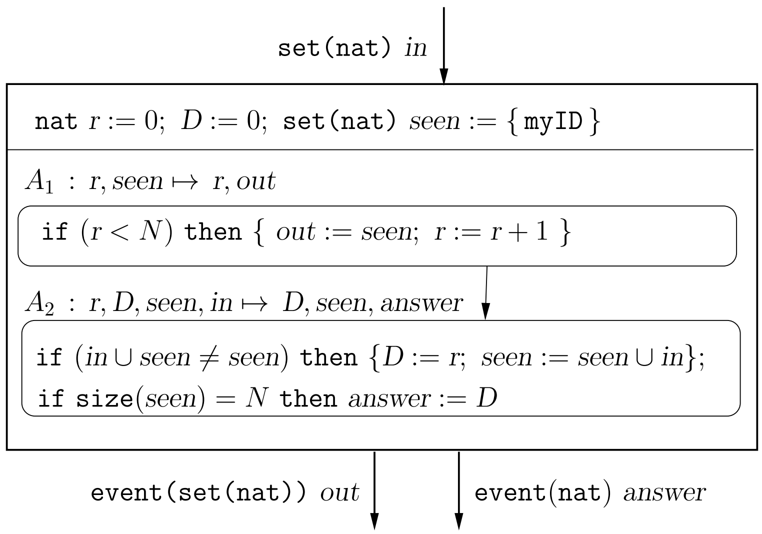
```
if (r < N) then {
  if (new = 1) then out := id;
  r := r + 1 }.
```

The first statement in the update code for the task A_2 is replaced by (the subsequent statement that assigns the value of the output variable *status* stays unchanged):

```
if (in ≠ ∅) then
  if (id < max in) then
    { id := max in; new := 1 }
  else new := 0.
```

■

Solution 2.23: Each node n maintains a state variable r that keeps track of the number of rounds: r is initialized to 0, and is incremented by 1 in each round. To compute the desired value D_n, the node n needs to know, for each node m, the length of the shortest path from the node m to node n. For this purpose, the node n maintains a state variable *seen* which contains the set of identifiers of all the nodes that the node n has encountered so far. Initially, the set *seen* contains just the node n itself. In each round, each node transmits this set to its neighbors, and updates the value of *seen* to include all the identifiers it receives from its neighbors. Thus, if the shortest path from the node m to node n is k, then, during the k-th round, the node n will receive m as one of the identifiers for the first time, and will add it to the set *seen*. To compute the desired value D_n, the node n maintains a state variable D: it is initialized to 0, and during the k-th round, if the node n receives an identifier that it has not seen before (that is, the input contains a value not in the current value of the variable *seen*), the variable D is updated to k. If the network is strongly connected, then within N rounds, the set *seen* will contain all the node identifiers: when the size of this set equals N, the value of the variable D equals the desired diameter D_n, and the node n can output this answer. The synchronous component that implements this protocol is shown below:


set(nat) in
nat r := 0; D := 0; set(nat) seen := { myID }
A1 : r, seen ↦ r, out
if (r < N) then { out := seen; r := r + 1 }
A2 : r, D, seen, in ↦ D, seen, answer
if (in ∪ seen ≠ seen) then {D := r; seen := seen ∪ in};
if size(seen) = N then answer := D
event(set(nat)) out
event(nat) answer

■

3 Safety Requirements

Solution 3.1: The transition system $\texttt{Mult}(m, n)$ has 3 state variables: *mode* of the enumerated type $\{\texttt{loop}, \texttt{stop}\}$, x of type `nat`, and y of type `nat`. The sole initial state is $(\texttt{loop}, m, 0)$. The set of transitions is defined as follows: for every positive natural number a and every natural number b, there is a transition from the state $(\texttt{loop}, a, b)$ to the state $(\texttt{loop}, a-1, b+n)$; and for every natural number b, there is a transition from the state $(\texttt{loop}, 0, b)$ to the state $(\texttt{stop}, 0, b)$. The transition system is deterministic, and has the following execution:

$$(\texttt{loop}, m, 0) \rightarrow (\texttt{loop}, m{-}1, n) \rightarrow \cdots (\texttt{loop}, m{-}j, n{\cdot}j) \rightarrow \cdots (\texttt{loop}, 0, m{\cdot}n) \rightarrow (\texttt{stop}, 0, m{\cdot}n).$$

Along this execution, only the last state satisfies the property $(\mathit{mode} = \texttt{stop})$, and in this state the property $(y = m \cdot n)$ also holds. Thus, the implication $(\mathit{mode} = \texttt{stop}) \rightarrow (y = m \cdot n)$ is satisfied in every reachable state, and is an invariant of the system. ■

Solution 3.2: Each state is denoted by listing the values of the variables *west*, *east*, mode_W, and mode_E, in that order. We use a, w, b, g, and r, as abbreviations for the values `away`, `wait`, `bridge`, `green`, `red`, respectively. Then, the initial state is *ggaa*, and has transitions to itself and to the states *rgaw*, *grwa*, and *rgww*. To compute the set of reachable states, we need to explore transitions from these three newly discovered states, and keep repeating till no new states are found to be reachable. It turns out that the following 13 states are reachable:

$\{ggaa,\ rgaw,\ grwa,\ rgww,\ rgab,\ rrwb,\ grba,\ rrbw,\ rgwb,\ ggwa,\ ggba,\ rgbw,\ rgbb\}$.

■

Solution 3.3: The task A_{31} reads out_E, writes near_E, and has the update-code

> `if` out_E ? `arrive then` $\mathit{near}_E := 1$;
> `if` out_E ? `leave then` $\mathit{near}_E := 0$.

The task A_{32} reads out_W, writes near_W, and has the update-code

> `if` out_W ? `arrive then` $\mathit{near}_W := 1$;
> `if` out_W ? `leave then` $\mathit{near}_W := 0$.

The task A_{33} reads the state variables near_E and *west*, and writes the state variable *east* using the update-code

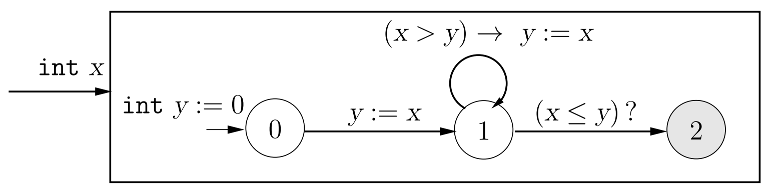
> `if` $\neg\, \mathit{near}_E$ `then` *east* := `red`
> `else if` $(\mathit{west} = \texttt{red})$ `then` *east* := `green`.

Finally, the task A_{34} reads the state variables near_W and *east*, and writes the state variable *west* using the update-code

> `if` $\neg\, \mathit{near}_W$ `then` *west* := `red`
> `else if` $(\mathit{east} = \texttt{red})$ `then` *west* := `green`.

The precedence constraints are: $A_{31} \prec A_{33}$, $A_{32} \prec A_{34}$, and $A_{33} \prec A_{34}$. The input-output dependencies of the revised controller coincide with the description in figure 3.8: in both cases, the outputs do not await of any of the inputs. In the revised description, observe that the tasks A_{31} and A_{32} are independent, and can be executed concurrently. Thus, if the implementation platform allows parallelism, the revised description can be executed more efficiently. ■

Solution 3.4 : The monitor, shown below, maintains a state variable y to record the value of the input from the preceding round. The monitor enters the mode 2 exactly when the desired safety requirement gets violated.



■

Solution 3.5 : The system `RailRoadSystem2` satisfies the requirement specified by the monitor `WestFairMonitor`: there is no execution of this system that can lead the monitor to the error state 3. For justification, consider an execution $s_0, \ldots s_i, s_{i+1}, \ldots s_j, s_{j+1}$ of the system `RailRoadSystem2` $\|$ `WestFairMonitor` such that the monitor switches from mode 0 to mode 1 during the transition $s_i \rightarrow s_{i+1}$, is in mode 1 in states $s_{i+1}, \ldots s_j$, and switches from mode 1 to mode 2 during the transition $s_j \rightarrow s_{j+1}$. For this to happen, the condition out_W ? `arrive` must hold during the ith round. Based on the reaction descriptions of `Train`$_W$ and `Controller2`, the value of $mode_W$ must be `wait` and $near_W$ must be 1 in state s_{i+1}. Since the monitor stays in mode 1 in states $s_{i+1}, \ldots s_j$, the condition ($signal_W$ = `green`) is false during this stretch, and thus, the value of $mode_W$ must be `wait` and $near_W$ must be 1 in all the states $s_{i+1}, \ldots s_j$. By assumption, during the jth round, the condition out_E ? `leave` holds. Executing the reaction description of `Controller2` under this condition with $near_W = 1$ sets the variable $west$ to `green`. As a result, in the next round, the output $signal_W$ of the controller is set to `green`, and this would force the monitor `WestFairMonitor` to switch to mode 0, preventing it from entering the error mode 3. ■

Solution 3.6 : The system has a single execution given by (a state is specified by listing the values of x, y, and z, in that order):

$$(0,0,0) \ \rightarrow\ (1,0,1) \ \rightarrow\ (1,1,0) \ \rightarrow\ (2,1,1) \ \rightarrow\ (2,2,0) \ \rightarrow \cdots$$

The formula φ given by $(x = y \ \vee\ x = y + 1)$ holds in every state of this execution, and is an invariant of the system.

The formula φ, however, is not an inductive invariant. The state $(1,0,0)$ satisfies the formula φ, and has a transition to the state $(2,0,1)$, which *does not* satisfy the formula φ.

Consider the formula ψ given by $(z = 0 \ \wedge \ x = y) \vee (z = 1 \ \wedge \ x = y+1)$. Observe that if a state s satisfies ψ, it must satisfy one of the disjuncts in ψ, and thus, must satisfy either $(x = y)$ or $(x = y+1)$, and thus, must satisfy φ. Thus, the property ψ is stronger than φ. The initial state $(0,0,0)$ satisfies ψ. Consider a state s that satisfies ψ. Then s satisfies either $(z = 0 \ \wedge \ x = y)$ or $(z = 1 \ \wedge \ x = y+1)$. In the former case, executing a transition from the state s increments x and sets z to 1, and thus, the resulting state satisfies $(z = 1 \ \wedge \ x = y+1)$. By a similar reasoning if the state s satisfies $(z = 1 \ \wedge \ x = y+1)$, then executing one transition from it leads to a state that satisfies $(z = 0 \ \wedge \ x = y)$. It follows that if there is a transition from the state s to state t, then the state t must satisfy ψ. Thus, the property ψ is an inductive invariant. ■

Solution 3.7: The transition system $\texttt{Mult}(m,n)$ has a transition from state $s = (\texttt{loop}, 0, k)$ to state $t = (\texttt{stop}, 0, k)$, for every natural number k. If $k \neq m \cdot n$, then the state s satisfies the property φ given by $(\mathit{mode} = \texttt{stop}) \rightarrow (y = m \cdot n)$, but the state t does not. It follows that the property φ is not an inductive invariant.

Consider the property ψ given by

$$[(\mathit{mode} = \texttt{loop}) \ \wedge \ y = (m - x) \cdot n] \ \vee \ [(\mathit{mode} = \texttt{stop}) \ \wedge \ (y = m \cdot n)].$$

If a state s satisfies the formula ψ, and the value of mode in state s is $\texttt{stop}$, then the state s must satisfy $(y = m \cdot n)$. It follows that a state satisfying the property ψ must satisfy the property φ.

The initial state $(\texttt{loop}, m, 0)$ satisfies the formula ψ. Now consider a state s satisfying ψ. Suppose the value of mode in state s is $\texttt{loop}$. We know that the condition $s(y) = (m - s(x)) \cdot n$ holds. If $s(x) > 0$ then executing a transition in state s leaves the mode unchanged, decrements x, and increases y by n. That is, $t(\mathit{mode}) = \texttt{loop}$, $t(x) = s(x) - 1$, and $t(y) = s(y) + n$. It is easy to establish that $t(y) = (m - t(x)) \cdot n$ also holds, and thus the state t satisfies ψ. If $s(x) = 0$, then the condition $s(y) = m \cdot n$ holds, and executing a transition in state s updates the mode to $\texttt{stop}$ and leaves the variables x and y unchanged. In this case also, the resulting state t satisfies the property ψ. If the value of mode in state s is $\texttt{stop}$, then there is no transition out of state s. It follows that the property ψ is preserved by transitions of the system $\texttt{Mult}(m,n)$, and it is an inductive invariant. ■

Solution 3.8: The state $(\texttt{on}, 0)$ satisfies the property φ, and has a transition to the state $(\texttt{off}, -1)$ which does not satisfy the property φ. This shows that the property is not an inductive invariant.

Consider the property ψ given by

$$(mode = \texttt{off} \ \wedge \ x \geq 0) \ \vee \ (mode = \texttt{on} \ \wedge \ x > 0).$$

We will first show that the property ψ is stronger than the property φ. Consider a state s that satisfies ψ. Depending of the value of *mode* in state s, either $x \geq 0$ or $x > 0$ holds in the state s. In either case the condition φ is satisfied.

The initial state $(\texttt{off}, 0)$ satisfies the property ψ. Now consider a state s that satisfies ψ. To prove that the property ψ is an inductive invariant, we need to establish that, if there is a transition from the state s to state t, then the state t also satisfies ψ. Suppose $s = (\texttt{off}, a)$. Then, it must be the case that $a \geq 0$. The state s has 2 successor states $t_1 = (\texttt{off}, a+1)$ and $t_2 = (\texttt{on}, a+1)$. Clearly, the state t_1 satisfies $(mode = \texttt{off} \ \wedge \ x \geq 0)$, and the state t_2 satisfies $(mode = \texttt{on} \wedge x > 0)$. Thus, both the states satisfy ψ. Now suppose $s = (\texttt{on}, a)$. Then, it must be the case that $a > 0$. The state s has only one outgoing transition to the state $t = (\texttt{off}, a-1)$, and this state satisfies $(mode = \texttt{off} \ \wedge \ x \geq 0)$ (since $a - 1 \geq 0$), and thus, the property ψ. ■

Solution 3.9: 1. The property is an invariant, and in fact, and an inductive invariant. In the initial state $near_E$ equals 0 and $mode_E$ is `away`. Consider an arbitrary state where $near_E$ equals 0 and $mode_E$ is `away`. The value of $mode_E$ changes (to `wait`) exactly when the condition (out_E ? `arrive`) holds, but this coincides with the condition under which the controller changes $near_E$ to a non-zero value, and thus, in the resulting state both the conditions $(near_E = 0)$ and $(mode_E = \texttt{away})$ are false, and the equivalence continues to hold. Now consider an arbitrary state where $near_E$ equals 1 and $mode_E$ is not `away`. During a transition the condition $(mode_E = \texttt{away})$ can become true only when (out_E ? `leave`) holds, which is precisely the condition under which the controller changes $near_E$ to 0.

2. The property is an invariant, but is not an inductive invariant. Consider a state in which $mode_E$ equals `bridge`, *east* equals `green`, and $near_E$ equals 0 (such a state is actually unreachable). This state satisfies the property, This state has a transition to a state in which $mode_E$ stays unchanged, $near_E$ stays unchanged, but *east* gets updated to `red`, violating the property.

3. The property is an inductive invariant. The initial state satisfies this property. By examining the update-code in figure 3.8 observe that the variable *east* is updated to `green` only under the condition $(west = \texttt{red})$, and the variable *west* is updated to `green` only under the condition $(east = \texttt{red})$. It follows that executing this code in a state where at least one of *east* or *west* equals `red` cannot lead to a state with both variables equal to `green`. ■

Solution 3.10: The reactive component `Switch` has 12 reachable states: $(\texttt{off}, 0)$, and $(\texttt{on}, n)$, for $0 \leq n \leq 10$. The state $(\texttt{off}, 0)$ has two transitions, to itself and to $(\texttt{on}, 0)$. For $0 \leq n < 10$, the state $(\texttt{on}, n)$ has two transitions, to $(\texttt{off}, 0)$ and to $(\texttt{on}, n+1)$. The state $(\texttt{on}, 10)$ has a single transition to $(\texttt{off}, 0)$. ■

Solution 3.11: Consider the modified algorithm that does not maintain the variable *Reach* to track which states have been encountered so far. The algorithm still explores only reachable states, and every state on the stack *Pending* has a transition from the state below it. Thus, when *DFS* is called with input s such that s satisfies the property φ, the stack contains an execution starting in an initial state and ending in the state s. Thus, when the algorithm returns a sequence of states, its output is indeed a witness to the reachability of φ. Whenever *DFS* is called with input s, it calls $\mathit{DFS}(t)$, for every state t such that there is a transition from s to t. As a result, unless a state satisfying φ is encountered, the algorithm will not terminate without exploring all the reachable states. Thus, if it stops with the answer 0, the property φ is not reachable. The claim 3 in theorem 3.2 about termination does not hold, however, for the modified algorithm. For example, suppose s_0 is an initial state (suppose *FirstInitState* returns s_0), does not satisfy the property φ, and has a transition to itself (suppose $\mathit{FirstSuccState}(s_0)$ returns s_0). Then the algorithm will keep looping calling $\mathit{DFS}(s_0)$ indefinitely. ■

Solution 3.12: The breadth-first-search algorithm is shown below. It maintains a set *Reach* that stores states that have been found to be reachable. The states to be explored are stored in the queue variable *Pending*. Note that a queue supports the operation `Enqueue` that adds an element at the end of the queue, and the operation `Dequeue` that removes and returns the first element in the queue. The algorithm examines all the initial states, and whenever it finds a "new" state, it adds it to the set *Reach*, and enqueues it for exploration. Then, as long as there are states to be explored, it dequeues a state from the queue *Pending*, and examines all its successors.

```
set(state) Reach := EmptySet;
queue(state) Pending := EmptyQueue;
foreach s in InitStates(T) {
  if Contains(Reach, s) = 0 then {
    if Satisfies(s, φ) = 1 then return 1;
    Insert(s, Reach);
    Enqueue(s, Pending);
    }
  };
while IsEmpty(Pending) = 0 {
  s := Dequeue(Pending);
  foreach t in SuccStates(s, T) {
    if Contains(Reach, t) = 0 then {
      if Satisfies(t, φ) = 1 then return 1;
      Insert(t, Reach);
      Enqueue(t, Pending);
      }
```

```
    }
  };
  return 0.
```

If the algorithm returns 1, the property φ is guaranteed to be reachable in the transition system T. If the algorithm returns 0, the property φ is not reachable in the transition system T. If the transition system has finitely many reachable states, then the algorithm is guaranteed to terminate. Note that the algorithm explores transitions out of each state only once, and thus if the transition system has n reachable states and m total transitions out of these reachable states, then the algorithm runs in time $O(m+n)$. ■

Solution 3.13: The transition system $\mathtt{Mult}(m,n)$ has state variables x of type `nat`, y of type `nat`, and *mode* of the enumerated type $\{\mathtt{loop},\mathtt{stop}\}$. The initialization is given by the formula

$$(\mathit{mode} = \mathtt{loop}) \ \wedge \ (x = m) \ \wedge \ (y = 0).$$

The transition formula is given as:

$$\begin{array}{ll} & [(\mathit{mode} = \mathtt{loop}) \ \wedge \ (x > 0) \ \wedge \ (x' = x - 1) \ \wedge \ (y' = y + n) \ \wedge \ (\mathit{mode}' = \mathtt{loop})] \\ \vee & [(\mathit{mode} = \mathtt{loop}) \ \wedge \ (x = 0) \ \wedge \ (x' = x) \ \wedge \ (y' = y) \ \wedge \ (\mathit{mode}' = \mathtt{stop})] \end{array}$$

■

Solution 3.14: The initialization formula is $(\mathit{mode} = \mathtt{off}) \ \wedge \ (x = 0)$. The reaction formula is given by

$$\begin{array}{ll} & [(\mathit{mode} = \mathtt{off}) \ \wedge \ (\mathit{press} = 0) \ \wedge \ (x' = x) \ \wedge \ (\mathit{mode}' = \mathtt{off})] \\ \vee & [(\mathit{mode} = \mathtt{off}) \ \wedge \ (\mathit{press} = 1) \ \wedge \ (x' = x) \ \wedge \ (\mathit{mode}' = \mathtt{on})] \\ \vee & [(\mathit{mode} = \mathtt{on}) \ \wedge \ (\mathit{press} = 0) \ \wedge \ (x < 10) \ \wedge \ (x' = x + 1) \ \wedge \ (\mathit{mode}' = \mathtt{on})] \\ \vee & [(\mathit{mode} = \mathtt{on}) \ \wedge \ (\mathit{press} = 1 \ \vee \ x \geq 10) \ \wedge \ (x' = 0) \ \wedge \ (\mathit{mode}' = \mathtt{off})]. \end{array}$$

To obtain the transition formula, we existentially quantify the input variable *press* from the above formula. This leads to

$$\begin{array}{ll} & [(\mathit{mode} = \mathtt{off}) \ \wedge \ (x' = x)] \\ \vee & [(\mathit{mode} = \mathtt{on}) \ \wedge \ (x < 10) \ \wedge \ (x' = x + 1) \ \wedge \ (\mathit{mode}' = \mathtt{on})] \\ \vee & [(\mathit{mode} = \mathtt{on}) \ \wedge \ (x \geq 10) \ \wedge \ (x' = 0) \ \wedge \ (\mathit{mode}' = \mathtt{off})]. \end{array}$$

■

Solution 3.15: The transition formula captures all possible relationships between the new and the old values of the state variables, but does not include the input/output variables. Since the interaction between two components is influenced by the input/output variables, if we first compute the transition formulas φ_T^1 and φ_T^2 for two components C_1 and C_2 separately, and take their conjunction, the result will not reflect the transitions of the composed component correctly. As a concrete example, suppose the component C_1 has a state variable x of type `nat`, and an output variable y of type `bool`. In each transition, either the output is 0 and x stays unchanged, or the output is 1 and x is incremented by 1. Thus, the reaction formula φ_R^1 is $(x' = x \ \wedge\ y = 0) \ \vee\ (x' = x+1 \ \wedge\ y = 1)$, and the transition formula φ_T^1 is $(x' = x) \ \vee\ (x' = x+1)$. The component C_2 has a state variable z of type `nat`, and the input variable y. If the input y is 0, z stays unchanged, and if the input y is 1, z is incremented by 1. Thus, the reaction formula φ_R^2 is $(z' = z \ \wedge\ y = 0) \ \vee\ (z' = z+1 \ \wedge\ y = 1)$, and the transition formula φ_T^2 is $(z' = z) \vee (z' = z+1)$. When we compose C_1 and C_2, in each round either y is 0 and both x and z stay unchanged, or y is 1 and both x and z are incremented. However, the conjunction $\varphi_T^1 \wedge \varphi_T^2$ is the formula

$$[(x' = x) \ \vee\ (x' = x+1)] \ \wedge\ [(z' = z) \ \vee\ (z' = z+1)].$$

This does not capture the transitions of $C_1 \| C_2$ accurately as it allows the possibility of x staying unchanged while z gets incremented. ■

Solution 3.16: Conjunction of the given region A and the transition formula gives

$$(x' = x+1) \ \wedge\ (y' = x) \ \wedge\ (0 \leq x \leq 4) \ \wedge\ (y \leq 7).$$

The existential quantification of the unprimed variables leads to $(x' = y'+1) \ \wedge$ $(0 \leq y' \leq 4)$. Renaming the primed variables to their unprimed counterparts gives the desired post-image: $(x = y+1) \ \wedge\ (0 \leq y \leq 4)$. ■

Solution 3.17: The transition formula is given by:

$$[(x < y) \wedge (x' = x+y) \wedge (y' = y)] \ \vee\ [(x \geq y) \wedge (x' = x) \wedge (y' = y+1)].$$

To obtain the post-image of the region $(0 \leq x \leq 5)$, we first conjoin this formula with the transition formula. Existentially quantifying y gives

$$[(x < y') \wedge (x' = x+y') \wedge (0 \leq x \leq 5)] \ \vee\ [(x \geq y'-1) \wedge (x' = x) \wedge (0 \leq x \leq 5)].$$

Existentially quantifying x from this formula gives

$$[(x' < 2y') \wedge (0 \leq x' - y' \leq 5)] \ \vee\ [(x' \geq y'-1) \wedge (0 \leq x' \leq 5)].$$

Renaming the primed variables to the corresponding unprimed ones gives the desired result:

$$[(x < 2y) \wedge (0 \leq x - y \leq 5)] \ \vee\ [(x \geq y-1) \wedge (0 \leq x \leq 5)].$$

■

Solution 3.18: Given a region A, to compute its pre-image, we first rename the unprimed variables to primed variables, and then intersect it with the transition region *Trans* over $S \cup S'$ to obtain all the transitions that lead to the states in

A. Then, we project the result onto the set S of unprimed state variables by existentially quantifying the variables in S'. Thus the pre-image operator `Pre` is defined as:

$$\texttt{Pre}(A, \mathit{Trans}) \;=\; \texttt{Exists}(\texttt{Conj}(\texttt{Rename}(A, S, S'), \mathit{Trans}), S').$$

The backward-search algorithm is symmetric to the algorithm in Figure 3.18. The region *Reach* contains all the states from which a state satisfying the property φ has been discovered to be reachable. It initially contains the states that satisfy φ, and in each iteration, states from which there is a transition to a state already in *Reach*, are added using the pre-image computation. At any step, if the region *Reach* contains an initial state, the algorithm has discovered an execution from an initial state to a state satisfying φ, and can terminate.

