4

Asynchronous Model

We now shift our focus to the *asynchronous* model of computation that does not require concurrent activities to execute in lock-step. Such models naturally capture multi-processor machines and networked distributed platforms. In this chapter, we first formalize this model of computation and then study how to design protocols to achieve coordination necessary to solve computing problems in the presence of asynchrony.

4.1 Asynchronous Processes

Like a synchronous reactive component, an asynchronous process interacts with other processes via inputs and outputs and maintains an internal state. However, the execution does not proceed in rounds, and the speeds at which different processes execute are *independent*. Within a process, the reception of inputs is decoupled from the production of outputs, and this corresponds to the assumption that any internal computation takes an unknown but nonzero amount of time.

As an example, consider the `Buffer` process shown in figure 4.1 that models the asynchronous version of the synchronous reactive component `Delay` of figure 2.1. The input and output variables of a process are called *channels*. The process `Buffer` has a Boolean input channel *in* and a Boolean output channel *out*. The internal state of the process `Buffer` is a buffer of size 1, which can either be empty or contain a Boolean value. This is modeled by the variable x that ranges over the enumerated type $\{\texttt{null}, 0, 1\}$. Initially, the buffer is empty. The key difference between the synchronous component `Delay` and the asynchronous process `Buffer` lies in the specifications of their dynamics. The process `Buffer` has two possible types of actions. It can process an input value available in the input channel *in* by copying it into its buffer. Alternatively, if the buffer is non-empty, then the process can output the buffer state by writing it to the output channel *out* and then reset the buffer to empty. Each type of action is specified using a task, and *in one step, only one of the tasks is executed*.

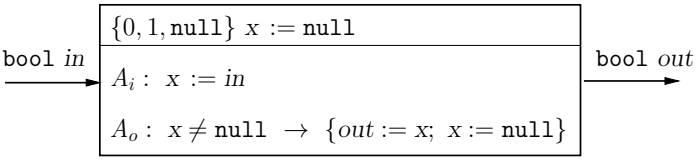

Figure 4.1: Asynchronous Process `Buffer`

4.1.1 States, Inputs, and Outputs

In general, an asynchronous process P has a set I of typed input channels, a set O of typed output channels, and a set S of typed state variables. All these three sets are finite and disjoint from one another so that there are no name conflicts.

As in the case of synchronous reactive components, a state of a process P is a valuation over the set S of its state variables, and the set of its states is the set Q_S of all possible valuations over S. The initialization *Init* assigns initial values to all the state variables in S. As before, we allow multiple initial values to capture situations where initial values are only partially known. A state q is called an initial state if, for every state variable x, the value $q(x)$ is consistent with the initialization of the variable x. The set of all initial states is denoted $[\![Init]\!]$.

In the asynchronous model of computation, when there are multiple input channels, the arrival of input values on different channels is not synchronized. Hence, an input of a process consists of a single input channel x along with a value v that belongs to the type of x. We denote such an input by $x\,?\,v$. Such an input can be interpreted as *receiving* the value v on the input channel x.

The modeling of outputs is symmetric. When there are multiple output channels, in one step, a process can produce a value for only one of the output channels. An output of a process consists of a single output channel y along with a type-consistent value v. We denote such an output by $y\,!\,v$. Such an output can be interpreted as *sending* the value v on the output channel y.

For the process `Buffer`, the set S of state variables is $\{x\}$, the set I of input variables is $\{in\}$, the set O of output variables is $\{out\}$, the set of states is $\{0, 1, \texttt{null}\}$, the set of initial states is $\{\texttt{null}\}$, the set of inputs is $\{in\,?\,0,\ in\,?\,1\}$, and the set of outputs is $\{out\,!\,0,\ out\,!\,1\}$.

