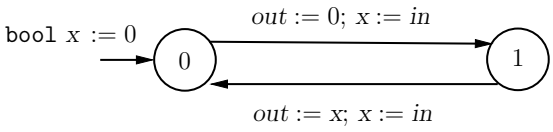
2 Synchronous Model

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:

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

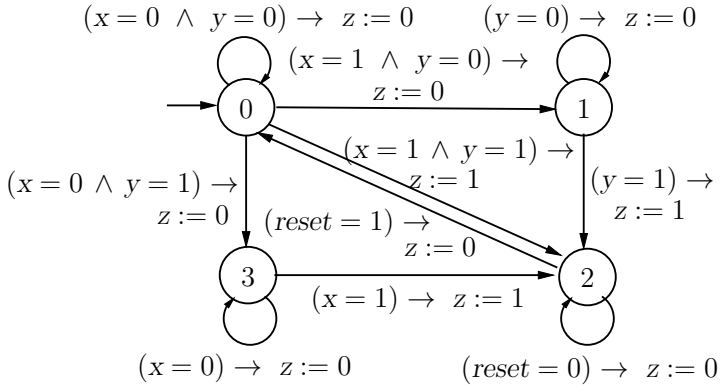
■

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.



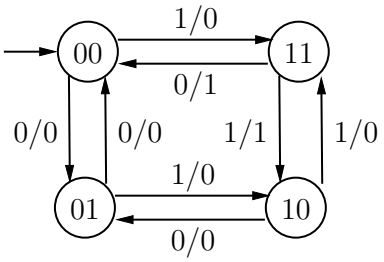
■

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 ≥ 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 }
}.
```

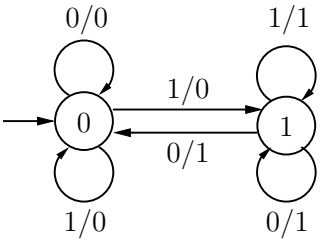
The component `SecondToMinute` is event-triggered. ■

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:



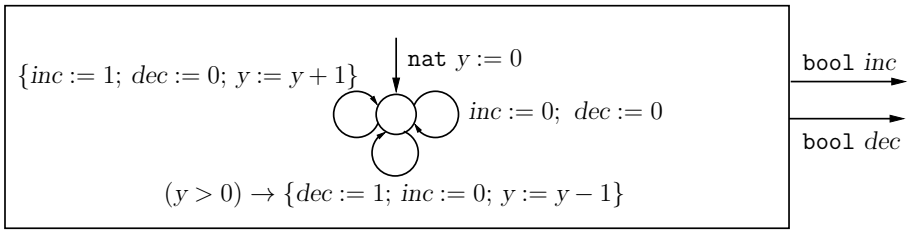
■

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!.
```

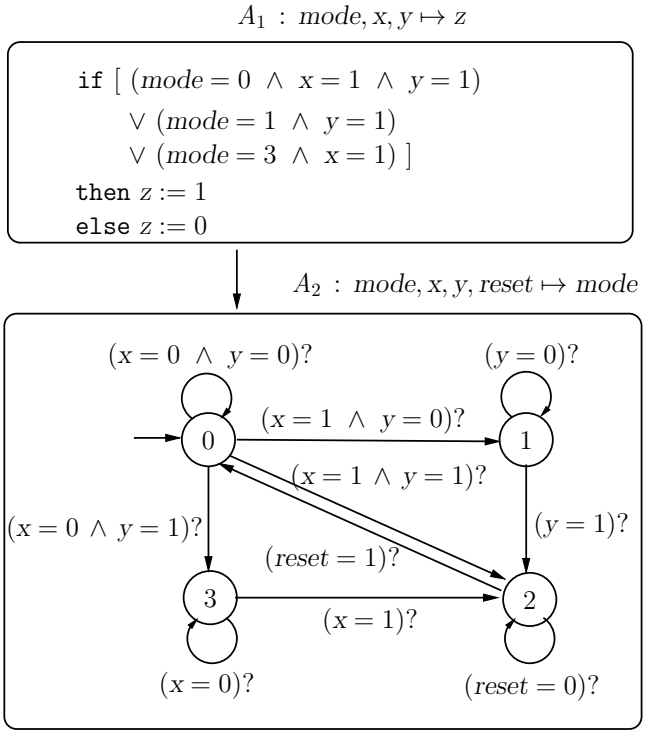
■

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.