Input: A transition system T given by a region *Init* for initial states and a region *Trans* for transitions, and a property φ.
Output: If φ is reachable in T, return 1, else return 0.

```
reg Reach := φ;
reg New := φ;
while IsEmpty(New) = 0 do {
  if IsEmpty(Conj(New, Init)) = 0 then return 1;
  New := Diff(Pre(New, Trans), Reach);
  Reach := Disj(Reach, New);
  };
return 0.
```

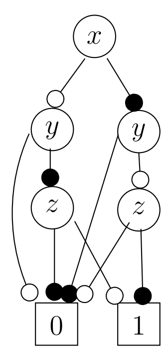
■

Solution 3.19 : During the execution of the algorithm of figure 3.18, let New_1 be the value of the region *New* at the beginning of the first iteration of the while-loop, let New_2 be its value at the beginning of the second iteration of the loop, and so on. Suppose during the ith iteration of the while-loop, the algorithm discovers a state satisfying the property φ, that is, the intersection of the regions New_i and φ is non-empty. To return a witness execution, we first choose a specific state, say, s_i that belongs to both New_i and φ. Then, we need to find a state that belongs to the region New_{i-1} and has a transition to state s_i. This can be achieved by computing the set of predecessors of the state s_i, intersect this set with the region New_{i-1}, and select a state s_{i-1} in this intersection (note that this intersection is guaranteed to be a non-empty region since the region New_i was obtained by applying the post-image operation to the region New_{i-1}). We can repeat this process till an initial state in the region New_1 is chosen. The modified algorithm is shown below. The sequence of regions $\mathit{New}_1, \mathit{New}_2, \ldots$ is stored using the stack *Frontiers*. The algorithm uses one new operation on regions: given a non-empty region A, `SelectState`(A) returns one state belonging to A (the specific choice does not matter). The pre-image computation operation `Pre` is the same as the one described in exercise 3.18, except it is now applied to a region containing a single state.

```
reg Reach := Init;
reg New := Init;
stack(reg) Frontiers := EmptyStack;
while IsEmpty(New) = 0 do {
  if IsEmpty(Conj(New, φ)) = 0 then {
    stack(state) Exec := EmptyStack;
    state s := SelectState(Conj(New, φ));
    Push(s, Exec);
    while IsEmpty(Frontiers) = 0 {
      s := SelectState(Conj(Pop(Frontiers), Pre(s, Trans)));
      Push(s, Exec);
      };
    return Exec
    };
  Push(New, Frontiers);
  New := Diff(Post(New, Trans), Reach);
  Reach := Disj(Reach, New);
  };
return 0.
```

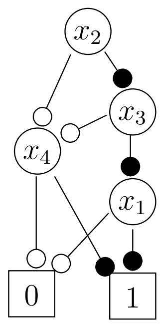
■

Solution 3.20: The ROBDD is shown below:



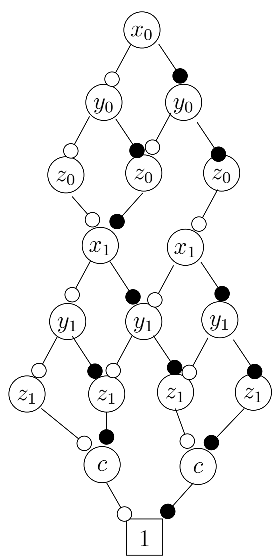
■

Solution 3.21: The ROBDD for the variable ordering $x_2 < x_3 < x_4 < x_1$ is shown below:



For each variable x_i, there is only one vertex labeled with x_i. Given that the Boolean function captured by the formula does depend on all the four variables, the ROBDD could not possibly have fewer than 4 internal vertices no matter which variable ordering we pick. Thus, this is the smallest possible ROBDD. ■

Solution 3.22: The natural variable ordering is $x_0 < y_0 < z_0 < x_1 < y_1 < z_1 < c$. The corresponding ROBDD is shown below:



Note that to simplify the drawing, we have omitted the terminal node 0, and the edges that lead to it (for example, the right-edge of the left node labeled with c and the left-edge of the right node labeled with c). ■

Solution 3.23: The algorithm for existential quantification uses the algorithm for computing disjunction of ROBDDs: given two ROBDDs B and B', the routine $\mathit{Disj}(B, B')$ returns the ROBDD representation of the function $f(B) \vee f(B')$. The implementation of $\mathit{Disj}(B, B')$ follows the same outline as the algorithm for $\mathit{Conj}(B, B')$ in figure 3.26.

Given an ROBDD B and a set X of identifiers of variables to be existentially quantified, the routine below computes the ROBDD representation for the function $\exists X. f(B)$. As in case of the algorithm for computing the conjunction, to avoid recomputation, it maintains a table *Done*, indexed by the input arguments for the routine *Exists*.

```
bdd Exists(B, X)
  if (B = 0 ∨ B = 1) return B;
  if Done[(B, X)] ≠⊥ then return Done[(B, X)];
  bddnode u := BDDPool[B];
  nat j := Label(u); bdd B_0 := Left(u); bdd B_1 := Right(u);
  if (j ∉ X) then B' := AddVertex(j, Exists(B_0, X), Exists(B_1, X))
  else B' := Disj(Exists(B_0, X), Exists(B_1, X));
  Done[(B, X)] := B';
  return(B').
```

The algorithm relies on the following observation. Suppose a function f equals $(\neg x \wedge f_0) \vee (x \wedge f_1)$. If $x \notin X$ then $\exists X. f$ equals $(\neg x \wedge \exists X. f_0) \vee (x \wedge \exists X. f_1)$, and if $x \in X$ then $\exists X. f$ equals $\exists X. f_0 \ \vee \ \exists X. f_1$. ■

Solution 3.24: Consider the boolean expression x (that is, the function that evaluates to 1 when x is 1 and 0 otherwise). The ROBDD for this function has one internal vertex and 2 terminal vertices. However, this function can be represented by a COBDD with just one internal vertex and one terminal vertex: the terminal vertex is labeled 0, and the internal vertex has label x, both of its edges go to terminal 0, and the right-edge is labeled negative.

It is possible to define reduction rules for COBDDs that guarantee canonicity. First, both the left-edge and right-edge of an internal vertex can lead to the same vertex only if the right-edge is labeled negative. Second, isomorphic vertices are not allowed (that is, if u and v are internal vertices, then either their labels should differ, or their left-edges should lead to distinct vertices, or their right-edges should lead to distinct vertices, or their right-edges should differ in positive/negative labels). Third, there are no two vertices that represent functions that are complements of one another. Finally, there is only one terminal vertex, labeled 0. By these rules, COBDD of Figure 3.27 is not reduced: to reduce it, we eliminate the terminal vertex 1, and the right-edge of the vertex labeled z is directed to the terminal vertex 0 with a negative label. Under these reduction rules, for every Boolean function f, for a given variable ordering, exactly one of the function f and $\neg f$ can be represented as a reduced COBDD, and that representation is unique. The ROBDD algorithms (such as the one for conjunction) can be modified for this new representation to maintain canonicity. ■

4 Asynchronous Model

Solution 4.1: The input variables of the asynchronous process `AsyncAdd` are x_1 and x_2 of type `nat`. Its output variable is y of type `nat`. It maintains two queues as state variables with the declaration given by

$$\texttt{queue}(\texttt{nat})\ z_1 := \texttt{null};\ z_2 := \texttt{null}.$$

The input task A_i^1 specified by

$$\neg\,\texttt{Full}(z_1)\ \rightarrow\ \texttt{Enqueue}(x_1, z_1)$$

stores the messages arriving on the input channel x_1 in the queue z_1. Symmetrically, the input task A_i^2 processes messages arriving on the input channel x_2, and is specified by

$$\neg\,\texttt{Full}(z_2)\ \rightarrow\ \texttt{Enqueue}(x_2, z_2).$$

The output task A_o is enabled when both the queues are nonempty and if so, removes a message from each of the two queues and transmits their sum on the output channel y:

$$\neg\,\texttt{Empty}(z_1)\ \wedge\ \neg\,\texttt{Empty}(z_2)\ \rightarrow\ y := \texttt{Dequeue}(z_1) + \texttt{Dequeue}(z_2).$$

■

Solution 4.2: The asynchronous process `Split` has a single input variable *in* of type `msg`. Its output variables are out_1 and out_2 of type `msg`. It maintains a single queue as its state variable with the declaration given by

$$\texttt{queue}(\texttt{msg})\ x := \texttt{null}.$$

The input task A_i specified by

$$\neg\,\texttt{Full}(x)\ \rightarrow\ \texttt{Enqueue}(in, x)$$

stores the messages arriving on the input channel in the queue x. The output task A_o^1 is enabled when the queue x is nonempty and if so, removes a message from the queue and transmits it on the output channel out_1:

$$\neg\,\texttt{Empty}(x)\ \rightarrow\ out_1 := \texttt{Dequeue}(x).$$

The output task A_o^2 is symmetric, and transmits messages on the output channel out_2:

$$\neg\,\texttt{Empty}(x)\ \rightarrow\ out_2 := \texttt{Dequeue}(x).$$

Note that a message stored in the queue x is transmitted on only one of the output channels, and the choice is nondeterministic. ■

Solution 4.3: The process `AsyncAnd` maintains the following state variables:

$$\texttt{bool}\ x_1 := 0;\ x_2 := 0;\ x := 0;\ \{\texttt{stable}, \texttt{unstable}, \texttt{hazard}\}\ mode := \texttt{stable}.$$

The variables x_1 and x_2 are used to remember the most recent values of the input variables in_1 and in_2, respectively; the variable x corresponds to the value of the output; and the mode variable indicates whether the gate is stable, or unstable, or has encountered a hazard.

Since the state variable x is intended to correspond to the current output, the output task is simply given by $out\,!\,x$.

The input task responsible for processing the input in_1 is specified by the update code:

$$\begin{array}{l} x_1 := in_1; \\ \texttt{if}\ (mode = \texttt{stable})\ \wedge\ (x \neq x_1 \wedge x_2)\ \texttt{then mode} := \texttt{unstable} \\ \texttt{else if}\ (mode = \texttt{unstable})\ \wedge\ (x = x_1 \wedge x_2)\ \texttt{then}\ mode := \texttt{hazard}. \end{array}$$

The logic is analogous to that of the asynchronous process `AsyncNot`. In the mode `stable`, if the newly received input warrants a change in the output, that is, when the value of x differs from the desired conjunction $x_1 \wedge x_2$, the mode switches to `unstable` indicating a pending change in the output. In the mode `unstable`, if the newly received input warrants another change in the output, the mode switches to `hazard` indicating unpredictable output. The input task responsible for processing the input in_2 is symmetric.

The internal task is responsible for the changes in the value of the state variable x. In the mode `unstable`, a change in the output value is pending, and the internal task can flip the value of x switching the mode to `stable`, while in the mode `hazard`, the output can change to an arbitrary value. Thus, the internal task is specified by the update code:

$$\begin{array}{l} \texttt{if}\ (mode = \texttt{unstable})\ \texttt{then}\ \{x := \neg\, x;\ \texttt{mode} := \texttt{stable}\} \\ \texttt{else if}\ (mode = \texttt{hazard})\ \texttt{then}\ x := \texttt{choose}\ \{0, 1\}. \end{array}$$

■

Solution 4.4: The input channels are in_1, in_2, and in_3, all of type `msg`. The output channel is out. When composing the two instances of `Merge`, we need to make sure that the state variables have distinct names. The state variables of the composite process and their initialization is specified by

$$\texttt{queue(msg)}\ x_1 := \texttt{null};\ x_2 := \texttt{null};\ y_1 := \texttt{null};\ y_2 := \texttt{null}.$$

The composite process has three input tasks corresponding to its three input channels specified by:

$$\begin{array}{lcl} A_i^1 & : & \neg\,\texttt{Full}(x_1)\ \rightarrow\ \texttt{Enqueue}(in_1, x_1) \\ A_i^2 & : & \neg\,\texttt{Full}(x_2)\ \rightarrow\ \texttt{Enqueue}(in_2, x_2) \\ A_i^3 & : & \neg\,\texttt{Full}(y_2)\ \rightarrow\ \texttt{Enqueue}(in_3, y_2) \end{array}$$

The composition has two internal tasks, each of which is obtained by synchronizing an output of the first instance on the channel *temp* with a corresponding input processing by the second:

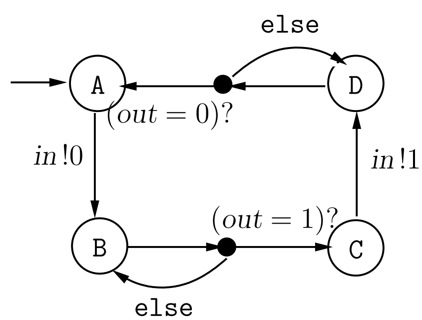
$$\begin{aligned}&A^1 : \neg\,\texttt{Empty}(x_1)\ \wedge\ \neg\,\texttt{Full}(y_1)\ \rightarrow\\ &\quad\{\ \texttt{local msg}\ temp := \texttt{Dequeue}(x_1);\ \texttt{Enqueue}(temp, y_1)\ \}\\ &A^2 : \neg\,\texttt{Empty}(x_2)\ \wedge\ \neg\,\texttt{Full}(y_1)\ \rightarrow\\ &\quad\{\ \texttt{local msg}\ temp := \texttt{Dequeue}(x_2);\ \texttt{Enqueue}(temp, y_1)\ \}\end{aligned}$$

Finally, the composite process has two output tasks that remove messages from the queues y_1 and y_2 in order to transmit them on the output channel *out*:

$$\begin{aligned}&A_o^1 : \neg\,\texttt{Empty}(y_1)\ \rightarrow\ out := \texttt{Dequeue}(y_1)\\ &A_o^2 : \neg\,\texttt{Empty}(y_2)\ \rightarrow\ out := \texttt{Dequeue}(y_2)\end{aligned}$$

The sequence of values output by the composite process represents a merge of the sequences of input values received on the three input channels. The relative order of values received on each of the input channels is preserved in the output sequence, but the three input sequences can be interleaved in any nondeterministic order. ■

Solution 4.5: The process `AsyncNotEnv` is shown as an extended state machine, with output *in* and input *out*, is shown below:



For the process obtained by composing `AsyncNot` and `AsyncNotEnv`, the state variables are `AsyncNot`.*mode*, *x*, and `AsyncNotEnv`.*mode*. It is easy to check that for the underlying transition system, the only reachable states are:

$$\{(\texttt{stable}, 0, \texttt{A}), (\texttt{unstable}, 0, \texttt{B}), (\texttt{stable}, 1, \texttt{B}), (\texttt{stable}, 1, \texttt{C}), (\texttt{unstable}, 1, \texttt{D}), (\texttt{stable}, 0, \texttt{D})\}$$

It follows that ($\texttt{AsyncNot}.mode \neq \texttt{hazard}$) is an invariant of the composite process. ■

Solution 4.6: A state is represented by listing the values of the variables P_1.`mode`, P_2.`mode`, *turn*, $flag_1$, and $flag_2$ in that order. We use I, T1, T2, T3, and C as abbreviations for the values `Idle`, `Try1`, `Try2`, `Try3`, and `Crit`, respectively. We use ? to denote the value of the variable *turn*, when a state is reachable with both $turn = 1$ and $turn = 2$. The transition system has two initial states $[I, I, ?, 0, 0]$. Besides these two initial states, 28 more states are reachable: $[I, T1, ?, 0, 1]$, $[T1, I, ?, 1, 0]$, $[T1, T1, ?, 1, 1]$, $[T2, I, 1, 1, 0]$, $[T2, T1, 1, 1, 1]$, $[I, T2, 2, 0, 1]$, $[T1, T2, 2, 1, 1]$, $[T2, T2, ?, 1, 1]$, $[I, C, 2, 0, 1]$, $[C, I, 1, 1, 0]$, $[T1, T3, 2, 1, 1]$, $[T1, C, 2, 1, 1]$, $[T3, T1, 1, 1, 1]$, $[C, T1, 1, 1, 1]$, $[T2, T3, ?, 1, 1]$, $[T3, T2, ?, 1, 1]$, $[T3, T3, ?, 1, 1]$, $[T2, C, 1, 1, 1]$, $[T3, C, 1, 1, 1]$, $[C, T2, 2, 1, 1]$, and $[C, T3, 2, 1, 1]$. ■

Solution 4.7: The modified protocol does satisfy the mutual exclusion requirement. First observe that the process P_1 sets flag_1 to 1 when it leaves the mode `Idle` and to 0 when it returns to the mode `Idle`, and the process P_2 does not write to flag_1. Thus, the variable flag_1 is 0 exactly when the mode of P_1 is `Idle`. Symmetrically, the variable flag_2 is 0 exactly when the mode of P_2 is `Idle`. Consider a state where one of the processes, say P_1, is already in the critical section and the other process, P_2, is attempting to switch to its critical section. Then, since P_1's mode is `Crit`, we must have $\mathit{flag}_1 = 1$, and this ensures that the process P_2 cannot find the condition ($\mathit{flag}_1 = 0$) needed to enter the critical section to be satisfied. As a result, a state with both mode variables equal to `Crit` is unreachable.

The protocol however deadlocks. Consider an execution in which the process P_1 first switches from `Idle` to `Try` setting flag_1 to 1, and then the process P_2 switches from `Idle` to `Try` setting flag_2 to 1. In the resulting state, both processes are blocked from entering their critical sections, and the state stays unchanged no matter which process we choose to execute. Thus, this is not a satisfactory solution to the mutual exclusion problem. ■

Solution 4.8: The modified protocol does not satisfy mutual exclusion. The following execution is a counterexample.

1. The process P_1 sets *turn* to 1 and switches to the mode `Try1`.
2. The process P_2 sets *turn* to 2 and switches to the mode `Try1`.
3. The process P_2 sets flag_2 to 1 and switches to the mode `Try2`.
4. The process P_2 checks the value of flag_1, finds it to be 0, and proceeds to the critical section.
5. The process P_1 now executes setting flag_1 to 1 and switches to the mode `Try2`.
6. The process P_1 checks the value of flag_2, finds it to be 1, and hence proceeds to the mode `Try3`.
7. The process P_1 reads *turn*, finds it to be 2, and hence, continues to the critical section causing a violation of the mutual exclusion requirement.

■

Solution 4.9 : All values are reachable: for every positive number n, there is an execution such that the value of the shared register x equals n at the end of the execution.

In the desired execution, we interleave the steps by the two processes so that (1) the execution starts with the two read operations by the process P_1 and (2) whenever the process P_1 writes a value to the shared register, the next two steps correspond its own read operations. This ensures that the sequence of read/write operations executed by the process P_1 is not influenced by the operations of the process P_2: the sequence of operations P_1 executes is $u_1 := 2^i;\ v_1 := 2^i;\ x := 2^{i+1}$, for $i = 0, 1, 2, \ldots$.

The steps of the process P_2 are interleaved with those of P_1 depending on the binary encoding of the number n. Suppose $n = 2^{i_1} + 2^{i_2} + \cdots + 2^{i_k}$, where $i_1 < i_2 < \cdots < i_k$. That is, the binary representation of n has 1s in k positions. Then, in the desired execution, the process P_2 executes its first read operation immediately after the process P_1 has executed the operation $v_1 := x$ i_1 times. This ensures that P_2 sets u_2 to the value 2^{i_1}. The process P_2 then executes its second read operation immediately after the process P_1 has executed i_2 times the operation $v_1 := x$. This ensures that P_2 sets v_2 to the value 2^{i_2}. It then immediately writes the sum $2^{i_1} + 2^{i_2}$ to the shared register x, and reads it back into the variable u_2. This pattern is repeated: the subsequent read by P_2 is scheduled after the process P_1 has executed the operation $v_1 := x$ i_3 times, ensuring that P_2 sets v_2 to the value 2^{i_3}; it then immediately writes the sum $2^{i_1} + 2^{i_2} + 2^{i_3}$ to the shared register x, and reads it back into the variable u_2. Repeating this pattern will result in P_2 writing the value n to x.

As an example, if $n = 11 = 1 + 2 + 8$, then the following sequence of operations leads to $x = 11$:

$P_1:\ u_1 := 1;\ v_1 := 1;$
$P_2:\ u_2 := 1;$
$P_1:\ x := 2;\ u_1 := 2;\ v_1 := 2;$
$P_2:\ v_2 := 2;\ x := 3;\ u_2 := 3;$
$P_1:\ x := 4;\ u_1 := 4;\ v_1 := 4;$
$P_1:\ x := 8;\ u_1 := 8;\ v_1 := 8;$
$P_2:\ v_2 := 8;\ x := 11.$

■

Solution 4.10 : Without any fairness assumption, the output task A_o^1 responsible for transmitting messages on the output channel out_1 may never get executed. Weak-fairness is not enough. Consider the execution in which the following two steps are repeatedly executed: (1) a message is received on the input channel and stored in the queue x; (2) it is then removed and transmitted on the output channel out_2 by the task A_o^2 making the queue empty. In this execution, infinitely many messages are received on the input channel, but no message is ever transmitted on the output channel out_1. The execution is weakly-fair to the task A_o^1 since it is not continuously enabled. Strong fairness assumption for the task A_o^1 will rule out this execution, and in fact, ensures that if the queue x is repeatedly nonempty (which is guaranteed to happen in an execution if infinitely many messages are received on the input channel), then the task A_o^1 is executed infinitely often transmitting infinitely many messages on the output channel out_1. Analogously, we also need a strong fairness assumption for the task A_o^2. ■

Solution 4.11: One way to achieve reordering is by introducing an additional internal task:

$$A^4 : \neg\,\texttt{Empty}(x) \rightarrow \{\ \texttt{local msg}\ v := \texttt{Dequeue}(x);\ \texttt{Enqueue}(v, x)\ \}.$$

Executing the task A^4 moves the message at the front of the queue to the end. When such a rearrangement is possible, an input message can be inserted in the current queue at any possible position: if the current queue contains k messages $v_1, v_2, \ldots v_k$, then executing the task A^4 i times, followed by the execution of the input task A_i to receive an input message v, followed by $(k-i+1)$ executions of the task A^4, will result in the queue $v_1, v_2, \ldots v_i, v, v_{i+1}, \ldots v_k$. Thus, all possible reorderings are captured by the introduction of the task A^4.

Following the discussion of fairness assumptions for the process `UnrelFIFO`, the task A^4 of the process `VeryUnrelFIFO` captures a potential anomaly and a protocol using this process should meet its correctness requirements no matter whether the task A^4 gets executed or not. Thus, no fairness should be assumed for this new task. ■

Solution 4.12: For the process P_1, the mode-switch from `Idle` to `Try` indicates that the process now wants to enter the critical section. We don't require any fairness assumption for this task since the protocol should work correctly even if this process never leaves the mode `Idle`. The mode-switch out of the mode `Try` reads the variable $flag_2$. This task is enabled as long as the process P_1 is in the mode `Try`, and we require weak fairness assumption for it to ensure that the process will eventually read the variable $flag_2$ (and decide to either proceed to the critical section or return to the mode `Try`). Finally, for the mode-switch from `Crit` to `Idle`, we also require weak fairness to ensure that the process will eventually relinquish the critical section. The fairness assumptions for the process P_2 are analogous.

The deadlock scenario described in the solution to exercise 4.7 can still occur: an execution can lead to a state where both processes are in the mode `Try` with both flag variables set to 1, and executing any of the enabled tasks in this state does not result in a change of state. Thus, even with the fairness assumptions, the protocol does not satisfy the requirement that *if a process wants to enter the critical section, then it eventually will enter the critical section.* ■

Solution 4.13: 1. In absence of any fairness assumption for the task A_1, only the task A_2 may get executed repeatedly leaving the value of x unchanged.

Thus, it is not guaranteed that the value of x eventually exceeds 5. Assuming weak fairness for the task A_1 ensures that it gets executed repeatedly since it is enabled at every step. Thus, under the weak fairness assumption for the task A_1 it is guaranteed that the value of x eventually exceeds 5.

2. In absence of any fairness assumption for the task A_2, only the task A_1 may get executed repeatedly leaving the value of y unchanged. Thus, it is not guaranteed that the value of y eventually exceeds 5. Assuming weak fairness for the task A_2 ensures that it gets executed repeatedly since it is enabled at every step. However, in absence of any fairness assumptions for the task A_1, it may never get executed leaving the value of x unchanged at 0, and in such a case, repeated execution of the task A_2 leaves the value of y also unchanged. If we assume weak fairness for both the tasks, it is guaranteed that both get executed repeatedly (note: strong fairness is not required since both tasks are always enabled), and this ensures that the value of x eventually becomes nonzero, which in turn ensures that repeated execution of the task A_2 is guaranteed to increase its value beyond 5.

3. If we execute the two tasks A_1 and A_2 alternately, in the resulting execution the value of x is strictly less than the value of y at every step. Thus, it is not guaranteed that the values of x and y become equal at some step. In this specific execution, both tasks are repeatedly executed, and thus, even under fairness assumptions, the desired property does not hold. ■

Solution 4.14: After the first phase five processes continue: process 3 with *id* set to 25, process 8 with *id* set to 19, process 4 with *id* set to 14, process 21 with *id* set to 22, and process 1 with *id* set to 24. Of these, only process 8 with *id* set to 25 continues to the next phase, and is elected as the leader. ■

Solution 4.15: Part a. Suppose there are N processes with identifiers $\{1, 2, \ldots N\}$. Suppose they are arranged in an increasing order in a ring: $1, 2, \ldots N$. Then, each process other than process 1 has an identifier higher than its predecessor. After one phase, only process 1 proceeds to the second phase, and all the remaining processes decide to become followers.

