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

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

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

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

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. ■

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:

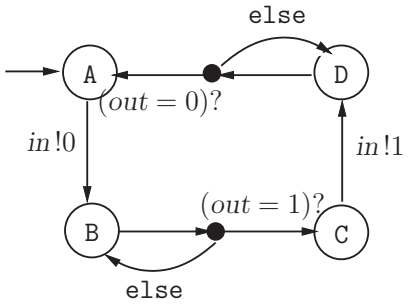
$$A^1 \ : \ \neg \texttt{Empty}(x_1) \ \wedge \ \neg \texttt{Full}(y_1) \ \rightarrow$$
$$\{ \ \texttt{local msg} \ \mathit{temp} := \texttt{Dequeue}(x_1); \ \texttt{Enqueue}(\mathit{temp}, y_1) \ \}$$
$$A^2 \ : \ \neg \texttt{Empty}(x_2) \ \wedge \ \neg \texttt{Full}(y_1) \ \rightarrow$$
$$\{ \ \texttt{local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

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*:

$$A^1_o \ : \ \neg \texttt{Empty}(y_1) \ \rightarrow \ \mathit{out} := \texttt{Dequeue}(y_1)$$
$$A^2_o \ : \ \neg \texttt{Empty}(y_2) \ \rightarrow \ \mathit{out} := \texttt{Dequeue}(y_2)$$

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 (`AsyncNot`.*mode* $\neq$ `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 $\mathit{turn} = 1$ and $\mathit{turn} = 2$. 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 $flag_1 = 1$, and this ensures that the process P_2 cannot find the condition ($flag_1 = 0$) 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

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