■

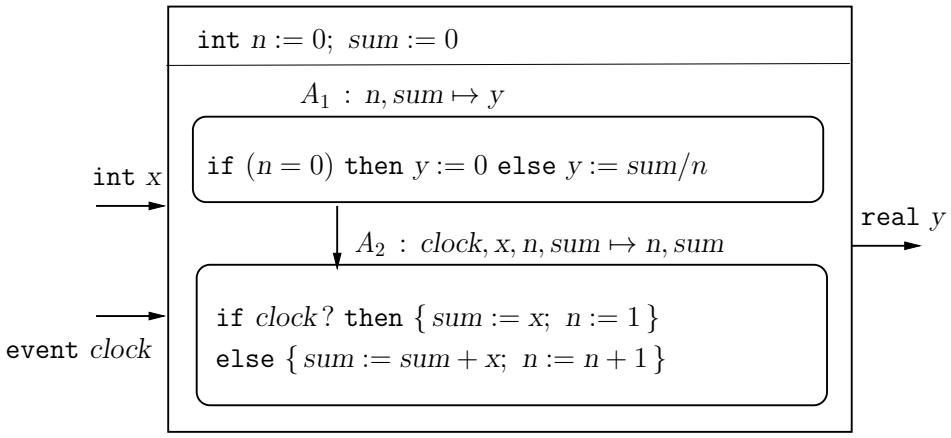
Solution 2.12: Since the task A_2 writes the output z, and z does not await the input x, we can conclude that the task A_2 does not read x and nor does a task that must precede A_2. Since the output y produced by the task A_1 awaits x, it must be the case that A_1 reads x. It follows that there cannot 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. ■

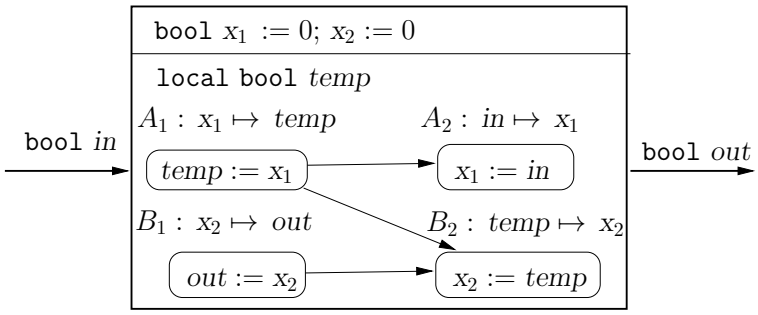
Solution 2.14: The component `ComputeAverage` is shown below. It maintains an integer state variable n that tracks the number of rounds elapsed since the presence of the input event *clock*, and an integer state variable *sum* that maintains the sum of the values of the input variable x since the presence 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.



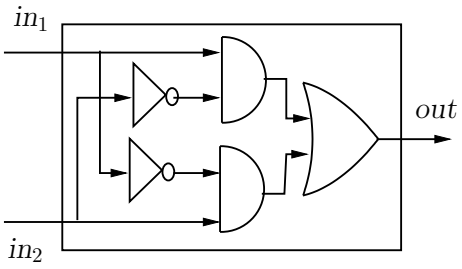
The output variable *out* does not await the input variable *in*. ■

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

(`SecondToMinute` ‖ `SecondToMinute`[*minute* ↦ *hour*][*second* ↦ *minute*])*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.