Part b. In a ring of processes, if every process at an odd-numbered position has an identifier higher than the identifiers of all the processes in the even-numbered positions, then every process in the even position proceeds to the next phase with every process in the odd position deciding to become a follower. Suppose there are N processes with identifiers $\{1, 2, \ldots N\}$. We can construct the worst-case scenario in the following manner: starting with position 1, place processes with identifiers $1, 2, \ldots N/2$ in every alternate position; then starting with position 2, place processes with identifiers $N/2+1, N/2+2, \ldots 3N/4$ in every alternate unfilled position; then starting with position 4, place processes with identifiers $3N/4+1, 3N/4+2, \ldots 7N/8$ in every alternate unfilled position; and so on. In such a ring, in every phase, every active process at an odd (active) position has an identifier higher than the identifiers of all the processes in even (active) positions, and exactly half of the currently active processes proceed to the next phase. This implies worst-case performance with $\log N$ phases. For example, if $N = 16$, consider the ring with processes $1, 9, 2, 13, 3, 10, 4, 15, 5, 11, 6, 14, 7, 12, 8, 16$, and check that in each phase exactly half of the remaining processes decide to become followers. ■

Solution 4.16: Suppose that the communication link from the receiver back to the sender transfers messages reliably. Then, there is no need for the receiver to send acknowledgments repeatedly. Every time it receives a new message, it acknowledges using exactly one message. Since the link from the sender to receiver can lose messages, the sender needs to send a message repeatedly till it receives an acknowledgment. Furthermore, the message still needs to be tagged so that the receiver can distinguish between a fresh message and a duplicate copy of an old message. There is no need for the acknowledgment to carry any value though.

To implement this simplification, the channels x_2 and y_2 do not need to carry any value. For the sender process, the tasks A_i and A_1 remain unchanged, but we can simplify the update code of the task A_2 responsible for processing input events on the channel x_2 to:

$$tag := \neg tag;\ \texttt{Dequeue}(x).$$

The receiver process keeps an additional state variable z of type `bool`, initialized to 0, to ensure that exactly one acknowledgment is sent. The input task A_1 for processing messages on the channel y_1 is changed to

$$\texttt{if Second}(y_1) \neq tag \ \texttt{then} \ \{\ tag := \neg\, tag;\ \texttt{Enqueue}(\texttt{First}(y_1), y);\ z := 1\ \}.$$

The output task A_2 responsible for transmitting acknowledgments is changed to

$$(z = 1) \ \rightarrow \ \{\ y_2!;\ z := 0\ \}.$$

■

Solution 4.17: If we replace each instance of `UnrelFIFO` with `VeryUnrelFIFO`, but leave the sender and receiver processes unchanged, then the following sequence of steps is a counterexample to the correctness of the protocol.

1. The sender process receives a message m_1 on its input channel and enqueues it in the queue x.
2. The sender process transmits the message $(m_1, 1)$ on its output channel x_1 and this message is enqueued in the state variable of the unreliable link $\texttt{VeryUnrelFIFO}_1$.
3. The link $\texttt{VeryUnrelFIFO}_1$ transmits the message on its output channel y_1, but without removing it from its queue by executing the task A_o^3. This message $(m_1, 1)$ is received by the receiver process. Since its internal tag is 0, it considers this to be a fresh message, enqueues m_1 in its queue y, and flips its tag to 1.

4. The receiver process transmits the message m_1 on its output channel *out*.

5. The receiver process transmits its tag value 1 on its output channel y_2, which is received by the unreliable link $\texttt{VeryUnrelFIFO}_2$ and stored in its internal queue.

6. The link $\texttt{VeryUnrelFIFO}_2$ dequeues the message 1 and transmits it on the channel x_2. This acknowledgment is received by the sender process. Since the tags match, it flips its tag to 0, and removes m_1 from its queue x.

7. The sender process receives a message m_2 on its input channel and enqueues it in the queue x.

8. The sender process transmits the message $(m_2, 0)$ on its output channel x_1 and this message is enqueued in the state variable of the unreliable link $\texttt{VeryUnrelFIFO}_1$.

9. The link $\texttt{VeryUnrelFIFO}_1$ executes the internal task A^4 that reorders the two messages in its queue, resulting in the state $[(m_2, 0), (m_1, 1)]$.

10. The link $\texttt{VeryUnrelFIFO}_1$ dequeues the message $(m_2, 0)$ and transmits it on its output channel y_1. This message is received by the receiver process. Since its internal tag is 1, it considers this to be a fresh message, enqueues m_2 in its queue y, and flips its tag to 0.

11. The receiver process transmits the message m_2 on its output channel *out*.

12. The link $\texttt{VeryUnrelFIFO}_1$ dequeues the message $(m_1, 1)$ and transmits it on its output channel y_1. This message is received by the receiver process. Since its internal tag is 0, it considers this also to be a fresh message, enqueues m_1 in its queue y, and flips its tag to 1.

13. The receiver process now transmits the message m_1 on its output channel *out*.

In this execution the sequence of messages received on the input channel *in* is m_1, m_2, while the sequence of messages transmitted on the output channel *out* is m_1, m_2, m_1. This is a violation of the desired correctness requirement.

In the revised version of the protocol, we change the type of the tag variable to `nat`. Initially the tag value is 1 for the sender and 0 for the receiver. Whenever the receiver finds the tag of an incoming message to be one higher than its internal state variable, it views it as a fresh message and increments its internal tag. As an acknowledgment, it sends this tag value to the sender. When the sender receives an acknowledgment that equals its internal tag, it concludes that the last message was successfully delivered, increments its tag, and starts transmitting a new message. Even though both the sender and the receiver transmit tag values in a nondecreasing order, due to reordering of messages in the unreliable link, they may receive tag values in an arbitrary order. However, any copy of the ith message received on the input channel is tagged with the value i, and the corresponding acknowledgment is i. This uniquely identifies each message, and the resulting protocol ensures that the sequence of messages transmitted on the output channel *out* equals the sequence of messages received on the input channel *in*.

The precise changes to the sender and the receiver process are the following: since tag is now a number, in all declarations, the type `bool` is replaced by `nat`; in the description of the task A_2 of the process P_s, the update of *tag* is changed to $tag := tag + 1$; and in the description of the task A_1 of the process P_r, the test is changed to $\texttt{Second}(y_1) = tag + 1$ and the update of *tag* is changed to $tag := tag + 1$. ■

Solution 4.18: Consider the case when the two processes write to different registers. That is, in the state s, the process P_1 writes some value m_1 to the register x leading to the 0-committed state s_1, and the process P_2 writes some value m_2 to a different register y leading to the 1-committed state s_2. Note that in the state s_1, even though the value of the register x is different from its value in the state s, the internal state of the process P_2 is the same in both the states s and s_1. Thus, in the state s_1, the process P_2 can write the same value m_2 to the register y leading to the state t. By a symmetric argument, in the state s_2, the process P_1 is ready to write the same value m_1 to the register x as in state s. Furthermore, the state resulting from executing the process P_1 in the state s_2 is the same as the state t. That is, when the processes are about to write to different registers in the state s, whether we execute P_1 first and then P_2, or P_2 first and then P_1, the resulting state is the same state t. Since t is a successor of the 0-committed state s_1, t must be 0-committed, but t is also a successor of the 1-committed state s_2, so t must be 1-committed. This is not possible leading to a contradiction. ■

Solution 4.19: Validity: The protocol satisfies this requirement. Suppose both processes start with the initial value 0. Since a process writes its initial preference to the register x at the first step, the value of the register changes from *null* to 0 and stays 0 (since both processes write the same value to x in this case). Since a process decides either its own initial value or the value it reads from x at the end, it must decide on 0. Analogously, when both processes start with the initial value 1, both decide on the value 1.

Agreement: The protocol does not satisfy this requirement. Suppose process P_1 has initial value 0 and process P_2 has initial value 1. Suppose process P_1 executes all its steps before process P_2 gets to execute any of its steps. In such a case, the test-and-set operation executed by process P_1 returns 1, and so it decides on its own initial value, namely, 0. At this point, process P_2 begins and executes all its steps. In this case, the first step of P_2 writes the value 1 to the shared register x, the test-and-set operation returns 0, and thus, process P_2 decides on the value it reads from the register x, namely, 1. Thus, the two processes decide on conflicting values.

Wait-freedom: The protocol satisfies this requirement. Each process executes only three steps and makes a decision without ever waiting for the other process. ■

Solution 4.20 : Suppose there are three processes P_1, P_2, and P_3. The natural generalization of the protocol of figure 4.27 has three atomic registers x_1, x_2, and x_3, and a test-and-set register y. Each process P_i has its initial value in the variable $pref_i$. Its first step is to write this initial preference to the atomic register x_i. The second step executes a test-and-set operation on the register y. If this operation returns 0, then this process P_i is the first one to execute the test-and-set operation, and it can decide on its own preference $pref_i$. If this operation returns 1, then the process P_i can conclude that some other process has successfully executed a test-and-set operation earlier. However, unlike the two-process case, now P_i does not know the identity of the winning process. Reading the other two atomic registers x_j, for $j \neq i$, is not useful for resolving this ambiguity since they may contain two different values.

As another strategy for solving three-process consensus, let us try to use two rounds of competition using two separate test-and-set registers. The two processes P_1 and P_2 can use shared atomic registers x_1 and x_2 and a test-and-set register y_1 to reach agreement among them, and the winner can compete with the process P_3 using shared atomic registers x_{12} and x_3 and a test-and-set register y_2 before deciding. In particular, the process P_1 first writes its own preference to the atomic register x_1. It then executes a test-and-set operation on the register y_1. If this returns 0, it continues and writes its preference to the atomic register register x_{12}. It then executes a test-and-set operation on the register y_2. If this second test-and-set operation also returns 0, it decides on its own initial preference. If the second test-and-set operation returns 1, then it can decide on the preference of the process P_3 by reading the atomic register x_3. The problem case, however, is when the first test-and-set operation on the register y_1 fails, that is, returns 1. In this case, the process P_1 knows that it has lost to the process P_2, but it cannot simply decide on the initial preference of P_2 (which can be read from the atomic register x_2) since in this case, P_2 is competing with P_3 in the second round, and the winner among them cannot be determined simply by reading any of the registers. ■

Solution 4.21 : We can design a consensus protocol using a single shared `StickyBit` register x that works for arbitrarily many processes. Each process P executes the following two steps: (1) write its own preference to the `StickyBit` register x, and (2) read the value of the `StickyBit` register x, and decide on the read value. Whichever process executes its first step first, the value of x will be changed to the initial preference of that process, and furthermore, this value stays unchanged even when the other processes attempt to write it again. As a result, all processes read the same value in the second step. ■

Solution 4.22 : Let us first assume that the initial position of the switch is up and known to all the prisoners. When the prisoners initially get together, they designate one of them as the leader. The leader maintains a count, initialized to 0, and executes the following protocol upon entering the room: if the switch is up, do nothing, else flick the switch from down to up and increment the count; if the count equals $N-1$, declare that "every prisoner has visited this room at least once." Each non-leader executes the following protocol upon entering the room: if the switch is up and I have not flicked the switch in the past, flick the switch from up to down, else do nothing (to implement this, such a prisoner needs to maintain a boolean-valued state, which is initialized to 0 and updated to 1 when (s)he flicks the switch from up to down for the first time).

The switch is initially up. As long as the guard keeps bringing the leader to the room, there is no change in status. Eventually, the guard is guaranteed to bring a non-leader to the room. This person will flick the switch to down, and subsequently plays no role in the protocol. Following this, as long as the guard keeps bringing non-leaders to the room, there is no change in the status. The fairness guarantee ensures that eventually the leader will be brought to the room, will flick the switch back to up and update the count to 1. This cycle now repeats. Eventually the guard will bring someone who is not a leader and who has not yet flicked the switch from up to down (till then there is no change in status), and this prisoner will flick the switch to down, and in future will play no active role. Following this, eventually the leader is brought to the room (and since only the leader changes the switch from down to up, no change will occur till then), flicks the switch back to up and increments the count. Thus, every non-leader flicks the switch from up to down exactly once, and for each such event, the leader flips it back to up. When the leader's count reaches $N-1$, the leader knows that all the $N-1$ non-leaders have flicked the switch (and thus have been brought to the room at least once), and makes the desired conclusion leading to freedom.

Now suppose the initial state of the switch is not known. The protocol above does not work as is: when the leader enters the room for the first time and finds the switch is down, (s)he cannot distinguish between the case when the switch was initially down and the case when the switch was initially up and was flicked down by a non-leader who was brought to the room before the leader. As a result, when the leader's count reaches $N-1$, (s)he cannot be sure whether the number of non-leaders who have flicked the switch from up to down is $N-1$ (this will be the case if the switch was initially up) or $N-2$ (this will be the case if the switch was initially down). The protocol described above is modified as follows to get a correct solution: each non-leader flicks the switch from up to down *twice* instead of once, and the leader makes the announcement when the count reaches $2N-2$. When the count reaches $2N-2$, the leader is not sure whether the number of times the switch was flicked from up to down is $2N-2$ (this will be the case if the switch was initially up) or $2N-3$ (this will be the case if the switch was initially down). Observe that since each non-leader flicks the switch at most twice, the former case corresponds to each non-leader flicking it twice, while the latter corresponds to $N-2$ non-leaders flicking the switch twice and one non-leader flicking it only once. In either case, every non-leader has flicked the switch at least once, and thus, everyone has visited the room at least once. Thus, it is safe for the leader to make the announcement when the count reaches $2N - 2$. ■

5 Liveness Requirements

Solution 5.1: Observe that a trace ρ satisfies the eventuality-formula $\Diamond\varphi$ iff ρ satisfies φ at a position $j \geq 1$, that is, at position 1, or at position $j \geq 2$, in which case the trace satisfies $\Diamond\varphi$ at position 2, that is, it satisfies $\bigcirc\Diamond\varphi$ at position 1. This means that the eventuality-formula $\Diamond\varphi$ is equivalent to $\varphi \vee \bigcirc\Diamond\varphi$.

By a similar reasoning, the until-formula $\varphi_1\,\mathcal{U}\,\varphi_2$ is satisfied exactly when either φ_2 is satisfied, or φ_1 is satisfied and the until-formula holds in the next position. Thus, the until-formula $\varphi_1\,\mathcal{U}\,\varphi_2$ is equivalent to $\varphi_2 \vee [\varphi_1 \wedge \bigcirc(\varphi_1\,\mathcal{U}\,\varphi_2)]$. ■

Solution 5.2: 1. The two are not equivalent. Suppose φ_1 is $x = 0$ and φ_2 is $x = 1$. Then if along the trace ρ the value of x alternates between 0 and 1, then ρ satisfies $(\Diamond\varphi_1 \wedge \Diamond\varphi_2)$, but not $\Diamond(\varphi_1 \wedge \varphi_2)$.

The formula $\Diamond(\varphi_1 \wedge \varphi_2)$ is stronger than $(\Diamond\varphi_1 \wedge \Diamond\varphi_2)$. Suppose a trace ρ satisfies $\Diamond(\varphi_1 \wedge \varphi_2)$. Then there exists a position j where both φ_1 and φ_2 are satisfied, and this implies that the trace satisfies both $\Diamond\varphi_1$ and $\Diamond\varphi_2$.

2. The two formulas are equivalent. A trace ρ satisfies the eventuality-formula $\Diamond(\varphi_1 \vee \varphi_2)$ precisely when $(\rho, j) \models (\varphi_1 \vee \varphi_2)$ for some position j. This holds precisely when $(\rho, j) \models \varphi_1$ or $(\rho, j) \models \varphi_2$ for some position j. This is equivalent to saying that the trace ρ satisfies either $\Diamond\varphi_1$ or $\Diamond\varphi_2$, that is, it satisfies $(\Diamond\varphi_1 \vee \Diamond\varphi_2)$.

3. The two are not equivalent. We can use the same example as in part 1. Suppose φ_1 is $x = 0$ and φ_2 is $x = 1$ and along the trace ρ the value of x alternates between 0 and 1. Then, ρ satisfies both $\Box\Diamond\varphi_1$ and $\Box\Diamond\varphi_2$, but not $\Box\Diamond(\varphi_1 \wedge \varphi_2)$.

The formula $\Box\Diamond(\varphi_1 \wedge \varphi_2)$ is stronger than $(\Box\Diamond\varphi_1 \wedge \Box\Diamond\varphi_2)$. Suppose a trace ρ satisfies $\Box\Diamond(\varphi_1 \wedge \varphi_2)$. Then there exist infinitely many positions where both φ_1 and φ_2 are satisfied, and this implies that the trace satisfies both $\Box\Diamond\varphi_1$ and $\Box\Diamond\varphi_2$.

4. The two formulas are equivalent. We will establish implication in each direction.

Suppose a trace satisfies $\Box\Diamond(\varphi_1 \vee \varphi_2)$. Then, there are infinitely many positions where the disjunction $\varphi_1 \vee \varphi_2$ holds. At each of these infinitely many positions either φ_1 holds or φ_2 holds. It follows that at least one of the formulas φ_1 or φ_2 must be satisfied at infinitely many positions. If φ_1 holds at infinitely many positions, then the trace satisfies $\Box\Diamond\varphi_1$, and if φ_2 holds at infinitely many positions, then the trace satisfies $\Box\Diamond\varphi_2$. In either case, the trace satisfies the disjunction $(\Box\Diamond\varphi_1 \vee \Box\Diamond\varphi_2)$.

Conversely, suppose the trace satisfies the disjunction $(\Box\Diamond\varphi_1 \vee \Box\Diamond\varphi_2)$. Then, the trace must satisfy either $\Box\Diamond\varphi_1$ or $\Box\Diamond\varphi_2$. If the former (the latter case is symmetric), then there are infinitely many positions where φ_1 is satisfied, and thus, so is the disjunction $(\varphi_1 \vee \varphi_2)$. Then the trace must satisfy $\Box\Diamond(\varphi_1 \vee \varphi_2)$. ■

Solution 5.3: The two formulas are not equivalent, and in fact, none is stronger than the other.

Consider a trace ρ such that the formula φ_1 holds at every position and the formula φ_2 holds at no position. Then the trace does not satisfy the until-formula $(\varphi_1 \,\mathcal{U}\, \varphi_2)$, and thus, satisfies $\neg(\varphi_1 \,\mathcal{U}\, \varphi_2)$. However, it does not satisfy the until-formula $(\neg\varphi_2)\;\mathcal{U}\;(\neg\varphi_1)$. This shows that $\neg(\varphi_1 \,\mathcal{U}\, \varphi_2)$ does not imply $(\neg\varphi_2)\;\mathcal{U}\;(\neg\varphi_1)$.

Conversely, consider a trace ρ such that at the first position φ_2 holds but φ_1 does not hold. By the semantics of the until-formulas, the trace satisfies both $(\neg\varphi_2)\;\mathcal{U}\;(\neg\varphi_1)$ and $(\varphi_1 \,\mathcal{U}\, \varphi_2)$. This shows that $(\neg\varphi_2)\;\mathcal{U}\;(\neg\varphi_1)$ does not imply $\neg(\varphi_1 \,\mathcal{U}\, \varphi_2)$. ■

Solution 5.4: The two formulas are equivalent. We will show that if a trace satisfies $\Box\Diamond(\varphi_1 \wedge \Diamond\varphi_2)$, then it must satisfy $\Box\Diamond(\varphi_2 \wedge \Diamond\varphi_1)$. By a symmetric argument, it follows that if a trace satisfies $\Box\Diamond(\varphi_2 \wedge \Diamond\varphi_1)$, then it must satisfy $\Box\Diamond(\varphi_1 \wedge \Diamond\varphi_2)$, establishing the desired equivalence.

Suppose the trace ρ satisfies $\Box\Diamond(\varphi_1 \wedge \Diamond\varphi_2)$. Then, there must be infinitely many positions j such that $(\rho, j) \models (\varphi_1 \wedge \Diamond\varphi_2)$. Then, there must be infinitely many positions j such that φ_1 holds at position j and φ_2 holds at some position $k \geq j$. It follows that there must be infinitely many positions k where φ_2 holds, and furthermore, at each such position k, there must be a position $j \geq k$ where φ_1 holds. Thus there are infinitely many positions k such that $(\rho, k) \models (\varphi_2 \wedge \Diamond\varphi_1)$. Hence, the trace ρ satisfies $\Box\Diamond(\varphi_2 \wedge \Diamond\varphi_1)$. ■

Solution 5.5: The desired requirement is expressed by the following LTL-formula:

$$\Box\Diamond(\mathit{inc} = 1) \;\rightarrow\; \Box\Diamond\,(\,\mathit{out}_0 = 1 \;\wedge\; \mathit{out}_1 = 1 \;\wedge\; \mathit{out}_2 = 1\,).$$

The circuit `3BitCounter` does not satisfy this specification. Suppose in every round both the input variables *start* and *inc* equal 1. The counter remains at zero in response to such a sequence of inputs. In the resulting execution the input *inc* is repeatedly high but the counter is never at its maximum value. This is a counterexample to the desired requirement. ■

Solution 5.6: The desired requirement is expressed by the following LTL-formula:

$$\Box\;[\,(\,\mathit{on} = 1 \wedge \Box\neg\,\mathit{cruise}\,? \wedge \Box\neg\,\mathit{inc}\,? \wedge \Box\neg\,\mathit{dec}\,?\,) \;\rightarrow\; \Diamond\Box\;(\,\mathit{speed} = \mathit{cruiseSpeed}\,)\,].$$

■

Solution 5.7: Consider a trace ρ and assume that it satisfies the formula φ_1. Then it satisfies either $\Box\Diamond\neg\,\mathit{Guard}(A)$ or $\Box\Diamond\,(\mathit{taken} = A)$. That is, there are infinitely many positions where either $\neg\,\mathit{Guard}(A)$ holds or $(\mathit{taken} = A)$ holds.

This implies that at every position j, there is a position $k \geq j$, where the disjunction $(\mathit{taken} = A) \vee \neg\, \mathit{Guard}(A)$ is satisfied. It follows that the always-formula φ_2 holds since to show that φ_2 is satisfied, we simply need to establish that every position j where the condition $\mathit{Guard}(A)$ holds is followed by a position $k \geq j$ where the disjunction $(\mathit{taken} = A) \vee \neg\, \mathit{Guard}(A)$ holds.

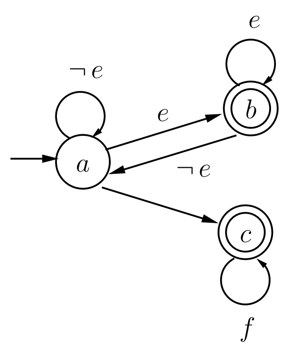
In the converse direction, suppose that a trace ρ satisfies the formula φ_2. To show that it also satisfies φ_1, assume that the trace also satisfies the antecedent $\Diamond \Box\, \mathit{Guard}(A)$ of φ_1. Then there exists a position j such that for all positions $k \geq j$, $\mathit{Guard}(A)$ holds at position k. Since the trace satisfies φ_2, it follows that for all positions $k \geq j$, the formula $\Diamond\, ((\mathit{taken} = A) \vee \neg\, \mathit{Guard}(A))$ holds. Since $\neg\, \mathit{Guard}(A)$ does not hold at any of these positions, it follows that for every position $k \geq j$, there is a later position where $(\mathit{taken} = A)$ is satisfied. This means that the trace satisfies the consequent $\Box \Diamond\, (\mathit{taken} = A)$ of φ_1. ■

Solution 5.8 : 1. Along an execution where the input task A_z is executed at each step, the value of x stays stuck at 0, and thus, the process does not satisfy $\Diamond\, (x > 5)$. If we assume weak fairness for the task A_x, since it is always enabled, we are guaranteed that it will be executed repeatedly along every fair execution. Thus, the value of x is guaranteed to be incremented repeatedly, and the process satisfies the eventuality property $\Diamond\, (x > 5)$.

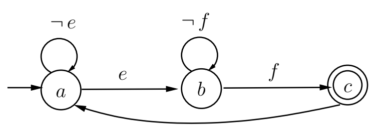
2. Along an execution where the task A_x is executed at each step, the value of y stays stuck at 0, and thus, the process does not satisfy $\Diamond\, (y > 5)$. In this specific execution, the value of z is also 0 at every step, and thus, the task A_y is never enabled and the execution is strongly-fair to the task A_y. Thus the specification is not satisfied even if we add fairness assumptions.

3. Along an execution where the input task A_z is executed at each step, the antecedent $\Box \Diamond\, (z = 1)$ is satisfied, but the value of y is stuck at 0. Thus, the specification $\Box \Diamond\, (z = 1) \;\; \rightarrow \;\; \Diamond\, (y > 5)$ is not satisfied. This execution is weakly-fair to the task A_y. However, if we assume strong fairness for the task A_y, then in every execution that satisfies the antecedent $\Box \Diamond\, (z = 1)$, the task A_y is repeatedly enabled, and is guaranteed to be executed repeatedly ensuring the satisfaction of $\Diamond\, (y > 5)$. Thus, the specification is satisfied assuming strong-fairness for the task A_y. ■

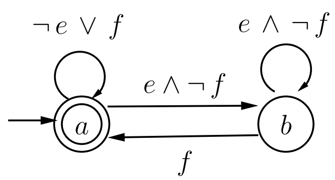
Solution 5.9 : 1. The Büchi automaton is shown below. The transitions between the initial state a and the accepting state b are similar to the automaton of figure 5.5 for the formula $\Box \Diamond\, e$, and the nondeterministic switch to the accepting state c is similar to the automaton of figure 5.6 for the formula $\Diamond \Box\, f$. A trace is accepted exactly when either the state b is visited repeatedly, which can happen only when the trace contains infinitely many positions satisfying e, or when the state c is visited repeatedly, which can happen exactly when the property f holds persistently.



2. The Büchi automaton is shown below. In the initial state a, the automaton is waiting for an input where e holds, and on such an input, switches to the state b. In the state b, it is waiting for an input where f holds, and on such an input, switches to the accepting state c and then at the next step returns to the initial state. The accepting state is visited infinitely often precisely when the input trace satisfies e repeatedly and f repeatedly.



3. The Büchi automaton is shown below. The automaton switches to the state b when it encounters an input where e holds but f does not hold. In the state b, it is waiting to satisfy the formula $e\,\mathcal{U}\,f$: it returns to the initial accepting state a on an input that satisfies f and continues to wait as long as the inputs satisfy e.



■

Solution 5.10: Observe that the input e must be 1 at every step for the automaton to take a transition. In addition, the accepting state is visited repeatedly only if the input trace contains infinitely many positions where f equals 1. Thus a trace is accepted by the Büchi automaton of figure 5.9 precisely when it satisfies the LTL-formula $\Box\, e \wedge \Box \Diamond\, f$. ■

Solution 5.11: Suppose the Büchi automaton M_1 has states Q_1, initial states $Init_1$, accepting states F_1, and edges E_1, and the automaton M_2 has states Q_2, initial states $Init_2$, accepting states F_2, and edges E_2. We first construct the "product" automaton M_{12} over the input variables V as follows: the set of states is $Q_1 \times Q_2$, the set of initial states is $Init_1 \times Init_2$, and for every edge $(q_1, Guard_1, q_1')$ in E_1 and every edge $(q_2, Guard_2, q_2')$ in E_2, the product M_{12} has an edge from the state (q_1, q_2) to the state (q_1', q_2') with the guard condition $Guard_1 \wedge Guard_2$. Observe that given a trace $\rho = v_1 v_2 \cdots$ over the input variables, $(q_0^1, q_0^2) \xrightarrow{v_1} (q_1^1, q_1^2) \xrightarrow{v_2} \cdots$ is an execution of the automaton M_{12} over ρ exactly when both $q_0^1 \xrightarrow{v_1} q_1^1 \xrightarrow{v_2} \cdots$ is an execution of the automaton M_1 over ρ and $q_0^2 \xrightarrow{v_1} q_1^2 \xrightarrow{v_2} \cdots$ is an execution of the automaton M_2 over ρ. If we view the automaton M_{12} as a generalized Büchi automaton with two accepting sets $F_1 \times Q_2$ and $Q_1 \times F_2$, then such an execution of M_{12} is an accepting execution when the first component of state is in the accepting set F_1 of M_1 infinitely many times and the second component of state is in the accepting set F_2 of M_2 infinitely many times. Thus, such a generalized Büchi automaton accepts input trace ρ exactly when both the automata M_1 and M_2 accept ρ. We can now use the construction of proposition 5.1 to convert this generalized Büchi automaton with two accepting sets to a standard Büchi automaton (by adding a counter that cycles through values $1, 2, 0$ in the state). ■

Solution 5.12: The claim does not hold. As a counterexample to the claim, consider the automaton of figure 5.5 that accepts exactly those traces that satisfy $\Box \Diamond\, e$. If we keep states, initial states, and edges unchanged, and declare the state a to be accepting (instead of b), then the resulting automaton accepts an input trace exactly when it satisfies the recurrence-formula $\Box \Diamond \neg e$. However, this set of traces does not coincide with the set of traces not accepted by the original automaton. In particular, the input trace in which the value of e alternates between 0 and 1 is accepted by both. Since the automaton of figure 5.5 is deterministic, the claim does not hold even for deterministic Büchi automata. ■

Solution 5.13: The closure $Sub(\varphi)$ is the set $\{e, f, \Diamond e, \bigcirc \Diamond e, \Box \Diamond e, \bigcirc \Box \Diamond e, \Box f, \bigcirc \Box f, \varphi\}$. The tableau has the following 32 states (that is, there are 16 consistent subsets of $Sub(\varphi)$):

$$
\begin{aligned}
q_0 &= \{e, f, \bigcirc \Diamond e, \bigcirc \Box \Diamond e, \bigcirc \Box f, \Diamond e, \Box \Diamond e, \Box f, \varphi\};\\
q_1 &= \{e, f, \bigcirc \Diamond e, \bigcirc \Box \Diamond e, \Diamond e, \Box \Diamond e, \varphi\};\\
q_2 &= \{e, f, \bigcirc \Diamond e, \bigcirc \Box f, \Diamond e, \Box f, \varphi\};\\
q_3 &= \{e, f, \bigcirc \Diamond e, \Diamond e\};\\
q_4 &= \{e, f, \bigcirc \Box \Diamond e, \bigcirc \Box f, \Diamond e, \Box \Diamond e, \Box f, \varphi\};\\
q_5 &= \{e, f, \bigcirc \Box \Diamond e, \Diamond e, \Box \Diamond e, \varphi\};\\
q_6 &= \{e, f, \bigcirc \Box f, \Diamond e, \Box f, \varphi\};\\
q_7 &= \{e, f, \Diamond e\};\\
q_8 &= \{e, \bigcirc \Diamond e, \bigcirc \Box \Diamond e, \bigcirc \Box f, \Diamond e, \Box \Diamond e, \varphi\};\\
q_9 &= \{e, \bigcirc \Diamond e, \bigcirc \Box \Diamond e, \Diamond e, \Box \Diamond e, \varphi\};
\end{aligned}
$$

$$
\begin{aligned}
q_{10} &= \{\, e, \bigcirc\Diamond e, \bigcirc\Box f, \Diamond e \,\};\\
q_{11} &= \{\, e, \bigcirc\Diamond e, \Diamond e \,\};\\
q_{12} &= \{\, e, \bigcirc\Box\Diamond e, \bigcirc\Box f, \Diamond e, \Box\Diamond e, \varphi \,\};\\
q_{13} &= \{\, e, \bigcirc\Box\Diamond e, \Diamond e, \Box\Diamond e, \varphi \,\};\\
q_{14} &= \{\, e, \bigcirc\Box f, \Diamond e \,\};\\
q_{15} &= \{\, e, \Diamond e \,\};\\
q_{16} &= \{\, f, \bigcirc\Diamond e, \bigcirc\Box\Diamond e, \bigcirc\Box f, \Diamond e, \Box\Diamond e, \Box f, \varphi \,\};\\
q_{17} &= \{\, f, \bigcirc\Diamond e, \bigcirc\Box\Diamond e, \Diamond e, \Box\Diamond e, \varphi \,\};\\
q_{18} &= \{\, f, \bigcirc\Diamond e, \bigcirc\Box f, \Diamond e, \Box f, \varphi \,\};\\
q_{19} &= \{\, f, \bigcirc\Diamond e, \Diamond e \,\};\\
q_{20} &= \{\, f, \bigcirc\Box\Diamond e, \bigcirc\Box f, \Box f, \varphi \,\};\\
q_{21} &= \{\, f, \bigcirc\Box\Diamond e \,\};\\
q_{22} &= \{\, f, \bigcirc\Box f, \Box f, \varphi \,\};\\
q_{23} &= \{\, f \,\};\\
q_{24} &= \{\, \bigcirc\Diamond e, \bigcirc\Box\Diamond e, \bigcirc\Box f, \Diamond e, \Box\Diamond e, \varphi \,\};\\
q_{25} &= \{\, \bigcirc\Diamond e, \bigcirc\Box\Diamond e, \Diamond e, \Box\Diamond e, \varphi \,\};\\
q_{26} &= \{\, \bigcirc\Diamond e, \bigcirc\Box f, \Diamond e \,\};\\
q_{27} &= \{\, \bigcirc\Diamond e, \Diamond e \,\};\\
q_{28} &= \{\, \bigcirc\Box\Diamond e, \bigcirc\Box f, \Diamond e, \Box\Diamond e, \varphi \,\};\\
q_{29} &= \{\, \bigcirc\Box\Diamond e \,\};\\
q_{30} &= \{\, \bigcirc\Box f \,\};\\
q_{31} &= \{\,\}.
\end{aligned}
$$

Of these, 18 states that contain φ (such as state q_0 and q_{28}) are initial states. The edges are defined by the rules of the tableau. We give edges out of a few states as examples. The state q_0 has edges, all with the guard $e \wedge f$, to states that contain all of $\Diamond e$, $\Box\Diamond e$, and $\Box f$, that is, to states q_0, q_4, and q_{16}. The state q_{10} has edges, all with the guard $e \wedge \neg f$, to states that include both $\Diamond e$ and $\Box f$ but exclude $\Box\Diamond e$, that is, to states q_2, q_6, and q_{18}. The state q_{31} has edges, all with the guard $\neg e \wedge \neg f$, to states that exclude all of $\Diamond e$, $\Box\Diamond e$, and $\Box f$, that is, to states q_{21}, q_{23}, q_{29}, q_{30}, and itself. The automaton M_φ has three accepting sets $F_{\Box f}$, $F_{\Diamond e}$, and $F_{\Box\Diamond e}$. The set $F_{\Box f}$ contains states that either include $\Box f$ or exclude f (there are 24 such states). The other two accepting sets are defined similarly. ■

Solution 5.14: To handle until-subformulas, the definition of the closure $Sub(\varphi)$ is extended as follows: for a subexpression of the form $\psi_1 \,\mathcal{U}\, \psi_2$ of φ, the closure contains the next-formula $\bigcirc(\psi_1 \,\mathcal{U}\, \psi_2)$. The definition of consistency for a subset $q \subseteq Sub(\varphi)$ of the closure now also demands: $\psi_1 \,\mathcal{U}\, \psi_2$ belongs to q exactly when either ψ_2 is in q or both ψ_1 and $\bigcirc(\psi_1 \,\mathcal{U}\, \psi_2)$ belong to q. With this revised definition of the closure and its consistent subsets, the input variables, states, initial states, and edges of the tableau M_φ are defined as before. In addition to the accepting sets corresponding to the always and eventuality subformulas, now for every until-subformula $\psi = \psi_1 \,\mathcal{U}\, \psi_2$ in the closure $Sub(\varphi)$, there is an accepting set F_ψ containing states q such that either $\psi_2 \in q$ or $\psi \notin q$.

The construction can be proved correct exactly as in the proof of proposition 5.2. Whenever an until-formula $\psi = \psi_1 \,\mathcal{U}\, \psi_2$ belongs to a state q along an accepting run of the tableau, the consistency condition ensures that either ψ_2 belongs to the state q (and in such a case, the satisfaction of ψ_2 suffices to ensure the satisfaction of ψ), or both ψ_1 and $\bigcirc\psi$ belong to q. In the latter case, the rules for adding edges in the tableau ensure that the next state along the execution contains ψ, and the accepting condition imposed by F_ψ ensures the eventual satisfaction of ψ_2 later in the execution. ■

Solution 5.15: Let $\varphi = (e \,\mathcal{U}\, \bigcirc f) \vee \neg e$ and let $\psi = (e \,\mathcal{U}\, \bigcirc f)$. Then

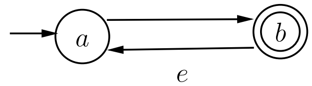
$$Sub(\varphi) \;=\; \{\, e, \neg e, f, \bigcirc f, \psi, \bigcirc \psi, \varphi \,\}.$$

The tableau has the following 16 states (that is, there are 16 consistent subsets of $Sub(\varphi)$):

$$\begin{array}{lllllll}
q_0 & = & \{\, e, f, \bigcirc f, \bigcirc\psi, \psi, \varphi \,\}; & q_1 & = & \{\, e, f, \bigcirc f, \psi, \varphi \,\}; \\
q_2 & = & \{\, e, f, \bigcirc\psi, \psi, \varphi \,\}; & q_3 & = & \{\, e, f \,\}; \\
q_4 & = & \{\, e, \bigcirc f, \bigcirc\psi, \psi, \varphi \,\}; & q_5 & = & \{\, e, \bigcirc f, \psi, \varphi \,\}; \\
q_6 & = & \{\, e, \bigcirc\psi, \psi, \varphi \,\}; & q_7 & = & \{\, e \,\}; \\
q_8 & = & \{\, \neg e, f, \bigcirc f, \bigcirc\psi, \psi, \varphi \,\}; & q_9 & = & \{\, \neg e, f, \bigcirc f, \psi, \varphi \,\}; \\
q_{10} & = & \{\, \neg e, f, \bigcirc\psi, \varphi \,\}; & q_{11} & = & \{\, \neg e, f, \varphi \,\}; \\
q_{12} & = & \{\, \neg e, \bigcirc f, \bigcirc\psi, \psi, \varphi \,\}; & q_{13} & = & \{\, \neg e, \bigcirc f, \psi, \varphi \,\}; \\
q_{14} & = & \{\, \neg e, \bigcirc\psi, \varphi \,\}; & q_{15} & = & \{\, \neg e, \varphi \,\}.
\end{array}$$

Of these all states but q_3 and q_7 contain φ and are initial states. The edges are defined by the rules of the tableau. We give edges out of a few states as examples. The state q_0 has edges, all with the guard $e \wedge f$, to states that contain both f and ψ, that is, to states q_0, q_1, q_2, q_8, and q_9. The state q_{11} has edges, all with the guard $\neg e \wedge f$, to states that exclude both f and ψ, that is, to states q_7, q_{14}, and q_{15}. The state q_{13} has edges, all with the guard $\neg e \wedge \neg f$, to states that include f but exclude ψ, that is, to states q_3, q_{10} and q_{11}. The automaton M_φ has a single accepting set F_ψ: it contains all states that either include $\bigcirc f$ or exclude ψ, that is, all states except q_2 and q_6. ■

Solution 5.16: For the Büchi automaton shown below, on a given input trace, the automaton has an infinite execution (that is, the automaton can keep processing the inputs at each step) exactly when the input e has value 1 in every even position. If this is the case, the input trace is accepted since such an infinite execution contains the accepting state b repeatedly.



■

Solution 5.17: Let $\rho = v_1 v_2 \cdots$ be an input trace, and suppose $q_0 \xrightarrow{v_1} q_1 \xrightarrow{v_2} \cdots$ is an accepting execution of the automaton M_φ over ρ. We want to prove that for all $\psi \in Sub(\varphi)$, for all $i \geq 0$, $\psi \in q_i$ if and only if $(\rho, i+1) \models \psi$. The proof is by induction on the structure of ψ: we assume that for all smaller subformulas ψ' of ψ, the claim holds, that is, for all $i \geq 0$, $\psi' \in q_i$ if and only if $(\rho, i+1) \models \psi'$, and then prove the claim for ψ.

Suppose ψ is an atomic formula e. Consider a state q_i in the execution for $i \geq 0$. The tableau M_φ must have an edge from the state q_i to state q_{i+1} with guard-condition *Guard* such that the input v_{i+1} satisfies the guard. From the definition of the edges of the tableau, if $e \in q_i$, then e is a conjunct of *Guard*, and otherwise, $\neg e$ is a conjunct of *Guard*. It follows that if $e \in q_i$, then the input v_{i+1} satisfies e, and otherwise, the input v_{i+1} satisfies $\neg e$. This means that for all $i \geq 0$, $e \in q_i$ if and only if $(\rho, i+1) \models e$.

Suppose ψ is of the form $\neg \psi'$. Consider a state q_i for $i \geq 0$. Since ψ' is a subformula of ψ, by induction hypothesis, we can assume that $\psi' \in q_i$ if and only if $(\rho, i+1) \models \psi'$. Since the state q_i is a consistent set of subformulas, we know that it contains exactly one of ψ' or $\psi = \neg \psi'$. It follows that $\psi \in q_i$ if and only if $(\rho, i+1) \models \psi$.

Suppose ψ is of the form $\psi_1 \wedge \psi_2$. Consider a state q_i for $i \geq 0$. From consistency of q_i, we know that it contains ψ exactly when it contains both of ψ_1 and ψ_2. By induction hypothesis, we know that $\psi_1 \in q_i$ if and only if $(\rho, i+1) \models \psi_1$ and $\psi_2 \in q_i$ if and only if $(\rho, i+1) \models \psi_2$. Finally, $(\rho, i+1) \models \psi$ exactly when both $(\rho, i+1) \models \psi_1$ and $(\rho, i+1) \models \psi_2$. It follows that $\psi \in q_i$ if and only if $(\rho, i+1) \models \psi$.

The proof for the case when ψ is of the form $\psi_1 \vee \psi_2$ is analogous.

Suppose ψ is of the form $\Diamond \psi'$. Consider a state q_i in the execution for $i \geq 0$. We will prove that $\psi \in q_i$ if and only if $(\rho, i+1) \models \psi$ by establishing the implication in both directions.

Suppose $(\rho, i+1) \models \psi$. Then, let $j+1 \geq i+1$ be the smallest position such that $(\rho, j+1) \models \psi'$ (such a position must exist by the semantics of eventuality formulas). We prove, for $k = j, j-1, \ldots i$, by backward induction, that each state q_k along this execution fragment contains ψ. Since $(\rho, j+1) \models \psi'$ and ψ' is a subformula of ψ, by the induction hypothesis, it follows that $\psi' \in q_j$. Since the state q_j is consistent, also $\psi \in q_j$. Now assume that the state q_k contains ψ, for $j > k > i$, and we want to prove that the state q_{k-1} also contains ψ. Since the automaton has a transition from the state q_{k-1} to state q_k, and $\bigcirc \psi$ is in the closure, from the definition of the tableau edges, we can conclude that $\bigcirc \psi$ belongs to q_{k-1}. Then, by the definition of consistency, the state q_{k-1} must contain ψ.

Now suppose $\psi \in q_i$. From the definition of consistency, q_i contains either ψ' or $\bigcirc \psi$. In the latter case, by the definition of the edges of the tableau, ψ must be in q_{i+1}. Furthermore $q_i, q_{i+1}, \ldots$ is an accepting execution, which means that it cannot be the case that $q_j \notin F_\psi$ for all $j \geq i$. This implies that $\psi' \in q_j$ for some $j \geq i$. From the inductive hypothesis, we know that $(\rho, v_{j+1}) \models \psi'$. It follows that $(\rho, i+1) \models \psi$.

The proof for the case when ψ is of the form $\Box\,\psi'$ is analogous. ■

Solution 5.18: The algorithm of figure 5.11 returns 1 when it finds a state s that is reachable, satisfies φ, and belongs to a cycle. Upon termination, the stack *Pending* contains a sequence of states that demonstrates the reachability of the state s (with s on the top of the stack and an initial state at the bottom). However, the cycle that contains s is not stored by the given algorithm. We can modify the algorithm by introducing a second stack called *NPending*. The stack is initialized as: `stack(state)` *NPending* := `EmptyStack`. The function *NDFS* when invoked pushes its argument state on this stack by executing `Push`(s, *NPending*) as the first instruction. Furthermore, before terminating unsuccessfully (that is, by returning 0) *NDFS* pops the stack by executing the instruction `Pop`(*NPending*) (this statement should be added just before the last instruction `return` 0).

Now when the algorithm terminates by returning 1, the stack *Pending* contains a sequence of states $s_0 s_1 \ldots s_k$ (with s_k at the top), the stack *NPending* contains a sequence of states $t_0 t_1 \ldots t_j$ (with t_j at the top), such that (1) s_0 is an initial state, (2) there is a transition from each state s_i to state s_{i+1} for $0 \leq i < k$, (3) the state s_k satisfies the property φ, (4) the state s_k at the top of the stack *Pending* equals the state t_0 at the bottom of the stack *NPending*, (5) there is a transition from each state t_i to state t_{i+1} for $0 \leq i < j$, and (6) there is a transition from the state t_j to some state s_i, $0 \leq i \leq k$, in the stack *Pending*. Thus, the desired counterexample can be output from the contents of the two stacks. ■

Solution 5.19: The transition formula *Trans* is given by

$$(\,x > y \ \wedge\ x' = x+1 \ \wedge\ y' = y\,) \ \vee\ (\,x \leq y \ \wedge\ x' = x \ \wedge\ y' = x\,).$$

To obtain the pre-image of the region A, we first rename the variables x and y in the description of A to x' and y', respectively, and conjoin the result with the transition formula. This gives:

$$\begin{array}{ll} & [\,x > y \ \wedge\ x' = x+1 \ \wedge\ y' = y \ \wedge\ 1 \leq y' \leq 5\,] \\ \vee & [\,x \leq y \ \wedge\ x' = x \ \wedge\ y' = x \ \wedge\ 1 \leq y' \leq 5\,]. \end{array}$$

Existential quantification of the variables x' and y' from this formula gives us the desired pre-image:

$$(\,x > y \ \wedge\ 1 \leq y \leq 5\,) \ \vee\ (\,x \leq y \ \wedge\ 1 \leq x \leq 5\,).$$

Note that this region is equivalent to the formula

$$x \geq 1 \ \wedge\ y \geq 1 \ \wedge\ (\,x \leq 5 \ \vee\ y \leq 5\,).$$

■

Solution 5.20: With this modification, the inner loop computes the set of states that can be reached in one or more transitions from the states in *Recur*. Thus, each set $Recur_i$ now contains exactly those states in $Recur_{i-1}$ that are reachable from some state in $Recur_{i-1}$ in one or more transitions. The modified algorithm is not correct.

The proof of correctness of the original algorithm established that: if a reachable state s satisfying φ appears in an infinite execution along which φ holds repeatedly, then the state s is never removed from the set *Recur*. This claim does not hold for the modified algorithm. As a counterexample, consider a transition system with a single variable x of type `nat` with all states initial and with the transitions given by the statement $x := x + 1$. An execution for this system is of the form $n, n+1, n+2, \cdots$. Suppose the property φ is true in every state, then each such execution demonstrates the desired repeatability, and each state appears on such an execution. However, in the modified algorithm, $Recur_0$ is the region $x \geq 0$, $Recur_1$ is the region $x \geq 1$ (the state 0 is removed in the first iteration since it is not reachable using one or more transitions), $Recur_2$ is the region $x \geq 2$, and so on.

More significantly, the algorithm can give a wrong result: even if $Recur_i = Recur_{i-1}$, we are not guaranteed the existence of an infinite execution in which the property φ holds repeatedly. As a counterexample, consider a transition system with a single variable x of type `nat` with all states initial and with the transitions given by the statement `if` $x > 0$ `then` $x := x - 1$. An execution for this system is of the form $n, n-1, n-2, \cdots, 0$. Suppose the property φ is true in every state. It is not repeatable since the transition system has no infinite execution. However, $Recur_0 = x \geq 0$, and every state n in this region is reachable from the state $n+1$ also in this region using one transition. Thus $Recur_1$ equals $Recur_0$ in the modified algorithm, and the algorithm terminates incorrectly concluding repeatability. ■

Solution 5.21: For the transition system $\texttt{GCD}(m, n)$, the state variables are x and y of type `nat` and *mode* ranging over $\{\texttt{loop}, \texttt{stop}\}$. We want to establish that the transition system satisfies the eventuality formula $\Diamond\,(mode = \texttt{stop})$. Observe that a transition of this system corresponds to executing the self-loop on the mode `loop` or the mode-switch from `loop` to `stop`. The latter causes the goal of the eventuality formula to be satisfied, while the former causes either x or y to be decremented by a non-zero amount. As a result, we choose the sum of x and y to be the ranking function: for a state s, $rank(s) = s(x) + s(y)$. Observe that the range of the ranking function is the set of natural numbers since both x and y range over natural numbers. To show that executing a system transition either leads to the goal or decreases the rank, we don't need to restrict attention to a subset of the states. That is, we choose ψ to be the property 1 which holds in all states, and thus, is an invariant. Consider a transition (s, t) of the system. If this transition corresponds to the mode-switch from `loop` to `stop`, then the property $(mode = \texttt{stop})$ holds in the state t. Otherwise, the transition corresponds to executing the self-loop on the mode `loop`. In such a case, the guard-condition $(x > 0 \ \wedge \ y > 0)$ must be true in the state s. From the update code of the self-loop, we know that either $t(x) = s(x) - s(y)$ and $t(y) = s(y)$, or $t(y) = s(y) - s(x)$ and $t(x) = s(x)$. In either case, $t(x + y) < s(x + y)$, that is, $rank(t) < rank(s)$. From the proof-rule for eventuality properties, it follows that the transition system $\mathtt{GCD}(m, n)$ satisfies the eventuality formula $\Diamond\,(mode = \mathtt{stop})$. ■

Solution 5.22: The task A_1 is always enabled. Thus the weak-fairness assumption for the task A_1 simplifies to the condition $\Box\Diamond\,(taken = A_1)$. We want to establish that the transition system satisfies the conditional response formula

$$\Box\Diamond\,(taken = A_1) \ \rightarrow \ \Box\Diamond\,(x = 0).$$

In order to apply the proof rule for conditional response property, the properties ψ_1 and φ_1 are always true, the property ψ_2 corresponds to $(taken = A_1)$, and the property φ_2 corresponds to $(x = 0)$. As the invariant ψ we choose the property that is always true. Observe that the conditions (1) and (2) in the proof rule are satisfied. The ranking function maps a state to the value of x. Consider a transition (s, t) of the system.

If $s(x) = 0$, then the state s satisfies the desired response φ_2. Note that in such a case, if the transition (s, t) corresponds to the execution of the task A_1, then $t(x) = s(y)$ and thus $rank(t)$ can be greater than $rank(s)$. This is acceptable since such an increase occurs only when the response property $\varphi_2 : (x = 0)$ becomes true (our use of the rule differs slightly from the precise statement in the textbook: in our case the source s of the transition satisfies φ_2 rather than the target t).

Now suppose $s(x) > 0$. If the transition corresponds to the execution of the task A_1, then $t(taken) = A_1$ and $t(x) = s(x) - 1$. In this case, the state t satisfies ψ_2 and $rank(t) < rank(s)$. If the transition corresponds to the execution of the task A_2, then $t(taken) = A_2$ and $t(x) = s(x)$. In this case, the state t does not satisfy ψ_2 and $rank(t) = rank(s)$.

Thus, the condition (3) of the proof rule is satisfied, and it follows that the transition system satisfies the desired conditional response property. ■

6 Dynamical Systems

Solution 6.1: Yes, the continuous-time component modeling the simple pendulum has Lipschitz-continuous dynamics. The rate of change of the angle φ is given by the expression ν, which is linear, and hence, Lipschitz-continuous function from `real` to `real`. The rate of change of the angular velocity ν is a linear combination of $\mathtt{sin}\,\varphi$ which is Lipschitz-continuous since $\|\mathtt{sin}\ \varphi\| \leq 1$ for all values of φ, and the input u which is also Lipschitz-continuous, and thus, is Lipschitz-continuous. ■

Solution 6.3: Suppose the dynamics of the component H_1 is given by $\dot{x} = f_1(x, u)$ and $v = h_1(x, u)$, and the dynamics of the component H_2 is given by $\dot{y} = f_2(y, v)$ and $w = h_2(y, v)$. We know that the functions f_1, h_1, f_2, and h_2 are all Lipschitz-continuous.

The parallel composition $H = H_1 \| H_2$ has input variable u, state variables x and y, and output variables v and w. Its dynamics is specified by

$$\begin{aligned}
&\dot{x} = f_1(x, u);\\
&\dot{y} = f_3(x, y, u) = f_2(y, h_1(x, u));\\
&v = h_1(x, u); \text{ and}\\
&w = h_3(x, y, u) = h_2(y, h_1(x, u)).
\end{aligned}$$

To show that the component H is Lipschitz-continuous we need to show that the functions f_1, f_3, h_1, and h_3 are Lipschitz-continuous. We already know that f_1 and h_1 are Lipschitz-continuous. Below we show that the function f_3 is Lipschitz-continuous. The proof for the Lipschitz-continuity of h_3 is analogous.

To prove Lipschitz-continuity of f_3, we need to find a constant K such that for all (x, y, u) and (x', y', u'), $\|f_3(x, y, u) - f_3(x', y', u')\| \leq K\|(x, y, u) - (x', y', u')\|$. We know that the functions h_1 and f_2 are Lipschitz-continuous. Let the corresponding Lipschitz constants be K_1 and K_2 respectively.

$$\begin{array}{rcl}
\|f_3(x, y, u) - f_3(x', y', u')\| & = & \|f_2(y, h_1(x, u)) - f_2(y', h_1(x', u'))\| \\
& \leq & K_2\, \|(y, h_1(x, u)) - (y', h_1(x', u'))\| \\
& \leq & K_2\, [\|y - y'\| + \|h_1(x, u) - h_1(x', u')\|] \\
& \leq & K_2\, [\|y - y'\| + K_1\, \|(x, u) - (x', u')\|] \\
& \leq & K_2\, [\|(x, y, u) - (x', y', u')\| + K_1\, \|(x, y, u) - (x', y', u')\|] \\
& = & K_2(K_1 + 1)\|(x, y, u) - (x', y', u')\|
\end{array}$$

■

Solution 6.4: The continuous-time component of figure 6.8 has two state variables. To calculate equilibria of the system, we set the rate of change of each state variable to be 0. This gives the equations $v = 0$ and $(-kv - m\, g\, \mathtt{sin}\,\theta)/m = 0$ (since the input force F is also 0). These equations can be satisfied only when $\mathtt{sin}\,\theta$ equals 0. When the grade θ is 5 degrees, this does not hold, and hence, the model does not have any equilibrium state. ■

Solution 6.5: To find equilibrium states, we set $3\,s_1+4\,s_2 = 0$ and $2\,s_1+s_2 = 0$. Since these two equations are linearly independent, the system of equations has a unique solution, namely, $s_1 = s_2 = 0$. Thus, the origin $(0,0)$ is the sole equilibrium of the system. To check whether this equilibrium is stable, suppose we perturb the state to, say, $s_0 = (\delta,\delta)$ for a small value of $\delta > 0$. Then, the system response $\overline{S}_0(t)$ is the solution to the linear differential equation $\dot{s}_1 = 3\,s_1 + 4\,s_2$ and $\dot{s}_2 = 2\,s_1 + s_2$ with the initial state s_0. While it is possible to solve this initial value problem, for current purpose simply observe that at time 0, both the quantities $\dot{s}_1$ and $\dot{s}_2$ are positive values causing the initial state s_0 to flow away from the origin. The magnitudes of the rates only increase as a result causing the resulting signal $\overline{S}_0(t)$ to diverge. That is, no matter how small a value of $\delta > 0$ we choose, for the initial state $s_0 = (\delta,\delta)$, the quantity $\|\overline{S}_0(t)\|$ grows unboundedly. Thus, the equilibrium $(0,0)$ is unstable. ■

Solution 6.6: Setting $x^2 - x = 0$ gives us two equilibria: $x = 0$ and $x = 1$. To analyze the stability of the equilibrium $x = 0$, let us consider the behavior of the system starting from the initial state $x_0 = \delta$, for $0 < \delta < 0.5$. Then the rate of change $\delta^2 - \delta$ is negative, and furthermore, the magnitude of this rate decreases as the state gets closer to the equilibrium. The resulting response signal $\overline{x}(t)$ converges to 0 with $0 \le \overline{x}(t) \le \delta$ for all t. Analogously, if the initial state is $x_0 = -\delta$, for $0 < \delta < -0.5$, Then the rate of change is positive, and the resulting response signal converges to 0 with $-\delta \le \overline{x}(t) \le 0$ for all t. Thus, the equilibrium 0 is asymptotically stable.

For the equilibrium $x = 1$, if we set the initial state to $x_0 = 1 + \delta$, for $\delta > 0$, then no matter how small δ we choose, the rate of change is positive with its magnitude increasing as the state moves away from the equilibrium 1. Thus, the response signal $\overline{x}(t)$ diverges, and the equilibrium $x = 1$ is unstable. ■

Solution 6.7: We model the tuning fork as a closed continuous-time component H with two state variables x—denoting the displacement, and v—denoting the velocity. Initially, the displacement x equals x_0 and the velocity v equals 0. The dynamics is given by $\dot{x} = v$ and $\dot{v} = (-k/m)\,x$. The output of the system can be the displacement x.

We want to find the solution to the differential equation $\ddot{x} = (-k/m)\,x$ with the initial condition $\overline{x}(0) = x_0$. If we set $\overline{x}(t) = b\,\texttt{cos}\,(a\,t)$, then $\overline{x}(0) = b$, $(d/dt)\,\overline{x}(t) = -b\,a\,\texttt{sin}\,(a\,t)$, and $(d^2/dt^2)\,\overline{x}(t) = -b\,a^2\,\texttt{cos}\,(a\,t)$ which equals $-a^2\,\overline{x}(t)$. Thus, if we choose $b = x_0$ and $a = \sqrt{k/m}$, we have the desired solution. That is, for the initial displacement x_0, the state response of the tuning fork is described by the signal $\overline{x}(t) = x_0\,\texttt{cos}\,(\sqrt{k/m}\;t)$. This corresponds to the fork oscillating between x_0 to $-x_0$ in a perpetual rhythmic motion.

Setting $\dot{x} = 0$ and $\dot{v} = 0$ gives the sole equilibrium of the system: $x = 0$ and $v = 0$. This corresponds to the situation where the fork is stationary and vertical. If we set the initial displacement to x_0, we know that the state response is given by $\overline{x}(t) = x_0\,\texttt{cos}\,(\sqrt{k/m}\;t)$. For such a signal, $\|\overline{x}(t)\| \le \|x_0\|$

for all t. This means that the equilibrium $x = 0$ is stable. However, it is not asymptotically stable: as time increases, the state keeps oscillating between x_0 and $-x_0$ without converging to 0, that is, $\overline{x}(t) = x_0$ for infinitely many times t no matter how small the initial displacement x_0 is. ■

Solution 6.8: Consider two states $\overline{x} = (x_1, x_2, \ldots, x_n)$ and $\overline{y} = (y_1, y_2, \ldots y_n)$ in $\mathtt{real}^n$. Then

$$\begin{aligned}
\|e(\overline{x}) - e(\overline{y})\| &= \|(a_1 x_1 + \cdots + a_n x_n) - (a_1 y_1 + \cdots + a_n y_n)\| \\
&\leq \sum_{i=1}^{n} \|a_i x_i - a_i y_i\| \\
&= \sum_{i=1}^{n} a_i \, \|x_i - y_i\|.
\end{aligned}$$

The Euclidean distance between two points $\overline{x}$ and $\overline{y}$ is at least the magnitude of difference between the coordinates x_i and y_i along a specific axis i. This gives:

$$\|e(\overline{x}) - e(\overline{y})\| \;\leq\; \sum_{i=1}^{n} a_i \, \|\overline{x} - \overline{y}\|.$$

This means that for all states $\overline{x}$ and $\overline{y}$, $\|e(\overline{x}) - e(\overline{y})\| \leq K\|\overline{x} - \overline{y}\|$, where $K = a_1 + a_2 + \cdots + a_n$. This establishes the Lipschitz-continuity of the expression e. ■

Solution 6.9: For the car model of figure 4.6, with input grade θ replaced by the disturbance $d = m\,g\,\mathtt{sin}\,\theta$, we have the state variables $S = \{x, v\}$, input variables $I = \{F, d\}$, and output variables $O = \{v\}$. Thus, $m = n = 2$ and $k = 1$. The dynamics is given the equations $\dot{x} = v$ and $\dot{v} = (F - kv - d)/m$. Note that the output v equals the state variable v. Then, the desired matrices are:

$$A = \begin{bmatrix} 0 & 1 \\ 0 & -k/m \end{bmatrix}; \quad B = \begin{bmatrix} 0 & 0 \\ 1/m & -1/m \end{bmatrix}; \quad C = [0 \;\; 1]; \quad D = [0 \;\; 0].$$

■

Solution 6.10: Let θ_0 be a constant such that the value of the $\mathtt{sin}$ function in the range $[0, \theta_0]$ is small. Then, we can declare the state variables φ and ν by

$$\mathtt{real}\,[-\theta_0, \theta_0]\;\varphi := \varphi_0;\;\; \nu := 0.$$

The input and output variables are the same as the model in figure 6.6. The revised linear dynamics is given by the equations:

$$\dot{\varphi} \;=\; \nu; \quad \dot{\nu} \;=\; (-g/\ell)\,\varphi \;+\; (1/m\ell^2)\,u.$$

■

Solution 6.11: Consider a closed linear component H with state variables S, output variables O, and the dynamics given by $\dot{S} = AS$ and $O = CS$. Suppose $\overline{S}_0$ and $\overline{O}_0$ are the state and output responses, respectively, starting from the initial state s_0, and $\overline{S}_1$ and $\overline{O}_1$ are the state and output responses, respectively, starting from the initial state s_1. Thus, $\overline{S}_0(0) = s_0$ and for all t, $(d/dt)\,\overline{S}_0(t) = A\,\overline{S}_0(t)$, and $\overline{S}_1(0) = s_1$ and for all t, $(d/dt)\,\overline{S}_1(t) = A\,\overline{S}_1(t)$. Also for all t, $\overline{O}_0(t) = C\,\overline{S}_0(t)$ and $\overline{O}_1(t) = C\,\overline{S}_1(t)$.

Given constants $\alpha, \beta \in$ `real`, consider the state signal $\overline{S}(t) = \alpha\,\overline{S}_0(t) + \beta\,\overline{S}_1(t)$. Then,

$$\overline{S}(0) \;=\; \alpha\,\overline{S}_0(0) \,+\, \beta\,\overline{S}_1(0) \;=\; \alpha\,s_0 \,+\, \beta\,s_1,$$

and for all times t,

$$\begin{aligned}
(d/dt)\,\overline{S}(t) &= (d/dt)\,[\,\alpha\,\overline{S}_0(t) \,+\, \beta\,\overline{S}_1(t)\,] \\
&= \alpha\,(d/dt)\,\overline{S}_0(t) \,+\, \beta\,(d/dt)\,\overline{S}_1(t) \\
&= \alpha\,A\,\overline{S}_0(t) \,+\, \beta\,A\,\overline{S}_1(t) \\
&= A\,[\,\alpha\,\overline{S}_0(t) \,+\, \beta\,\overline{S}_1(t)\,] \\
&= A\,\overline{S}(t)
\end{aligned}$$

Thus, the state signal $\overline{S}(t)$ is the solution to the linear differential equation $\dot{S} = AS$ with the initial state $\alpha\,s_0 \,+\, \beta\,s_1$, and thus must be the state response of the component H starting from the initial state $\alpha\,s_0 + \beta\,s_1$. The corresponding output response then

$$\begin{aligned}
\overline{O}(t) &= C\,\overline{S}(t) \\
&= C\,[\,\alpha\,\overline{S}_0(t) \,+\, \beta\,\overline{S}_1(t)\,] \\
&= \alpha\,C\,\overline{S}_0(t) \,+\, \beta\,C\,\overline{S}_1(t) \\
&= \alpha\,\overline{O}_0(t) \,+\, \beta\,\overline{O}_1(t)
\end{aligned}$$

This establishes that the output response of the component H from the initial state $\alpha\,s_0 \,+\, \beta\,s_1$ is the signal $\alpha\,\overline{O}_0 \,+\, \beta\,\overline{O}_1$. ■

Solution 6.12: Suppose the dynamics is given by $\dot{s} \,=\, a\,s + b\,u$. As explained in section 6.2.2, the state signal starting from the initial states s_0 in response to the input signal $\overline{u}(t)$ is given by:

$$\overline{s}(t) \;=\; e^{at}s_0 \;+\; \int_0^t e^{a(t-\tau)}\,b\,\overline{u}(\tau)\,d\tau.$$

Setting the input signal to $\overline{u}(t) = c$, we get

$$\begin{aligned}
\overline{s}(t) &= e^{at}s_0 \;+\; \int_0^t e^{a(t-\tau)}\,b\,c\,d\tau \\
&= e^{at}s_0 \;+\; e^{at}b\,c \int_0^t e^{-a\tau}d\tau \\
&= e^{at}s_0 \;+\; e^{at}b\,c\,(1 - e^{-at})/a \\
&= e^{at}s_0 \;+\; b\,c\,(e^{at} - 1)/a.
\end{aligned}$$

Verify that $\overline{s}(0)$ evaluates to s_0, and differentiating the expression for $\overline{s}(t)$ gives $(d/dt)\,\overline{s}(t) \;=\; a\,e^{at}s_0 \,+\, b\,c\,e^{at}$ which equals $a\,\overline{s}(t) \,+\, b\,c$ as desired. ■

Solution 6.13: For this system, the dynamics matrix is

$$A = \begin{bmatrix} -1 & 2 \\ 0 & 1 \end{bmatrix}.$$

The characteristic polynomial $\mathtt{det}(\,A \,-\, \lambda\,\mathbf{I})$ turns out to be $\lambda^2 - 1$. As a result, we have two eigenvalues $\lambda_1 = 1$ and $\lambda_2 = -1$. To obtain the eigenvector $x_1 = [x_{11} \;\; x_{12}]^T$ corresponding to the eigenvalue 1, we need to solve the equation $A\,x_1 = x_1$. This gives $x_{11} = x_{12}$, and we choose $[1 \;\; 1]^T$ as the eigenvector x_1. To obtain the eigenvector $x_2 = [x_{21} \;\; x_{22}]^T$ corresponding to the eigenvalue -1, we need to solve the equation $A\,x_2 = x_2$, which gives $x_{22} = 0$, and we choose $[1 \;\; 0]^T$ as the eigenvector x_2.

To compute the state response by diagonalization using theorem 6.2, the transformation matrix is:

$$P \;=\; [\,x_1 \;\; x_2\,] \;=\; \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

To compute the inverse P^{-1}, we can view the entries in the desired matrix as unknowns and solve the system of simultaneous linear equations given by $P\,P^{-1} = \mathbf{I}$. This gives

$$P^{-1} \;=\; \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}.$$

Starting in the initial state s_0, the state of the system at time t is described by $P\,\mathbf{D}(e^t, e^{-t})\,P^{-1}s_0$. If the initial state vector s_0 is $[s_{01} \;\;\; s_{02}]^T$, then by calculating the matrix products, we get a closed-form solution for the state of the system at time t:

$$\begin{aligned} \overline{S}_1(t) &= e^{-t}\,s_{01} \;+\; (e^t - e^{-t})\,s_{02} \\ \overline{S}_2(t) &= e^t\,s_{02}. \end{aligned}$$

■

Solution 6.14: For this exercise, the dynamics matrix is

$$A = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}.$$

The characteristic polynomial then is $\lambda^2 + 3\lambda + 2$. This gives two eigenvalues $\lambda_1 = -1$ and $\lambda_2 = -2$. By solving the equation $A\,x_1 = -x_1$, we get the first eigenvector $[1 \;\;\; -1]^T$, and by solving the equation $A\,x_2 = -2\,x_2$, we get the second eigenvector $[1 \;\;\; -2]^T$.

To compute the state response by diagonalization using theorem 6.2, the transformation matrix is:

$$P \;=\; [\,x_1 \;\; x_2\,] \;=\; \begin{bmatrix} 1 & 1 \\ -1 & -2 \end{bmatrix}.$$

The inverse of this matrix is:

$$P^{-1} = \begin{bmatrix} 2 & 1 \\ -1 & -1 \end{bmatrix}.$$

Starting in the initial state s_0, the state of the system at time t is described by $P\,\mathbf{D}(e^t, e^{-t})\,P^{-1}s_0$. This gives a closed-form solution for the state of the system at time t:

$$\begin{aligned} \overline{S}_1(t) &= (\,2\,e^{-t} - e^{-2t}\,)\,s_{01} + (\,e^{-t} - e^{-2t}\,)\,s_{02} \\ \overline{S}_2(t) &= 2\,(\,-e^{-t} + e^{-2t}\,)\,s_{01} + (\,-e^{-t} + 2\,e^{-2t}\,)\,s_{02}. \end{aligned}$$

■

Solution 6.15 : For this example, the dynamics matrix is

$$A = \begin{bmatrix} 3 & 4 & 0 \\ 0 & 2 & 0 \\ 4 & 0 & 9 \end{bmatrix}.$$

The characteristic polynomial then is $(3-\lambda)(2-\lambda)(9-\lambda)$. This gives three eigenvalues $\lambda_1 = 2$, $\lambda_2 = 3$, and $\lambda_3 = 9$. By solving the equation $A\,x_1 = 2\,x_1$, we get the first eigenvector $[-28\;\;7\;\;16]^T$, by solving the equation $A\,x_2 = 3\,x_2$, we get the second eigenvector $[3\;\;0\;\;-2]^T$, and by solving the equation $A\,x_3 = 9\,x_3$, we get the third eigenvector $[0\;\;0\;\;1]^T$.

To compute the state response by diagonalization using theorem 6.2, the transformation matrix is:

$$P = [\,x_1\;\;x_2\;\;x_3\,] = \begin{bmatrix} -28 & 3 & 0 \\ 7 & 0 & 0 \\ 16 & -2 & 1 \end{bmatrix}.$$

The inverse of this matrix is:

$$P^{-1} = \begin{bmatrix} 0 & 1/7 & 0 \\ 1/3 & 4/3 & 0 \\ 2/3 & 8/21 & 1 \end{bmatrix}.$$

Starting in the initial state s_0, the state of the system at time t is described by $P\,\mathbf{D}(e^{2t}, e^{3t}, e^{9t})\,P^{-1}s_0$. This gives a closed-form solution for the state of the system at time t:

$$\begin{aligned} \overline{S}_1(t) &= e^{3t}\,s_{01} + 4\,(e^{3t} - e^{2t})\,s_{02} \\ \overline{S}_2(t) &= e^{2t}\,s_{02} \\ \overline{S}_3(t) &= 2/3\,(e^{9t} - e^{3t})\,s_{01} + 1/21\,(48\,e^{2t} - 56\,e^{3t} + 8\,e^{9t})\,s_{02} + e^{9t}\,s_{03}. \end{aligned}$$

■

Solution 6.16: Let us first note the following property of vector norms: if A is an $(n \times n)$-matrix, then there exists a constant Δ_A such that for all n-dimensional vectors x, $\|A\,x\| \leq \Delta_A\,\|x\|$. Let us prove this for the Euclidean norm. Suppose $A_1, A_2, \ldots A_n$ are the rows of the matrix A. Then the entries of the vector $A\,x$ are the dot-products $A_1\,x$, $A_2\,x, \ldots, A_n\,x$. Then,

$$\begin{aligned}
(\|A\,x\|)^2 &= \sum_{i=1}^{n} (\|A_i\,x\|)^2 \\
&\leq \sum_{i=1}^{n} (\|A_i\|\,\|x\|)^2 \\
&= (\|x\|)^2 \sum_{i=1}^{n} (\|A_i\|)^2.
\end{aligned}$$

In the above the second inequality follows from the Cauchy-Schwartz inequality: the magnitude of the dot-product of two vectors is bounded by the product of the norms of the two vectors. This calculation shows that if we choose Δ_A to be the square-root of $\sum_{i=1}^{n} (\|A_i\|)^2$, then for every vector x, $\|A\,x\| \leq \Delta_A\,\|x\|$.

Now we proceed to prove proposition 6.1. Consider an n-dimensional linear system H with state vector S and dynamics $\dot{S} = A\,S$ and another n-dimensional linear system H' with state vector S' and dynamics $\dot{S}' = J\,S'$, where $J = P^{-1}A\,P$ for some invertible matrix P. We will prove that if the system H is stable, then so is the system H'.

Suppose 0 is a stable equilibrium of the system H. To show stability of the system H', we need to prove that for every $\epsilon' > 0$, there exists $\delta' > 0$ such that for the state signal $\overline{S'}$ of H' starting in an initial state s_0' with $\|s_0'\| < \delta'$, for all times t, $\|\overline{S}'(t)\| < \epsilon'$. Consider an $\epsilon' > 0$. Let ϵ be $\epsilon'/\Delta_{P^{-1}}$. Since the system H is stable, given such an ϵ, there exists $\delta > 0$ such that whenever the initial state of H is δ-close to the origin, its state response stays ϵ-close to the origin at all times. Choose $\delta' = \delta/\Delta_P$. Consider an initial state s_0' of H with $\|s_0'\| < \delta'$. Let $s_0 = P\,s_0'$. We know that

$$\begin{aligned}
\|s_0\| &\leq \Delta_P\,\|s_0'\| \\
&< \Delta_P\,\delta' \\
&= \Delta_P\,(\delta/\Delta_P) \\
&= \delta.
\end{aligned}$$

Let $\overline{S}$ be the state signal of H starting from the initial state s_0. We know that for all time t, $\|\overline{S}(t)\| < \epsilon$. Define the signal $\overline{S}'(t) = P^{-1}\overline{S}(t)$. Observe that $\overline{S}'(0) = P^{-1}s_0 = P^{-1}P\,s_0' = s_0'$, and for all times t,

$$\begin{aligned}
(d/dt)\,\overline{S}'(t) &= (d/dt)\,P^{-1}\overline{S}(t) \\
&= P^{-1}(d/dt)\,\overline{S}(t)
\end{aligned}$$

$$
\begin{aligned}
&= P^{-1} A \overline{S}(t) \\
&= P^{-1} A P \overline{S}'(t) \\
&= J \overline{S}'(t).
\end{aligned}
$$

Thus, the signal $\overline{S}'(t)$ is the state response of the system H' starting from the initial state s_0'. Now for all times t,

$$
\begin{aligned}
\|\overline{S}'(t)\| &= \|P^{-1}\overline{S}(t)\| \\
&\leq \Delta_{P^{-1}} \|\overline{S}(t)\| \\
&< \Delta_{P^{-1}}\ \epsilon \\
&= \Delta_{P^{-1}}\ \epsilon'/\Delta_{P^{-1}} \\
&= \epsilon'.
\end{aligned}
$$

Thus, we have proved that the state of the system H' stays ϵ'-close to the origin at all times provided its initial state is δ'-close to the origin. The proof that if the system H' is stable, then so is the system H is identical with the roles of the transformations P and P^{-1} reversed. The proof that the system H is asymptotically stable if and only if the system H' is asymptotically stable is analogous. ■

Solution 6.17: For the system in exercise 6.13, one of the eigenvalues is 1, and hence, the system is unstable. For the system in exercise 6.14, all the eigenvalues are negative, and hence, the system is asymptotically stable. For the system in exercise 6.15, all the eigenvalues are positive, and hence, the system is unstable. ■

Solution 6.18: For the given dynamical system

$$
A = \begin{bmatrix} 1/2 & 1 \\ 1 & 2 \end{bmatrix};\ B = \begin{bmatrix} 1 \\ 1 \end{bmatrix};\ \mathbf{C}(A,B) = \begin{bmatrix} 1 & 3/2 \\ 1 & 3 \end{bmatrix}.
$$

For the matrix A, the characteristic polynomial is $\lambda^2 - 5\,\lambda/2$. Thus the eigenvalues are 0 and $5/2$. From the stability test for linear systems, we can conclude that the system is unstable.

For the controllability matrix $\mathbf{C}(A, B)$, the two columns are independent and its rank is 2. From theorem 6.5, we can conclude that the system is controllable.

The desired gain matrix F is a (1×2)-matrix, and let its entries be f_1 and f_2. Consider the matrix $A - BF$:

$$
\begin{bmatrix} 1/2 & 1 \\ 1 & 2 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} [f_1 \quad f_2] = \begin{bmatrix} 1/2 - f_1 & 1 - f_2 \\ 1 - f_1 & 2 - f_2 \end{bmatrix}.
$$

The characteristic polynomial for this matrix is

$$
\begin{aligned}
P(\lambda, f_1, f_2) &= (1/2 - f_1 - \lambda)(2 - f_2 - \lambda) - (1 - f_2)(1 - f_1); \\
&= \lambda^2 + (f_1 + f_2 - 5/2)\lambda + (f_2/2 - f_1).
\end{aligned}
$$

The roots of this characteristic polynomial are λ_1 and λ_2 exactly when

$$\begin{aligned} f_1 + f_2 - 5/2 &= -\lambda_1 - \lambda_2; \\ f_2/2 - f_1 &= \lambda_1 \lambda_2. \end{aligned}$$

If we want the eigenvalues to be $-1+j$ and $-1-j$, then we need to solve

$$f_1 + f_2 - 5/2 \;=\; 2; \quad f_2/2 - f_1 \;=\; 2.$$

Solving these equations give us $f_1 = 1/6$ and $f_2 = 13/3$. This means that with the choice of the gain matrix F to be $[1/6 \;\; 13/3]$, the resulting closed-loop system is asymptotically stable with eigenvalues $-1+j$ and $-1-j$. ■

Solution 6.19: For the dynamical system in this exercise

$$A \;=\; \begin{bmatrix} 0 & -2 \\ 1 & -3 \end{bmatrix}; \; B \;=\; \begin{bmatrix} 1 \\ 1 \end{bmatrix}; \; \mathbf{C}(A,B) \;=\; \begin{bmatrix} 1 & -2 \\ 1 & -2 \end{bmatrix}.$$

For the matrix A, the characteristic polynomial is $\lambda^2 + 3\lambda + 2$. Thus the eigenvalues are -2 and -1. For the controllability matrix $\mathbf{C}(A,B)$, the two rows are identical, and thus, the rank is 1 and the system is not controllable.

Setting the entries of the gain matrix to be f_1 and f_2, consider the matrix $A - BF$:

$$\begin{bmatrix} 0 & -2 \\ 1 & -3 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} [f_1 \;\; f_2] = \begin{bmatrix} -f_1 & -2-f_2 \\ 1-f_1 & -3-f_2 \end{bmatrix}.$$

The characteristic polynomial for this matrix is

$$\begin{aligned} P(\lambda, f_1, f_2) &= (f_1+\lambda)(3+f_2+\lambda) + (2+f_2)(1-f_1); \\ &= \lambda^2 + (f_1+f_2+3)\lambda + (f_1+f_2+2). \end{aligned}$$

The roots of this characteristic polynomial are λ_1 and λ_2 exactly when $-\lambda_1 - \lambda_2 = f+1$ and $\lambda_1\lambda_2 = f$, where $f = f_1 + f_2 + 2$. For any given f, $\lambda_1 = -1$ and $\lambda_2 = -f$ is the solution to this system of equations. This means that one of the eigenvalues of the matrix $A - BF$ is always -1. If we wish the other eigenvalue to be e, we can choose the entries f_1 and f_2 of the gain matrix so that $e = -(f_1 + f_2 + 2)$. ■

Solution 6.22: The sequence of values computed by Euler's method for simulation is given by: for every $i \geq 0$, $s_{i+1} = s_i + \Delta\, s_i = (1+\Delta)\, s_i$. Thus, $s_1 = (1+\Delta)\, s_0$, $s_2 = (1+\Delta)\, s_1 = (1+\Delta)^2\, s_0$, and after n steps of simulation, the state s_n equals $(1+\Delta)^n\, s_0$. For $s_0 = 2$, $\Delta = 0.1$, and $n = 50$, we get $s_{50} = 2\,(1.1)^{50}$, which equals 234.78171. ■

Solution 6.23: Consider the calculation of the state s_{i+1} from state s_i using the second-order Runge-Kutta method: $k_1 = s_i$, $k_2 = s_i + \Delta\, s_i$, and $s_{i+1} = s_i + \Delta\,(k_1 + k_2)/2$. This gives $s_{i+1} = (1 + \Delta + \Delta^2/2)\, s_i$. Thus, after n steps of simulation, the state s_n equals $(1+\Delta+\Delta^2/2)^n\, s_0$. For $s_0 = 2$, $\Delta = 0.1$, and $n = 50$, we get $s_{50} = 2\,(1.105)^{50}$, which equals 294.53974. ■

Solution 6.24: Suppose the barrier function Ψ is $7\,s_1^2\,-\,6\,s_1\,s_2\,+\,28\,s_2^2\,+\,k$. The Lie derivative $\mathcal{L}_A\,\Psi$ does not depend on the constant k, and equals the expression $-146\,s_1^2\,-\,566\,s_2^2\,+\,564\,s_1\,s_2$, which is always negative. Thus, the condition (3) in theorem 6.6 is satisfied independent of k.

The choice of k should be such that for every initial state s, $\Psi(s) \leq 0$. Since the initial region is a rectangle, it suffices to ensure that the value of Ψ is non-positive at all its corners. Setting $\Psi(5,1) \leq 0$ gives $k \leq -173$; setting $\Psi(5,-1) \leq 0$ gives $k \leq -233$; setting $\Psi(6,1) \leq 0$ gives $k \leq -244$; and setting $\Psi(6,-1) \leq 0$ gives $k \leq -316$. Thus, as long as $k \leq -316$, the condition (1) in theorem 6.6 is satisfied.

Finally, we need to ensure that the value of Ψ is positive in all unsafe states. Since the safe region is convex, and we have required the initial region to be safe, it suffices to show that the horizontal lines $s_2 = 4$ and $s_2 = -4$ do not intersect the boundary of the barrier. Thus, we want to ensure that if $\Psi(s) = 0$, then s_2 cannot be either 4 or -4. Setting $\Psi(s) = 0$ and $s_2 = 4$ gives the quadratic equation $7\,s_1^2\,-\,24\,s_1\,+\,448\,+\,k\,=0$. This equations has no solution if $24^2\,<\,4\cdot 7\cdot(448+k)$, that is, when $k > -427.42$. Setting $\Psi(s) = 0$ and $s_2 = -4$ gives the quadratic equation $7\,s_1^2 + 24\,s_1 + 448 + k\,= 0$, which does not have a solution if $k > -427.42$. This implies that if $k > -427.42$, the condition (2) in theorem 6.6 is satisfied.

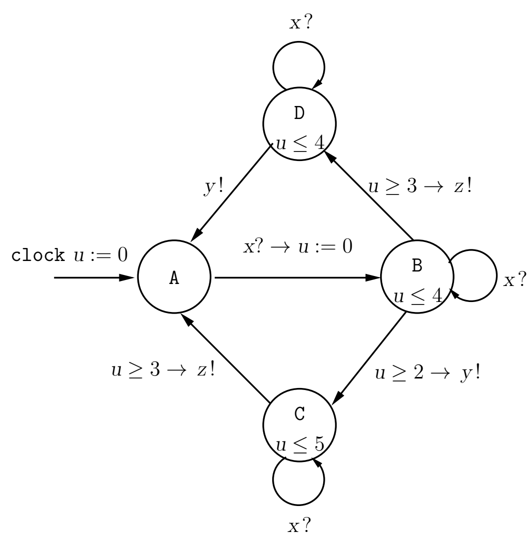
Thus, $7\,s_1^2\,-\,6\,s_1\,s_2\,+\,28\,s_2^2\,+\,k$ is a barrier certificate exactly when $k \leq -316$ and $k > -427.42$. ■

Solution 6.25: The revised rule is not sound as shown below. Consider a single-dimensional dynamical system with the dynamics $\dot{x} = 1$. Suppose there is a single initial state $x_0 = 0$. The set of unsafe states is described by $x \geq 1$. Clearly, the system is not safe: starting at the initial state 0, the system enters the unsafe region at time 1. Consider the Barrier certificate $\Psi(x) = x^2$. In the initial state, $\Psi(x_0) = 0$, and thus, the condition (1) of theorem 6.6 holds. In every unsafe state, $x \geq 1$, and thus, $\Psi(x) \geq 1$, so condition (2) of theorem 6.6 also holds. The Lie derivative $(\mathcal{L}_f\,\Psi)(x)$ equals $(d/dx)\,x^2\,\cdot\dot{x} = 2\,x$. When $x^2 = 0$, we have that $x = 0$, and thus, $(\mathcal{L}_f\,\Psi)(x) = 0$. The condition (3) of theorem 6.6 requires this directional derivative to be negative. However, if we relax the condition and require it only to be non-positive, then the modified condition holds. But it would be incorrect to conclude the system to be safe.

Observe that in the above example, at $x = 0$, $(\mathcal{L}_f\,\Psi)(x) = 2\,x = 0$ and the *second-order* Lie derivative, $(\mathcal{L}_f^2\,\Psi)(x)$ equals 2, which is positive. Condition (3) of theorem 3.3 can be relaxed to say that “if $\Psi(S) = 0$ then either $(\mathcal{L}_f\,\Psi)(S) < 0$ or $(\mathcal{L}_f\,\Psi)(S) = 0$ and $(d/dt)(\mathcal{L}_f\,\Psi)(S) < 0$” (or more generally first k derivatives are 0 and the $(k+1)$th derivative is negative). ■

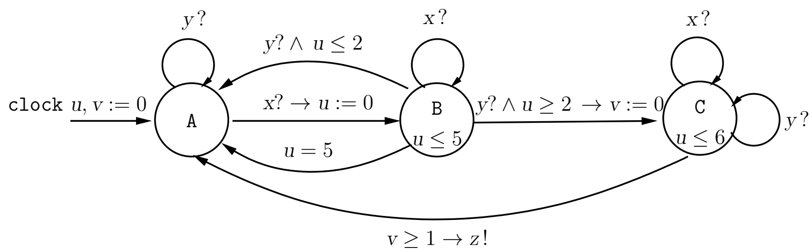
7 Timed Model

Solution 7.1: The desired behavior is specified by the timed state machine below. In the initial mode A, when the process receives an input event x, it switches to mode B, and the value of the clock variable u indicates the time elapsed since this mode-switch. The process can issue the output event y and switch to mode C when at least 2 time units have elapsed, and can issue the output event z and switch to mode D when at least 3 time units have elapsed. In mode C, the process generates the output event z returning to the initial mode (the clock-invariant $u \leq 5$ associated with mode C along with the guard $u \geq 3$ associated with the mode-switch from C to A enforce the desired bounds on the delay with respect to the relevant input event). Similarly, in mode D, the process generates the output event y returning to the initial mode. Note that when the process enters mode D, at least 3 time units have elapsed since the occurrence of the relevant input event, and thus, the lower bound on the delay for producing the output event y holds automatically.



■

Solution 7.2: The desired behavior is specified by the timed state machine below. In the initial mode, if the process receives an input event y, the state stays unchanged, and if it receives an input event x, it switches to mode B, where the value of the clock variable u indicates the time elapsed since this mode-switch. The clock-invariant of mode B ensures that the process can wait in this mode only for 5 time units, and if no event y is received until the condition $(u = 5)$ holds, the process returns to the initial mode. When the process receives an input event y while in mode B, if the condition $u \leq 2$ holds it returns to the initial mode, and otherwise, it resets the clock v and switches to mode C. The clock-invariant associated with mode C along with the guard associated with the mode-switch from `C` to `A` enforce the desired bounds on the delay between the output event z and the relevant input events x and y.



■

Solution 7.3: The process maintains three Boolean variables: a keeps track the value of the input channel x, b the value of the input channel y, and c the value of the desired output z. It also has one clock variable u. The Boolean state variables are initialized to 0 at the beginning. The process can be in three modes `Idle`, `Wait1`, and `Wait2`. The initial mode is `Idle`. Whenever an input event x (or y) occurs, it toggles the value of the corresponding state variable a (or b, respectively). Then it checks if this changes the value of $a \vee b$: if it does not, then the process remains in the idle mode, otherwise it switches to mode `Wait1` and resets the clock u to schedule a change in the output variable c. The process can stay in the mode `Wait1` for at most 1 time unit. While in `Wait1`, if another input event is received and this causes a change in the value of the condition $a \vee b$ back to the old value, the process cancels the scheduled output change and goes back to the idle mode. Once the time delay of 1 time unit is up, the process switches to mode `Wait2`, in which it is too late for any change in the condition $a \vee b$ to cancel the scheduled change in output value. While it is waiting in mode `Wait2`, if the process detects any input event, it simply toggles the corresponding state variable. In mode `Wait2`, once the clock exceeds 2, the process can execute an internal transition that toggles the output variable c, and switches back to either `Idle` or `Wait1` depending on whether or not the output variable is consistent with the inputs. The clock-invariant ensures that the process can wait in `Wait2` only as long as the clock does not exceed 4.

The declaration of state variables is given by

$$\texttt{bool}\ a := 0;\ b := 0;\ c := 0;\ \{\texttt{Idle}, \texttt{Wait1}, \texttt{Wait2}\}\ mode := \texttt{Idle};\ \texttt{clock}\ u := 0.$$

The input task for processing the input channel x is given by

$$\begin{array}{l} a := \neg a; \\ \texttt{if}\ (mode = \texttt{Idle}\ \wedge\ c \neq a \vee b)\ \texttt{then}\ \{\ u := 0;\ mode := \texttt{Wait1}\ \} \\ \texttt{else if}\ (mode = \texttt{Wait1}\ \wedge\ c = a \vee b)\ \texttt{then}\ mode := \texttt{Idle}. \end{array}$$

The task for processing the input channel y is analogous but toggles the state variable b first. The output task for the channel z is always enabled and simply transmits the current value of the state variable c on the channel z. There are two internal tasks. The first one is given by

$$(\, u = 1 \;\wedge\; \mathit{mode} = \texttt{Wait1}\,) \;\rightarrow\; \mathit{mode} := \texttt{Wait2}.$$

The second one is given by

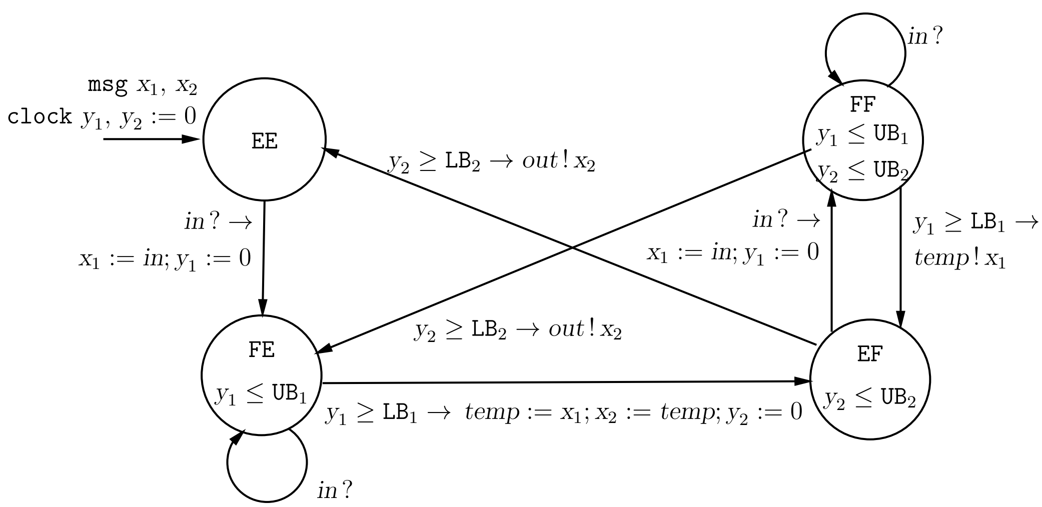
$$\begin{array}{l} (u \geq 2 \;\wedge\; \mathit{mode} = \texttt{Wait2}) \;\rightarrow\; \{ \\ \qquad c := \neg\, c; \\ \qquad \texttt{if } (c = a \vee b) \texttt{ then } \mathit{mode} := \texttt{Idle} \\ \qquad \texttt{else } \{\, u := 0;\; \mathit{mode} := \texttt{Wait1}\,\}\}. \end{array}$$

The clock-invariant is given by

$$(\,\mathit{mode} = \texttt{Idle}\,) \;\vee\; (\,\mathit{mode} = \texttt{Wait1} \;\wedge\; u \leq 1\,) \;\vee\; (\,\mathit{mode} = \texttt{Wait2} \;\wedge\; u \leq 4\,).$$

■

Solution 7.4: The product machine is shown below and is constructed in a manner similar to the product in figure 7.6.



■

Solution 7.5: Consider a reachable state s of the timed process `TimedInc` with $s(x_1) = m$ and $s(x_2) = n$. We want to establish that $m \leq 2n+2$ and $n \leq 2m+2$. Consider an execution of the timed process that leads to the state s and let t be the sum of the durations of all the timed actions during this execution. Since the task A_1 is the only task that updates the variable x_1, it must execute exactly m times during this execution, and let $t_1, t_2, \ldots t_m$ be the sequence of times when the task A_1 executes. The guard condition for the task A_1 ensures that $t_1 \geq 1$ and $t_{i+1} - t_i \geq 1$ for $i = 1, \ldots m-1$. It follows that $t_m \geq m$ and thus $t \geq m$. Furthermore, the clock-invariant of the process ensures that the total duration of timed actions between successive executions of the task A_1 is at most 2. That is, $t_1 \leq 2$ and $t_{i+1} - t_i \leq 2$ for $i = 1, \ldots m-1$ and $t - t_m \leq 2$. This implies that $t \leq 2m + 2$. By a symmetric argument based on analyzing when the task A_2 that increments x_2 executes, we can conclude that $t \geq n$ and $t \leq 2n + 2$. From the constraints $t \geq m$ and $t \leq 2n + 2$, we can conclude that $m \leq 2n + 2$, and the constraints $t \geq n$ and $t \leq 2m + 2$ imply that $n \leq 2m + 2$. ■

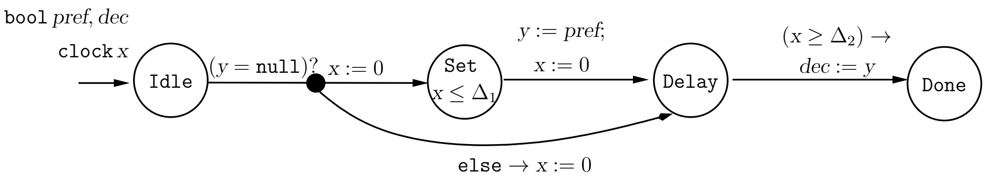
Solution 7.6: The two processes are not equivalent. As a counterexample, suppose $\mathtt{LB}_1 = \mathtt{UB}_1 = 1$ and $\mathtt{LB}_2 = \mathtt{UB}_2 = 2$. Then the timed process P switches to mode `B` when its (imperfect) clock x equals 1 and leaves mode `B` when the clock x equals 2. Due to the drift, this implies that the process P spends between $(1 - \epsilon)$ to $(1 + \epsilon)$ time units in the mode `A`, and between $(1 - \epsilon)$ to $(1 + \epsilon)$ time units in the mode `B`. For the process P', the guard of the mode-switch from `A` to `B` is $y \geq (1 - \epsilon)$, and the clock-invariant associated with mode `B` is $y \leq 2(1 + \epsilon)$. Thus, in one possible execution of the process P', it switches to mode `B` at time $(1 - \epsilon)$, spends a total of $(1 + 3\epsilon)$ time in mode `B`, and leaves the mode at time $(2 + 2\epsilon)$. Such a timing of events is not possible for the process P. ■

Solution 7.7: The protocol does not satisfy the starvation-freedom requirement. Consider the execution with two processes with the following sequence of actions:

1. Process P_1 changes its mode to `Test`.
2. Process P_2 changes its mode to `Test`.
3. Process P_1 reads *Turn*, finds it to be 0, and switches to mode `Set`.
4. Process P_2 reads *Turn*, finds it to be 0, and switches to mode `Set`.
5. Process P_1 writes 1 to *Turn*, sets its clock to 0, and switches to mode `Delay`.
6. Process P_2 writes 2 to *Turn*, sets its clock to 0, and switches to mode `Delay`.
7. A timed action of duration Δ_2 causes clocks of both processes to increase to Δ_2.
8. Process P_1 reads *Turn*, finds it to be 2, and changes its mode to *Test*.
9. Process P_2 reads *Turn*, finds it to be 2, and changes its mode to `Crit`.
10. Process P_2 exits its critical section by setting *Turn* to 0, and switches to mode `Idle`.
11. Process P_2 changes its mode to `Test`.

At the end of this execution, *Turn* equals 0 and both processes are in the mode `Test`, and thus, the sequence of actions from step 3 to 11 can be executed repeatedly. In the resulting infinite execution the process P_1 is starved: it is repeatedly in mode `try` without ever entering its critical section. ■

Solution 7.8: The protocol uses a single shared atomic register y ranging over $\{\texttt{null}, 0, 1\}$ initialized to `null`. The protocol executed by each process P is shown below. The variable *pref* contains the preference value of the process. When it starts executing the protocol, it first reads the shared register y. If y still has its initial value `null`, then the process writes its own preference to y. The clock-invariant $x \leq \Delta_1$ ensures that the delay between a process finding y to be `null` and writing its own preference to y is at most Δ_1. If initially the process finds y to be non-null, it skips the writing step and proceeds to `Delay` straight away. In mode `Delay`, the process waits for at least Δ_2 time units, reads the shared register y, chooses it be the decision value, and terminates by switching to the mode `Done`.



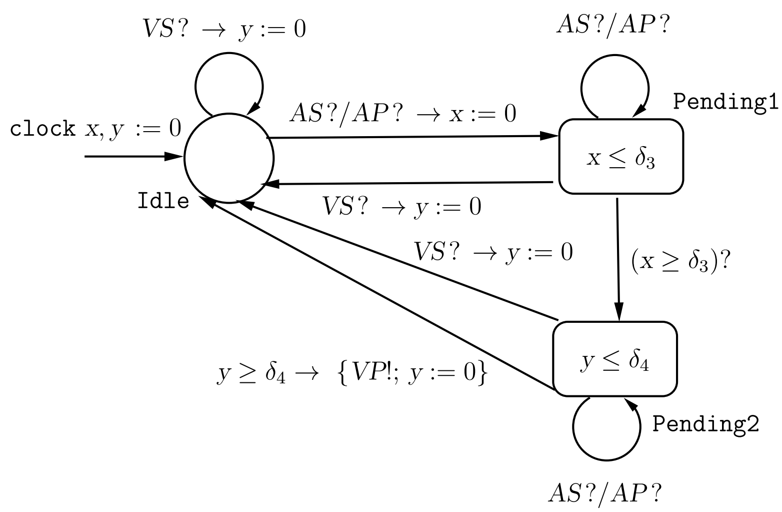
A process executes at most two read actions, one write action, and delays itself by at least Δ_2 time units. Thus it is not blocked by other processes and requirement of wait-freedom is satisfied. If a process finds the value of y to be `null`, it writes its own preference to y (which may be overwritten by preferences of other processes). Thus, when a process executes the read operation $dec := y$ before terminating, the value of y is guaranteed to be the preference of one of the processes, and thus, the requirement of validity is also satisfied. Assuming $\Delta_2 > \Delta_1$, the agreement requirement is satisfied. The reasoning is analogous to the one in Fischer's timing-based mutual exclusion protocol. Suppose a process enters mode `Delay` at time t_1 and leaves it at time t_2. Because of the guard-condition of the mode-switch out of the mode `Delay`, we know that $t_2 - t_1 \geq \Delta_2$. The process decides on the value of y it reads at time t_2. We argue that the value of y stays unchanged after time t_2. The value of y at time t_1 must be non-null. Thus a process that reads y for the first time after time t_1 will not write to it, and thus cannot cause the value of y to change. Consider a process that enters the mode `Set` at time $t' \leq t_1$. Because of the upper bound on how long a process can stay in `Set`, it must write to y at time no later than $t' + \Delta_1$, which is no later than $t_1 + \Delta_1$, which must be strictly smaller than t_2 if $\Delta_2 > \Delta_1$. Thus if some process decides on the value of y that it reads at time t_2, then no process writes to y after time t_2. This implies that the decision values are identical for all processes. ■

Solution 7.9: The table below shows the first few steps of a possible execution of the protocol when the message string 100110100 is supplied to the sender at time 0, where the error rate ϵ equals 0.25. It uses the same notational conventions as the ones used in figure 7.15. Note that the delay between the first two *up* events issued by the sender process is only 4.5 time units instead of 6 due to the skew. As a result the receiver can interpret the second bit as 1. At the end of this execution fragment the queue *out* starts with 11 which is not a prefix of the input message, and thus, this execution demonstrates a violation of the desired correctness requirement.

Time	Event	x	Sender	Queue m	y	Receiver	Queue *out*
0			`B`	00110100		`Idle`	`null`
2	*up*	2	`D`	0110100		`Last1`	1
5	*down*	3	`F`	110100	3	`Last1`	1
6.5	*up*	1.5	`G`	110100	4.5	`Last1`	11

■

Solution 7.10: In the modified timed process shown below, the mode `Pending` is split into two modes `Pending1` and `Pending2`. In mode `Idle`, when the process receives an atrial input event, it switches to `Pending1` with clock-invariant $x \leq \delta_3$. When the clock x reaches δ_3, the process makes an internal transition to the mode `Pending2` with clock-invariant $y \leq \delta_4$. In this mode when the clock y reaches δ_4, we can be sure that the clock x is at least δ_3, and the process switches back to mode `Idle` issuing the output event *VP*. In both modes `Pending1` and `Pending2`, atrial input events do not cause a change in state, and the process responds to the input event *VS* by switching back to `Idle`.



■

Solution 7.11: The region $[\mathtt{A}, x = 0, y > 1]$ has an edge labeled $b?$ to the region $[\mathtt{B}, x = y = 0]$, and has τ-labeled edges to the following three regions: $[\mathtt{A}, 0 < x < 1, y > 1]$, $[\mathtt{A}, x = 1, y > 1]$, and $[\mathtt{A}, 1 < x < 2, y > 1]$. The region $[\mathtt{B}, 0 < x = y < 1]$ has $c!$-labeled edge to the region $[\mathtt{A}, 0 < x = y < 1]$, τ-labeled edge to itself, and τ-labeled edge to the region $[\mathtt{B}, x = y = 1]$. ■

Solution 7.12: There are three clock-regions where all three clocks are equal to one another:

$$(x = y = z = 0),\ (0 < x = y = z < 1),\ (x = y = z = 1).$$

There are 11 clock-regions where only the clocks x and y are equal:

$$\begin{array}{l}(x = y = 0 < z < 1),\ (x = y = 0, z = 1),\ (x = y = 0, z > 1),\\ (z = 0 < x = y < 1),\ (0 < z < x = y < 1),\ (0 < x = y < z < 1),\\ (0 < x = y < 1 = z),\ (0 < x = y < 1 < z),\\ (x = y = 1, z = 0),\ (0 < z < 1 = x = y),\ (x = y = 1 < z).\end{array}$$

Symmetrically, there are 11 clock-regions where only the clocks x and z are equal, and 11 clock-regions where only the clocks y and z are equal.

There are 19 clock-regions where there are no equalities among clock variables, and x is the smallest clock:

$$\begin{array}{l}(x = 0 < y < z < 1),\ (x = 0 < y < 1 = z),\ (x = 0 < y < 1 < z),\\ (x = 0 < z < y < 1),\ (x = 0 < z < 1 = y),\ (x = 0 < z < 1 < y),\\ (x = 0, y = 1 < z),\ (x = 0, z = 1 < y),\ (x = 0, y > 1, z > 1),\\ (0 < x < y < z < 1),\ (0 < x < y < 1 = z),\ (0 < x < y < 1 < z),\\ (0 < x < 1 = y < z),\ (0 < x < z < y < 1),\ (0 < x < z < 1 = y),\\ (0 < x < z < 1 < y),\ (0 < x < 1 = z < y),\\ (0 < x < 1, y > 1, z > 1),\ (x = 1, y > 1, z > 1).\end{array}$$

Symmetrically, there are 19 clock-regions without any equalities with y as the smallest clock, and 19 clock-regions without any equalities with z as the smallest clock.

Finally, we have the region $(x > 1, y > 1, z > 1)$. Thus, there are a total of 94 clock-regions. ■

Solution 7.13: Consider a state s of a timed automaton and a clock x. Let k_x be the constant used in the definition of region equivalence for the clock x. Let us define a number $p(s, x)$ as follows: if $s(x) > k_x$ then $p(s, x) = 0$; if $s(x) = k_x$ then $p(s, x) = 1$; if $k_x - 1 < s(x) < k_x$ then $p(s, x) = 2$; if $s(x) = k_x - 1$ then $p(s, x) = 3$; and so on. That is, if $s(x) = m$ for an integer $m \leq k_x$, then $p(s, x) = 2(k_x - m) + 1$, and if $m < s_x < m + 1$ for an integer $m \leq k_x$, then $p(s, x) = 2(k_x - m)$. In particular, if $s(x) = 0$ then $p(s, x) = 2k_x + 1$. Let $p(s)$ be the sum of $p(s, x)$ over all clocks x.

We show that every timed action can be split into a sequence of at most $p(s)$ timed actions of desired form. More precisely, consider the claim: For all states s, for all durations δ, there exist $s = s_0, s_1, \ldots s_n = s + \delta$ and delays $\delta_1, \ldots \delta_n$ with $\delta_1 + \cdots + \delta_n = \delta$ and $n \leq p(s)$ such that for each i, $s_i = s_{i-1} + \delta_i$, and for every $0 \leq \epsilon \leq \delta_i$, the state $s_{i-1} + \epsilon$ is region-equivalent to either s_{i-1} or s_i. The proof is by induction on the quantity $p(s)$.

Suppose $p(s) = 0$. Then in state s every clock x already exceeds k_x. Then for every δ, the states s and $s + \delta$ are region-equivalent. Thus the desired claim holds without any splitting, that is, with $n = p(s) = 0$.

Suppose $p(s) = n$. There are two cases to consider. Suppose there exists a clock x such that $s(x)$ is an integer $m \leq k_x$. Then, letting any non-zero amount of time elapse causes the state to be non-equivalent to s. Consider a timed action of duration δ. Let δ' be a value that is smaller than δ and smaller than the fractional part of every $s(y)$ that is not an integer. The timed action of duration δ can be split in two parts: from s to $s' = s+\delta'$ of duration δ' and from s' to $s+\delta$ of duration $\delta - \delta'$. Note that $p(s', x) = p(s, x) - 1$ and $p(s', y) \leq p(s, y)$ for every clock $y \neq x$. Thus, $p(s') < p(s)$. By induction hypothesis the timed action of duration $\delta - \delta'$ from state s' to $s + \delta$ can be split in the desired way: there exist states $s' = s_0$, s_1, $\ldots$ $s_n = s+\delta$ and delays $\delta_1, \ldots \delta_n$ with $\delta_1 + \cdots + \delta_n = \delta - \delta'$ and $n \leq p(s')$ such that for each i, $s_i = s_{i-1} + \delta_i$, and for every $0 \leq \epsilon \leq \delta_i$, the state $s_{i-1} + \epsilon$ is region-equivalent to either s_{i-1} or s_i. By the choice of δ', for all $0 < \epsilon \leq \delta'$, the state $s + \epsilon$ is region-equivalent to s'. This implies that the sequence s, $s_0\, s_1$, $\ldots$ s_n and delays $\delta', \delta_1, \ldots \delta_n$ give the desired splitting of timed action of duration δ from state s (note that $n + 1 \leq p(s)$).

Suppose $p(s) > 0$ and there is no clock x such that $s(x)$ is an integer $\leq k_x$. Then let x be the clock such that $s(x)$ has the highest fractional part and $s(x) < k_x$. Then as time elapses, x will be the first clock that becomes an integer and causes a change in region. Let δ' be the increment that causes $s(x)$ to become this integer. Let $s' = s+\delta'$. Again, $p(s', x) = p(s, x) - 1$ and $p(s', y) \leq p(s, y)$ for every clock $y \neq x$. Thus, $p(s') < p(s)$. Furthermore, for all $0 \leq \epsilon < \delta'$, the state $s + \epsilon$ is region-equivalent to s itself. Now we can proceed in a manner analogous to the previous case by using splitting of the timed action of duration $\delta - \delta'$ from state s' to $s + \delta$ into $p(s')$ parts as allowed by the inductive hypothesis.

Since $p(s, x) \leq 2k_x + 1$, the desired bound b is the summation of $2k_x + 1$ over all clocks x. ■

Solution 7.14: To handle updates of the form $\mathrm{x} := d$, we do not need any change in the definition of the region equivalence. Whenever two states s and t are region-equivalent, we want to make sure that every type of action from state s can be matched by an analogous action from state t leading to equivalent states. Allowing updates of the form $\mathrm{x} := d$ does not influence enabledness of actions, and thus, the proof continues to hold.

Now suppose we have tests of the form $x - y \leq k$ in addition to tests of the form $\mathrm{x} \leq k$ and $\mathrm{x} \geq k$. Constants appearing in such constraints also contribute to the maximum constant that we use for partitioning. More precisely, for each clock variable x, k_x is now the largest integer constant appearing in a constraint of the form $\mathrm{x} \leq k$, $\mathrm{x} \geq k$, $\mathrm{x} - y \leq k$ or $y - \mathrm{x} \leq k$ in the description of the timed automaton. When a clock x exceeds the maximum k_x, we still need to keep track of whether a constraint comparing its difference with another clock holds or not. In particular, in figure 7.24, if we extend the two diagonal lines upwards, the number of partitions remains finite, but now even when x exceeds 2 and y exceeds 1, clock-valuations belonging to the same partition agree on whether or not a constraint such as $x - y \leq 1$ holds. Formally, we modify the definition of region equivalence (page 322) so that requirement 1 stays unchanged, but requirement 2 says that: for every pair of clock variables x and y, the fractional part of $\nu(x)$ is less than or equal to the fractional part of $\nu(y)$ if and only if the fractional part of $\nu'(x)$ is less than or equal to the fractional part of $\nu'(y)$. This creates a finer partitioning, but there are still only finitely many regions. The proof of theorem 7.1 now holds even when the enabledness of an action depends on the truth of constraints of the form $x - y \leq k$. ■

Solution 7.15 : 1. The DBM corresponding to given constraints is shown below:

$$\begin{bmatrix} 0 & -3 & 0 \\ 4 & 0 & 6 \\ \infty & -1 & 0 \end{bmatrix}$$

2. The DBM is not canonical. In particular, from $x_2 - x_1 \leq -1$ and $x_1 - x_0 \leq 4$, we can conclude that $x_2 - x_0 \leq 3$ and tighten $[2, 0]$ entry to 3. The canonical DBM is given by:

$$\begin{bmatrix} 0 & -3 & 0 \\ 4 & 0 & 4 \\ 3 & -1 & 0 \end{bmatrix}$$

3. To capture the effect of elapse of time, in the DBM above, we set the $[1, 0]$ entry to ∞ and $[2, 0]$ entry to 5, and then canonicalize. The resulting DBM is:

$$\begin{bmatrix} 0 & -3 & 0 \\ 9 & 0 & 4 \\ 5 & -1 & 0 \end{bmatrix}$$

4. To capture the guard $x_1 \geq 7$, in the DBM above we set the entry $[0, 1]$ to -7. Canonicalization changes the entry $[0, 2]$ to -3 (this reflects that the guard-condition $x_1 \geq 7$, together with other constraints, implies $x_2 \geq 3$). To set x_1 to 0, we change $[0, 1]$ and $[1, 0]$ entries to 0, $[1, 2]$ and $[2, 1]$ entries to ∞, and canonicalize giving the result:

$$\begin{bmatrix} 0 & 0 & -3 \\ 0 & 0 & -3 \\ 5 & 5 & 0 \end{bmatrix}$$

■

Solution 7.16 : Upon entering the mode `A`, all clocks equal 0. Thus all entries in the DBM R_A equal 0. To capture the effect of elapse of time in mode `A`, in the matrix R_A, we set $[1, 0]$ entry to 5 due to the clock-invariant $x_1 \leq 5$, set $[2, 0]$ entry to 3 due to the clock-invariant $x_2 \leq 3$, and set $[3, 0]$ entry to ∞ due to lack of explicit upper bound on x_3. Canonicalization gives the DBM R'_A:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \end{bmatrix}$$

To intersect DBM R'_A with the guard-constraint $x_3 \geq 2$, we set the $[0, 3]$ entry to -2, and canonicalize. Then to capture the effect of resetting clock x_2 to 0, we update its lower and upper bounds and canonicalize. The resulting DBM R_B is:

$$\begin{bmatrix} 0 & -2 & 0 & -2 \\ 3 & 0 & 3 & 0 \\ 0 & -2 & 0 & -2 \\ 3 & 0 & 3 & 0 \end{bmatrix}$$

To capture the effect of elapse of time in mode `B`, in the matrix R_B, we set $[1, 0]$ entry to ∞, $[2, 0]$ entry to 2, and $[3, 0]$ entry to 6. Canonicalization gives the DBM R'_B:

$$\begin{bmatrix} 0 & -2 & 0 & -2 \\ 5 & 0 & 3 & 0 \\ 2 & -2 & 0 & -2 \\ 5 & 0 & 3 & 0 \end{bmatrix}$$

To intersect DBM R'_B with the guard-constraint $x_1 \geq 3$, we set the $[0, 1]$ entry to -3, and canonicalize. Then we update entries $[0, 3]$ and $[3, 0]$ to 0, $[1, 3]$, $[3, 1]$, $[2, 3]$ and $[3, 2]$ entries to ∞, and canonicalize to obtain the DBM R_C:

$$\begin{bmatrix} 0 & -3 & 0 & 0 \\ 5 & 0 & 3 & 5 \\ 2 & -2 & 0 & 2 \\ 0 & -3 & 0 & 0 \end{bmatrix}$$

Finally, to capture the effect of elapse of time in mode `C`, in the matrix R_C, we set $[1, 0]$ entry to 8, $[2, 0]$ entry to ∞, and $[3, 0]$ entry to ∞. Canonicalization gives the DBM R'_C:

$$\begin{bmatrix} 0 & -3 & 0 & 0 \\ 8 & 0 & 3 & 5 \\ 6 & -2 & 0 & 2 \\ 5 & -3 & 0 & 0 \end{bmatrix}$$

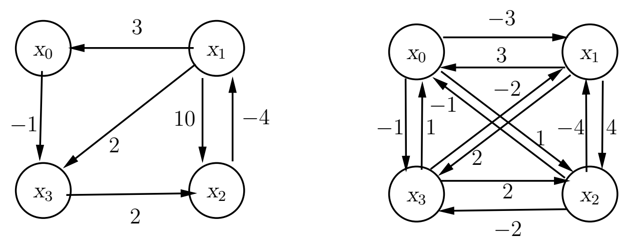
■

Solution 7.17: Note that the clock x_0 is implicitly always 0. Thus the entries in the 0th row and 0th column give bounds on the values of each clock: $x_j \leq R[j, 0]$ and $-x_j \leq R[0, j]$. When the clock x_i is reset to 0, its new value coincides with the value of x_0. For each clock x_j, we thus want the entries $R[j, i]$ and $R[i, j]$ to capture bounds on the value of clock x_j. This is achieved by executing the following code which updates the bounds in the ith row and ith column:

For $j = 0$ to m do $\{\ R[j,i] := R[j,0];\ R[i,j] := R[0,j]\ \}$.

∎

Solution 7.18: The graph representation of the constraints is shown in the figure below on left. After running the shortest-path algorithm, we get the graph on right that reflects the canonicalized version. Observe that this corresponds to the constraints $x_1 = 3$, $x_2 = -1$, and $x_3 = 1$.



∎

Solution 7.19: The set `Bounds` now contains the symbolic constant ∞ and pairs of the form (k,b), where k is an integer and $b \in \{0,1\}$. The comparison operation over integers is extended to the set `Bounds` in the following manner: for every integer k, $(k,0) < \infty$ and $(k,1) < \infty$, and $(k,0) < (k,1)$, and for two integers k and k' with $k < k'$, $(k,b) < (k',b')$ for all b and b'. Note that for any two distinct elements a and b in `Bounds`, either $a < b$ or $b < a$, and this defines the minimum: if $a < b$ then $min(a,b) = a$ else $min(a,b) = b$. Addition is defined by the following rule: for every element a in `Bounds`, $a + \infty = \infty + a = \infty$, and $(k,b) + (k',b') = (k+k', b \wedge b')$. That is, to add (k,b) and (k',b'), we add the integer parts k and k' and set the bit to 1 if both the bits equal 1. This means that combining two inequalities, one of which is a strict one, results in a strict inequality.

The given constraints are represented by the following DBM:

$$\begin{bmatrix} (0,1) & (-3,0) & (0,1) \\ (6,1) & (0,1) & (4,0) \\ \infty & (-1,1) & (0,1) \end{bmatrix}$$

This matrix is not canonical. The corresponding canonical DBM is shown below and reflects the implied constraint $x_2 \leq 5$.

$$\begin{bmatrix} (0,1) & (-3,0) & (0,1) \\ (6,1) & (0,1) & (4,0) \\ (5,1) & (-1,1) & (0,1) \end{bmatrix}$$

∎

8 Real-Time Scheduling

Solution 8.1: The first assignment statement takes c_1 time units. The WCET bound for the conditional statement "$\texttt{if}\ y > 1\ \texttt{then}\ y := z$" is $c_1 + c_2$, and for the sequence of assignments "$y := 0;\ z := x+1$" is $2\,c_1$. As a result, the WCET bound for the conditional statement "$\texttt{if}\ (x > z)\ \texttt{then}\ \{\,\texttt{if}\ y > 1\ \texttt{then}\ y := z\}\ \texttt{else}\ \{y := 0;\ z := x+1\}$" is the sum of c_2 and the maximum of $c_1 + c_2$ and $2\,c_1$, which equals $c_1 + c_2 + \texttt{max}\,\{c_1, c_2\}$. Thus, the WCET bound for the entire code fragment is $2\,c_1 + c_2 + \texttt{max}\,\{c_1, c_2\}$. ■

Solution 8.2: The utilization is $2/5 + 4/7$, which equals $34/35$. Consider the periodic schedule with period 35 where the sequence of jobs for the first 35 time slots is as follows (the 35th slot is idle):

$$J_1, J_1, J_2, J_2, J_2, J_2, J_1, J_1, J_2, J_2, J_2, J_2, J_1, J_1, J_2, J_2, J_2, J_2,$$
$$J_1, J_1, J_1, J_1, J_2, J_2, J_2, J_2, J_1, J_1, J_2, J_2, J_2, J_2, J_1, J_1, \bot\ .$$

This schedule meets all the deadlines. ■

Solution 8.3: The following periodic schedule with period 15 meets all the deadlines:

$$J_1, J_2, J_2, J_2, J_1, J_2, J_1, J_2, J_2, J_1, J_2, J_2, J_2, J_1, \bot\ .$$

The second instance of the job J_2 is preempted once, and this is the best one can do since there is no non-preemptive schedule.

To establish that there is no deadline-compliant non-preemptive schedule, let us try to construct such a schedule σ starting at the beginning. If we schedule the job J_2 at the beginning then allocating three consecutive slots to it will cause the first instance of the job J_1 to miss its deadline. Thus, $\sigma(0) = J_1$. Now we must schedule J_2 for the three consecutive slots: $\sigma(1) = \sigma(2) = \sigma(3) = J_2$ (note that leaving slots idle and waiting for the next instance of J_1 to arrive will cause the first instance of J_2 to miss its deadline). Now $\sigma(4)$ must be J_1, any other choice causes the second instance of the job J_1 to miss its deadline. If we choose to schedule J_2 in the next slot (that is, $\sigma(5) = J_2$) then to be non-preemptive $\sigma(6)$ and $\sigma(7)$ also must equal J_2 which causes the third instance of J_1 to miss its deadline. Alternatively suppose $\sigma(5) = \bot$ (note that the job J_1 is not ready at this point). Then $\sigma(6)$ must be J_1, and the next three slots must be allocated to J_2. Now $\sigma(10)$ must be J_1. Choosing $\sigma(11)$ to be J_2 requires us to allocate the next two slots also to J_2 which causes the fifth instance of J_1 to miss its deadline, and choosing $\sigma(11)$ to $\bot$ causes the third instance of J_2 to miss its deadline. ■

Solution 8.4: Let us call a sequence of jobs $J_1, J_2, \ldots J_k$ a *chain* if there is a precedence edge from each job in this sequence to the next one, that is, $J_i \prec J_{i+1}$ for $1 \le i < k$. We claim that the set $\mathcal{J}$ of jobs is schedulable exactly when $c\,k \le p$, where k is the length of the longest chain.

Let us consider how the first instances of all the jobs get scheduled during the initial period p. If $J_1, J_2, \ldots J_k$ is a chain, then due to the precedence constraints, no matter which processors are allocated to which jobs, each job J_i in this chain cannot start executing before the job J_{i-1} finishes. Since each instance takes c time units, it follows that the earliest time that the job J_i can finish is $i\,c$. Hence, if there is a chain of length k with $c\,k > p$, then the last job in this chain will miss its deadline. Thus the following condition is necessary for schedulability: $c\,k \leq p$, where k is the length of the longest chain.

Now suppose that $c\,k \leq p$, where k is the length of the longest chain. We construct a periodic schedule with period p as follows. The first instance of a job J is scheduled at time $c\,(k-1)$, where k is the length of the longest chain in which J is the last job. In particular, a job that has no incident precedence edges is scheduled at time 0. When this rule requires us to start multiple jobs at the same time, we can do so using multiple processors (note that since there are n processors, potentially all jobs can be executed in parallel). Observe that when a job is scheduled at time $c\,(k-1)$, it finishes at time $c\,k$, which is guaranteed not to exceed the period p by assumption. Notice also that this scheduling policy obeys all precedence constraints: if $J_1 \prec J_2$, then the length of the longest chain ending at job J_2 must be at least one greater than the length of the longest chain ending at J_1, thus ensuring that job J_2 gets scheduled only after job J_1 finishes. ■

Solution 8.5: Compared to the schedule shown in figure 8.3, now at time 6, since the deadlines for the current instance of both J_1 and J_2 equal 9, the scheduler chooses job J_2 instead of J_1, and then at times 7 and 8 chooses job J_1. Thus, the desired periodic schedule for the first 15 slots is given by:

$$J_2, J_1, J_1, J_1, J_2, J_1, J_2, J_1, J_1, J_2, J_1, J_1, J_1, J_2, \perp .$$

■

Solution 8.6: The deadline-compliant periodic schedule with period 24 is shown below:

$$\begin{gathered} J_2, J_2, J_1, J_1, J_3, J_3, J_3, J_3, J_1, J_1, J_2, J_2, \\ J_1, J_1, J_3, J_3, J_2, J_2, J_3, J_3, J_1, J_1, \perp, \perp . \end{gathered}$$

■

Solution 8.7: We know that $\sigma(t_1) \neq \sigma_1(t_1)$. The case when $\sigma(t_1) = J$ and $\sigma_1(t_1) = K$ is already discussed in detail in the proof. The remaining cases are considered below.

Suppose $\sigma(t_1) = \perp$ and $\sigma_1(t_1) = J$. Since the schedule σ is constructed according to the EDF policy, $\sigma(t_1) = \perp$ implies that there is no ready job at time t_1. In particular, the instance of the job J active at time t_1 has already been allocated enough time slots. Define the schedule σ_2 such that $\sigma_2(t) = \sigma_1(t)$ for all time slots $t \neq t_1$ and $\sigma_2(t_1) = \perp = \sigma(t_1)$. Since the allocation of the slot at time t_1 to job J by the schedule σ_1 is unnecessary and σ_1 is deadline-compliant, it follows that the schedule σ_2 is also deadline-compliant. Furthermore, $\mathit{diff}(\sigma, \sigma_2) > t_1$ leading to the desired contradiction.

Suppose $\sigma(t_1) = J$ and $\sigma_1(t_1) = \perp$. In this case, the job J is ready at time t_1 and yet the deadline-compliant schedule σ_1 leaves the corresponding time slot idle. Again, define the schedule σ_2 such that $\sigma_2(t) = \sigma_1(t)$ for all time slots $t \neq t_1$ and $\sigma_2(t_1) = J = \sigma(t_1)$. Since the schedule σ_1 is already deadline-compliant, no job instance can miss its deadline according to the schedule σ_2. Furthermore, $\mathit{diff}(\sigma, \sigma_2) > t_1$.

Finally, suppose $\sigma(t_1) = J$ and $\sigma_1(t_1) = K$, such that the job K is not ready at time t_1. Define the schedule σ_2 such that $\sigma_2(t) = \sigma_1(t)$ for all time slots $t \neq t_1$ and $\sigma_2(t_1) = J = \sigma(t_1)$. Since the allocation of the slot at time t_1 to job K by the schedule σ_1 is unnecessary and σ_1 is deadline-compliant, it follows that the schedule σ_2 is also deadline-compliant. As in previous cases, $\mathit{diff}(\sigma, \sigma_2) > t_1$ leading to the desired contradiction. ■

Solution 8.8: Consider a periodic job model with two jobs: the job J_1 has period 2, deadline 2, and WCET 1; and the job J_2 has period 5, deadline 5, and WCET 2. The EDF scheduling policy constructs a periodic schedule with period 10 where the following sequence repeats: $J_1, J_2, J_1, J_2, J_1, J_2, J_1, J_2, J_1, \perp$. This schedule is deadline-compliant, but has preemptions (in fact, every instance of the job J_2 get preempted). However, a non-preemptive deadline-compliant schedule does exist: $J_1, J_2, J_2, J_1, J_1, J_2, J_2, J_1, J_1, \perp$. ■

Solution 8.9: The deadline-monotonic scheduling policy causes missed deadlines. According to the deadline-monotonic scheduling policy, the job J_2 has the highest priority, the job J_1 has the next priority, and the job J_3 has the lowest priority. As a result, the first two time slots are allocated to the job J_2, the next two slots are allocated to the job J_1, and the next two slots are allocated to the job J_3. At time 6, the second instance of job J_1 arrives, and since it has a higher priority than job J_3, the next two time slots are allocated to this instance, causing a missed deadline for the first instance of job J_3. ■

Solution 8.10: Consider a periodic job model with two jobs: the job J_1 has period 4, deadline 1, and WCET 1; and the job J_2 has period 2, deadline 2, and WCET 1. The rate-monotonic policy assigns a higher priority to job J_2. According to this policy, the first time-slot is allocated to job J_2 which results in a missed deadline for the first instance of job J_1. The deadline-monotonic policy assigns a higher priority to job J_1. The resulting periodic schedule of period 4 is $J_1, J_2, J_2, \perp$ and is deadline-compliant. ■

Solution 8.11: Consider a job J_b with $b \neq a$. Recall that D_b is the time by which the first instance of the job J_b finishes its execution in the deadline-compliant schedule σ according to the original priority assignment ρ. That is, $\sigma(D_b - 1) = J_b$ and $\sigma(0, D_b, J_b) = \eta(J_b)$. Since this is a deadline-compliant schedule, we know that $D_b \leq \delta(J_b)$ holds. We prove that the schedule σ_1 also allocates $\eta(J_b)$ time slots to the job J_b in the time interval $[0, D_b]$.

Consider a job J_c and suppose $\rho_1(J_c) > \rho_1(J_b)$. Since $b \neq a$, it follows that the job J_c has a higher priority than the job J_b in the original priority assignment ρ also. Then we can proceed exactly as in the first case of the proof of the claim on page 369: if m is the number of instances of job J_c that overlap with the interval $[0, D_b]$, then the schedule σ allocates $m \cdot \eta(J_c)$ number of slots to the job J_c in the interval $[0, D_b]$, which is the maximum number of slots that can possibly be allocated to job J_c by any schedule during this interval.

We have established that, for every job J_c such that the priority assignment ρ_1 assigns a higher priority to job J_c than to job J_b, the schedule σ_1 does not allocate more slots to the job J_c during the interval $[0, D_b]$ than the schedule σ does. The claim follows. ■

Solution 8.12: For the given job model, $n = 3$ and its utilization is $1/4+1/3+3/8$, which equals $23/24 = 0.958$. Since the utilization exceeds $3(2^{1/3}-1) = 0.78$, the schedulability test of theorem 8.6 does not guarantee schedulability. The ordering of the three jobs according to the rate-monotonic priority assignment in a decreasing order of priorities is J_1, J_2, J_3. The corresponding schedule for the first eight time slots is: $J_1, J_2, J_2, J_3, J_1, J_3, J_2, J_3$. Observe that the first instances of all the three jobs meet their respective deadlines according to this schedule. By theorem 8.4, this suffices to conclude that the rate-monotonic policy results in a deadline-compliant schedule. ■

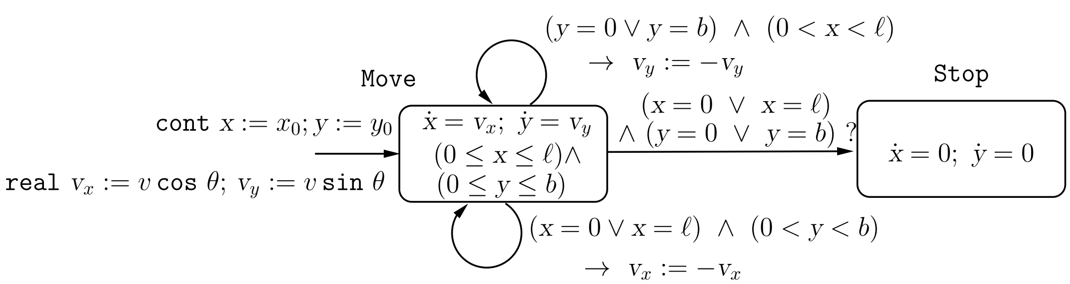
Solution 8.13: Let σ be the fixed-priority schedule for the job model $\mathcal{J}$ with respect to the priority assignment ρ. From theorem 8.4, we know that the schedule σ is deadline-compliant if and only if the deadline of the first instance of each job is met. Consider a job J and let $\delta(J) = D$. From the definition of a fixed-priority schedule, the schedule σ does not allocate a slot to a job with a priority lower than $\rho(J)$ unless the active instance of the job J has been allocated enough slots. If N is the number of first D slots that are allocated to a job with a priority higher than J, then $D - N$ time slots are available for scheduling the first instance of the job J till its deadline expires. It follows that if $D \geq \eta(J) + N$, then the deadline of the first instance is guaranteed to be met.

To get a bound on the value of N, consider a job K such that $\rho(K) > \rho(J)$. Instances of the job K arrive every $\pi(K)$ time units. As a result exactly $\lceil D/\pi(K) \rceil$ distinct instances of the job K are active during the first D slots. Each such instance is allocated at most $\eta(K)$ number of time slots by the schedule σ. Thus, for each job K such that $\rho(K) > \rho(J)$, the schedule σ assigns at most $\lceil D/\pi(K) \rceil \cdot \eta(K)$ number of time slots during the first D slots. The sum of these numbers is thus an upper bound for the number N. The claim follows. ■

9 Hybrid Systems

Solution 9.1: The hybrid process `BouncingBall` is formally specified using the following components. It has no input variables. It has two state variables h and v of type `cont` (note that since the state machine has a single mode, it need not be maintained as a state variable). It has a discrete output variable *bump* of type `event` and a continuously updated output variable h of type `cont`. Initially the state variable h equals h_0 and v equals v_0. It has no input or internal tasks. There is a single output task associated with the output channel *bump* with the guard condition $(h = 0)$ and update code $bump\,!;\ v := -a\,v$. The expression associated with the output variable h equals the corresponding state variable h. The expression for the rate of change of the state variable h equals v, and for the rate of change of the state variable v equals $-g$. The continuous-time invariant CI is given by the expression $h \geq 0$. ■

Solution 9.2: The hybrid process is shown below. It maintains continuously updated state variables x and y tracking the position of the ball, and discrete real-valued variables v_x and v_y corresponding to the components of the velocity of the ball along the two axes. The process starts in the mode `Move`, where the rates of change of the variables x and y are given by the velocities v_x and v_y respectively. The continuous-time invariant $(0 \leq x \leq \ell) \wedge (0 \leq y \leq b)$ ensures that a discrete transition is enforced when the ball position is at the edge of the table. When x equals 0, if y is either 0 or b, then the ball is in one of the holes, and this causes the process to switch to the mode `Stop` where the variables x and y do not change. When x equals 0 and $0 < y < b$, then the ball hits the left wall, and due to this collision, the vertical velocity v_y stays unchanged but the horizontal velocity v_x flips sign. The cases corresponding to x being equal to ℓ, y being equal to 0, and y being equal to b are handled similarly.



■

Solution 9.3: The process can be in one of the following modes: `Stop`, `Straight`, `Right`, `Left`, `Up`, and `Down`. The mode indicates the direction of motion. The initial mode is `Stop`. The process has two continuously-updated state variables x and y modeling the position of the robot, and two (non-negative) real-valued discrete variables x_t and y_t capturing the position of the current target. Initially, $x = 0$ and $y = 0$. There is a single input channel *in* of type $\texttt{real}_{\geq 0} \times \texttt{real}_{\geq 0}$.

The dynamics in the initial mode `Stop` is $\dot{x} = 0$ and $\dot{y} = 0$, and this mode has no continuous-time invariant associated with it. The mode-switch out of `Stop` is triggered by an input event. The input values are used to set the target location. The process compares time it takes to reach the target if the robot travels straight and the time it takes to reach the target traveling first horizontally and then vertically. If the former is smaller, then the process switches to the mode `Straight`, and otherwise, to the mode either `Right` or `Left` depending on whether the target is to the right or left of the current position. Let

$$\varphi \ = \ (|x_t - x|/6 + |y_t - y|/8) \leq \sqrt{(x_t - x)^2 + (y_t - y)^2}/5$$

be the formula that holds when it is preferable to travel zig-zag than moving straight. Then the code for the mode-switch is described by the following:

$(x_t, y_t) \ := \ in;$
`if` φ `then if` $(x < x_t)$ `then` $mode :=$ `Right else` $mode :=$ `Left`
`else` $mode :=$ `Straight`.

In the mode `Straight`, the dynamics is given by

$$\dot{x} = 5 \cdot (x_t - x)/\sqrt{(x_t - x)^2 + (y_t - y)^2}; \ \dot{y} = 5 \cdot (y_t - y)/\sqrt{(x_t - x)^2 + (y_t - y)^2}.$$

The continuous-time invariant ensures that the robot is not yet at the target, and the preferred option is still to move straight: $(x, y) \neq (x_t, y_t) \wedge \neg\,\varphi$. There is a mode-switch from `Straight` to `Stop` with the guard condition $(x, y) = (x_t, y_t)$, to `Right` with the guard condition $\varphi \wedge (x < x_t)$, and to `Left` with the guard condition $\varphi \wedge (x \geq x_t)$.

In the mode `Right`, the dynamics is given by $\dot{x} = 6$ and $\dot{y} = 0$. The continuous-time invariant ensures that there is still horizontal distance to be covered, and the preferred option is still to move zig-zag: $(x \neq x_t) \wedge \varphi$. There is a mode-switch from `Right` to `Straight` with the guard condition $\neg\,\varphi$, to `Up` with the guard condition $(x = x_t) \wedge (y < y_t)$, and to `Down` with the guard condition $(x = x_t) \wedge (y > y_t)$. The mode `Left` is similar.

Finally, in the mode `Up`, the dynamics is given by $\dot{x} = 0$ and $\dot{y} = 8$. Once in this mode, the robot can keep moving vertically till it reaches the target. The continuous-time invariant then is the condition $(y < y_t)$, and there is a mode-switch to `Stop` with the guard condition $(y = y_t)$. The mode `Down` is symmetric. ■

Solution 9.5: We model the behavior by a hybrid process with three modes: `West` (the bee is moving west), `East` (the bee is moving east), and `Crash` (the trains have collided). There are three state variables b, w, and e modeling the positions of the bee B, the train W, and the train E, respectively. Let us assume that the position increases for objects moving west.

Initially the mode is `West`, and the initialization of the state variables is given by $b := b_0$; $w := w_0$; $e := e_0$, where b_0, w_0, and e_0 give the initial positions of the bee B, the train W, and the train E, respectively. We assume that $w_0 \leq b_0 \leq e_0$. We also assume that the bee travels faster than the trains: $v_b > v_e$ and $v_b > v_w$.

In the mode `Crash`, the dynamics is given by $\dot{b} = \dot{w} = \dot{e} = 0$, and this mode has no explicit continuous-time invariant associated with it.

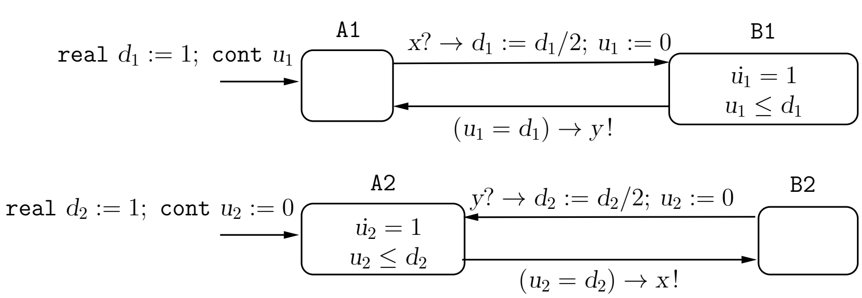
In the mode `West`, the dynamics is given by $\dot{b} = v_b$ and $\dot{w} = v_w$ and $\dot{e} = -v_e$. The continuous-time invariant is $(e \neq w) \wedge (b \leq e)$, which states that the trains have not yet collided and the bee has not yet reached the train E. There is a mode-switch from `West` to `Crash` with the guard $(w = e)$ and to `East` with the guard $(b = e)$.

The mode `East` is symmetric. The dynamics is given by $\dot{b} = -v_b$ and $\dot{w} = v_w$ and $\dot{e} = -v_e$. The continuous-time invariant is $(e \neq w) \wedge (b \geq w)$. There is a mode-switch from `East` to `Crash` with the guard $(w = e)$ and to `West` with the guard $(b = w)$.

The distance between the trains decreases independent of the motion of the bee at the constant rate $v_w + v_e$. Thus, the distance at time t is $(e_0 - w_0) - (v_w + v_e)t$.

If the bee switches to the mode `West` at time t with the two trains d apart, then it can stay in the mode `West` for a duration of $d/(v_b + v_e)$ time units, and switch to the mode `East` at time $t + d/(v_b + v_e)$. Similarly, if the bee switches to the mode `East` at time t with the two trains d apart, then it can stay in the mode `East` for a duration of $d/(v_b + v_w)$ time units, and switch to the mode `West` at time $t + d/(v_b + v_w)$. Let $d_0 = (e_0 + w_0)$ be the initial distance between the trains, and let $t_0 = 0$ be the initial time. Then, we have $t_1 = t_0 + d_0/(v_b + v_e)$ and $d_1 = d_0(1 - (v_w + v_e)/(v_b + v_e))$; $t_2 = t_1 + d_1/(v_b + v_w)$ and $d_2 = d_1(1 - (v_w + v_e)/(v_b + v_w))$; $t_3 = t_2 + d_2/(v_b + v_e)$ and $d_3 = d_2(1 - (v_w + v_e)/(v_b + v_e))$; and so on. Thus, the sequence $d_0, d_1, d_2, d_3, \ldots$ converges to 0 but with each $d_i > 0$. As a result, the process is forced to switch between the modes `West` and `East` with a converging sequence of times $t_0, t_1, t_2, \ldots$ of mode-switches. Thus the process is a Zeno process. The limit of the sequence $t_0, t_1, t_2, \ldots$ is strictly less than the time $t^* = d_0/(v_e + v_w)$ when the trains will collide. ∎

Solution 9.6:



The top state machine models the hybrid process HP_1. It has two modes, a clock variable u_1, and a discretely updated delay variable d_1. Initially d_1 is 1. Every time the process receives an input event $x\,?$, it halves the value of d_1, and then waits for exactly d_1 time units in the mode `B1` before transmitting the output event $y\,!$ and returning to the initial mode. Observe that the process can wait in the mode `A1` for an arbitrary amount of time. Thus, it is always possible to produce an execution along which time diverges, and the process is non-Zeno.

The bottom state machine captures the hybrid process HP_2. It has two modes, a clock variable u_2, and a discretely updated delay variable d_2 initialized to 1. The process starts in the mode `A2` where it waits for exactly d_2 time units before transmitting the output event $x\,!$ and switching to the mode `B2` where it waits to receive an input. Every time the process receives the input event $y\,?$, it halves the value of d_2, and returns to the initial mode to wait before issuing the next output. The process can wait in the mode `B2` for an arbitrary amount of time, and thus, the process is non-Zeno.

In the composed process, the first output event is y at time 1, the second event is x at time 1.5, the third event is y at time 2, the fourth event is x at time 2.25, the fifth event is y at time 2.5, and so on. The sequence of times converges, and in particular, is bounded by 3. The composite process is forced to produce an infinite number of output events in a bounded time, and thus, is Zeno. ■

Solution 9.11: The temperature changes in the mode `off` in the same manner as in case of the model of figure 9.1, namely, it decreases at a constant rate of k_2. When the temperature is in the range $[60, 62]$, the process switches to the mode `on1`, where the temperature increases at a constant rate in the range $[3k_1, 10k_1]$. When it reaches 67, the process switches to the mode `on2`, where the temperature increases at a constant rate in the range $(0, 3k_1]$. The process switches back to the initial mode `off` when the temperature is in the range $[68, 70]$. Compared to the original model where the temperature increases at an exponential rate in the mode `on`, the evolution is now approximated by piecewise linear functions with bounds on slopes of the two pieces. Observe that the set of possible values of the temperature is the same for the two models. However, the possible total time that the modified process can spend in the modes `on1` and `on2` together is an over-approximation (that is, a superset) of the possible durations for which the original process can be in the mode `on`. In particular, assuming that the temperature T^* when leaving the mode `off` does not exceed 67, the minimum duration spent by the original process in the mode `on` is $-\texttt{ln}\ (2/(70 - T^*))/k_1$ seconds, and the minimum duration spent by the approximate model in the two modes `on1` and `on2` before switching back to `off` is $(1/3 + (67 - T^*)/10)/k_1$ seconds. The latter value is strictly smaller than the former: for example, for $k_1 = 1$ and $T^* = 62$, the former bound is 1.386 while the latter is 0.833. ■

Solution 9.12: To figure out a better strategy, first let us consider the case when $p < e$. If the evader keeps on moving clockwise, then it takes the evader

$(40-e)/5$ seconds to reach the rescue car. Moving in the counterclockwise direction, it takes the pursuer $2p$ seconds to reach the car. If $2p > (40-e)/5$, then the optimal strategy for the evader is to move clockwise: the pursuer cannot capture the evader by moving counterclockwise, and if the pursuer follows the evader also moving clockwise, then clearly moving counterclockwise could not have been a better choice for the evader. Now suppose $2p \leq (40-e)/5$. Committing to always move clockwise is not a good strategy for the evader since that will guarantee capture. If the evader moves clockwise for the next two seconds, then after two seconds, the evader will be at position $e+10$. In this case, the pursuer can get closest to the evader by moving counterclockwise, and in such a case, the pursuer will be at position $p-1$ after two seconds. The separation between the two then will be $(p-1)+40-(e+10) = 29+p-e$. Analogously, if the evader moves counterclockwise for the next two seconds, then after two seconds, the evader will be at position $e-10$, and the closest the pursuer can be is at position $p+12$ (this corresponds to the pursuer moving clockwise at full speed). The minimum separation after two seconds then will be $(e-10)-(p+12) = e-p-22$. To decide whether to move clockwise or counterclockwise for the next two seconds the evader should then compare the minimum distance from the pursuer at the end of this interval: if $29+p-e > e-p-22$ which simplifies to $e-p < 25.5$, then move clockwise, else move counterclockwise. Note that after two seconds, depending on the actual position of the pursuer, the evader compares these quantities again, and may end up reversing the direction.

The analysis for the case $e < p$ is analogous. In this case, for the evader, heading away from the pursuer means moving counterclockwise, and in this case it takes $e/5$ seconds to reach the rescue car. The pursuer can reach the car in $(40-p)/6$ seconds by moving clockwise. If $e/5 < (40-p)/6$, that is, $6e+5p < 200$, then committing to moving counterclockwise is the best strategy for the evader. Otherwise, the evader should compare the minimal distance from the pursuer if the decision is to move clockwise or counterclockwise for the next two seconds. If the evader moves clockwise, then after two seconds the minimum separation between the two is $(p-1)-(e+10) = p-e-11$. If the evader moves counterclockwise, then after two seconds the minimum separation between the two is $(e-10)+40-(p+12) = e-p+18$. Thus, if $p-e-11 > e-p+18$, that is, $p-e > 14.5$, then the evader should move clockwise.

In summary, the evader should choose to move clockwise if the following condition is true (and move counterclockwise, otherwise):

$$[\,(p<e) \wedge (e{+}10p > 40 \vee e{-}p < 25.5)\,] \vee [\,(e<p) \wedge (6e{+}5p \geq 200) \wedge (p{-}e > 14.5)\,].$$

As discussed above, this is indeed the optimal strategy. Suppose the initial position is $e=20$ and $p=1$. In this case, the evader chooses to move clockwise for the first two seconds. At time $t=2$, the evader is at position 30 meters. If the pursuer had moved full speed clockwise, then the pursuer is at position 13 at $t=2$, and in this case, at $t=2$, the evader commits to moving clockwise, and will reach the rescue car safely at $t = 4$. If the pursuer had moved full speed counterclockwise for the first two seconds, then at $t = 2$, the pursuer is at the rescue car, and in such a case, at $t = 2$, the evader chooses to move counterclockwise for the next two seconds. The game continues with either the evader reaching the rescue car, or with the pursuer staying close to the car and the evader switching back and forth between moving clockwise and counterclockwise. ■

Solution 9.13: The operations `Conj` and `Disj` have already been defined for regions represented by formulas of type `AffForm`. We define the negation operation `Not` below, and using this operation we can define $\texttt{Diff}(A, B)$ to be $\texttt{Conj}(A, \texttt{Not}(B))$.

Following the definition of affine formulas, it has already been noted that if A is an atomic affine formula, then it is possible to define $\texttt{Not}(A)$ to be an affine formula. Now suppose A is a conjunctive affine formula $\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_k$, where each conjunct φ_i is an atomic affine formula. Then

$$\texttt{Not}(A) \;=\; \texttt{Disj}(\texttt{Not}(\varphi_1), \texttt{Disj}(\texttt{Not}(\varphi_2), \ldots, \texttt{Not}(\varphi_k) \cdots)).$$

Thus we have defined the negation operation for conjunctive affine formulas. Finally, suppose A is the disjunction $\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_l$, where each disjunct φ_j is a conjunctive affine formula. Then,

$$\texttt{Not}(A) \;=\; \texttt{Conj}(\texttt{Not}(\varphi_1), \texttt{Conj}(\texttt{Not}(\varphi_2), \ldots, \texttt{Not}(\varphi_l) \cdots)).$$

Thus, we have defined the negation operation `Not` for all affine formulas. ■

Solution 9.14: Consider a conjunctive affine formula A and a variable x. Let us first consider the case when x is a continuously updated variable. The formula A can have two types of atomic affine formulas as conjuncts. Thus A can be written as the conjunction $A_1 \wedge A_2$, where A_1 is a conjunction of atomic formulas of the form $(y = d)$, where y is a discrete variable, and A_2 is a conjunction of affine constraints of the form $(a_1x_1 + a_2x_2 + \cdots + a_nx_n \sim a_0)$. The textbook describes how to eliminate the variable x from the formula A_2. Let $A_3 = \texttt{Exists}(A_2, x)$ be the formula obtained by this procedure, and it gives a conjunctive affine formula. The constraints of the form $(y = d)$ are unaffected by the quantification of x, and thus, $\texttt{Exists}(A, x)$ can be defined to be the conjunctive affine formula $A_1 \wedge A_3$.

Now suppose A is a conjunctive affine formula and x is a discrete variable. If A has two atomic conjuncts of the form $(x = d)$ and $(x = d')$ for two distinct values d and d', then the formula A can never be satisfied, and in this case, $\texttt{Exists}(A, x)$ is the constant formula 0. If the variable x does not appear in A at all, then $\texttt{Exists}(A, x)$ equals A itself. If there is a unique value d such that $(x = d)$ is a conjunct of A, then A must be of the form $(x = d) \wedge A'$, for some conjunctive affine formula A'. In this case, $\texttt{Exists}(A, x)$ is defined to be A' since eliminating x corresponds to dropping the constraint on x.

Now suppose A is the disjunction $\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_l$, where each disjunct φ_i is a conjunctive affine formula. We already know how to apply quantifier elimination to conjunctive affine formulas φ_i to obtain another conjunctive affine formula. Then, $\texttt{Exists}(A, x)$ is defined to be the disjunction of the conjunctive affine formulas $\texttt{Exists}(\varphi_i, x)$. This is because existential quantification distributes over disjunction. ■

Solution 9.15: To quantify the variable x_1, we first find bounds on its value implied by each conjunct. The first conjunct gives the lower bound constraint $x_1 \geq 6x_4 - 17$, the second conjunct gives the strict upper bound constraint $x_1 < (1 - 12x_3)/3$, the third conjunct gives the upper bound constraint $x_1 \leq (7 + 3x_2 - 5x_4)/2$, the fourth conjunct does not constrain x_1, and the fifth conjunct gives the strict lower bound constraint $x_1 > (-5 - 2x_2 + x_3)/5$. We can eliminate x_1 by requiring every lower bound not to exceed every upper bound:

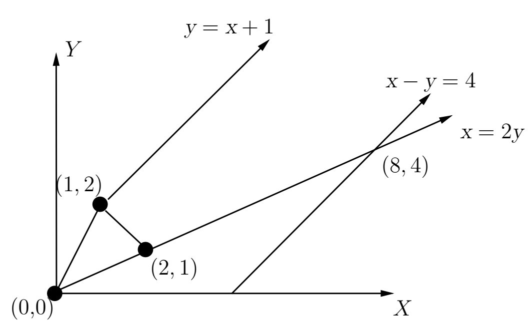
$$(6x_4 - 17) < (1 - 12x_3)/3 \ \wedge \ (6x_4 - 17) \leq (7 + 3x_2 - 5x_4)/2 \ \wedge$$
$$(-5 - 2x_2 + x_3)/5 < (1 - 12x_3)/3 \ \wedge \ (-5 - 2x_2 + x_3)/5 < (7 + 3x_2 - 5x_4)/2.$$

We can rewrite each conjunct into the desired affine form, and also retain the fourth conjunct from the original affine formula. This gives the desired region:

$$(\, 12x_3 + 18x_4 < 52 \,) \ \wedge \ (\, -3x_2 + 17x_4 \leq 41 \,) \ \wedge \ (\, -6x_2 + 63x_3 < 20 \,) \ \wedge$$
$$(\, -19x_2 + 2x_3 + 25x_4 < 45 \,) \ \wedge \ (\, 7x_2 - x_3 - 8x_4 > 0 \,).$$

■

Solution 9.16: To understand how the state of the system evolves, consider the illustration shown below. The initial region is the triangle connecting the points $(0, 0)$, $(1, 2)$, and $(2, 1)$. The rate of change of x is 1 and the rate of change of y is between 0.5 and 1. Thus, each state evolves within the cone bounded by rays of slope 1 and 0.5 starting at that state. As a result, the region describing the set of reachable states as time elapses is bounded by the ray $y = x + 1$ starting at the point $(1, 2)$ and the ray $x = 2y$. The continuous-time invariant requires that the state stays on or above the line $x - y = 4$. Thus the desired timed post-image is the shape bounded by (1) the line segment joining $(0, 0)$ and $(1, 2)$, (2) the line segment joining $(0, 0)$ and $(8, 4)$, (3) the ray $y = x + 1$ starting at the point $(1, 2)$, and (4) the ray $x - y = 4$ starting at the point $(8, 4)$.



■

Acknowledgments

Many thanks to Nimit Singhania for carefully proof-reading these solutions.