4.1.2 Input, Output, and Internal Actions

For synchronous reactive components, execution during a round is specified using a set of tasks, where the execution of a single task captures an atomic unit of computation. For asynchronous processes, we also specify its computation using a set of tasks. As before, the update description of a task assigns values

to variables in its write-set using the values of variables in its read-set and is usually given as a straight-line code consisting of conditional and assignment statements. In contrast to the synchronous case, during one step, instead of executing all the tasks, only one task is executed. To indicate whether a task is ready to be executed, we explicitly associate a *guard* condition with each task. This condition is given as a Boolean formula over the state variables, and the task is enabled in a given state if the state satisfies this formula. If multiple tasks are enabled, then one of them is chosen nondeterministically for execution. Precedence constraints among tasks are no longer meaningful since there is no need to order tasks within a round. In the synchronous model, a careful specification of the subset of state variables that a task reads and writes is necessary to identify potential write-conflicts, and we require that tasks with write-conflicts are ordered by precedence constraints for scheduling within a single round. This is not relevant in the asynchronous model, and we assume that each task reads and writes all the state variables. To ensure that a process either receives a single input value or sends a single output value in a step, we require that each task can either read at most one input channel or write at most one output channel.

Input Tasks

Processing of an input is called an *input action*. During an input action, the process can only update its state and does not produce outputs. Input actions are specified using *input tasks*, each of which is associated with one input channel. The description of an input task A associated with an input channel x is given as $\mathit{Guard} \rightarrow \mathit{Update}$, where *Guard* gives the condition under which this task is willing to process inputs on the channel x and *Update* describes how the task updates state variables based on the old values of the state variables together with the input value received on the channel x. Semantically, *Guard* defines a set $[\![\mathit{Guard}]\!]$ of valuations over S, and *Update* defines a relation $[\![\mathit{Update}]\!]$ from valuations over the read-set $S \cup \{x\}$ to valuations over the write-set S. An input task A is said to be enabled in a state s if the state s satisfies the guard condition *Guard*. Such a task defines the set of input actions of the form $s \xrightarrow{x\,?\,v} t$ such that the state s satisfies the guard *Guard* and the state t can be obtained by executing the description *Update* in state s given the value v for the input channel x, that is, if $s \in [\![\mathit{Guard}]\!]$ and $(s[x \mapsto v], t) \in [\![\mathit{Update}]\!]$.

For the process `Buffer` of figure 4.1, there is a single input task A_i that reads the input channel *in*. The task is always enabled, meaning that the process is always willing to accept an input on the channel *in*. In such a case, the guard equals the Boolean constant 1 and is omitted from the description. The task updates the state variable x using the assignment $x := \mathit{in}$. This task leads to six input actions: for each state $s \in \{0, 1, \texttt{null}\}$, $s \xrightarrow{\mathit{in}\,?\,0} 0$ and $s \xrightarrow{\mathit{in}\,?\,1} 1$. Note that if the process is supplied an input value when the buffer is non-empty, then the old state is lost.

Usually each input channel x has one input task associated with it. If no input

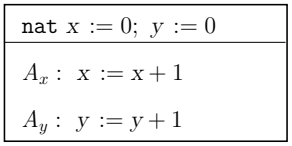

Figure 4.2: Asynchronous Process `AsyncInc`

task is associated with a channel, then the process cannot receive any inputs on this channel. We can associate multiple tasks with the same channel to specify different ways of processing input values on this channel. The set of all input tasks associated with an input channel x is denoted $\mathcal{A}_x$. In our example, $\mathcal{A}_{in} = \{A_i\}$.

Output Tasks

Producing an output is called an *output action*. Output actions are specified using *output tasks*, where each output task is associated with one output channel y. An output task A associated with an output channel y is described using a guard condition *Guard* that specifies the set of states in which the output task is ready to be executed and an update description *Update* that specifies how the task updates the state variables and the output value for y based on the values of the state variables it reads. Thus, for such a task, $[\![\mathit{Update}]\!]$ is a relation from valuations over the set S of state variables to valuations over the set $S \cup \{y\}$. Given a state s that satisfies the guard condition *Guard*, we can execute the update description *Update* to compute the new values of the state variables resulting in a state t, along with a value v to be issued on the output channel y. Thus, such a task defines the set of output actions $s \xrightarrow{y\,!\,v} t$, such that $s \in [\![\mathit{Guard}]\!]$ and $(s, t[y \mapsto v]) \in [\![\mathit{Update}]\!]$. As in the case of input tasks, multiple output tasks may be associated with the same channel, and the set of all tasks associated with an output channel y is denoted $\mathcal{A}_y$.

For the process `Buffer` of figure 4.1, there is a single output task A_o that produces an output on the channel *out*. The guard for this task is the condition $x \neq$ `null`: the output task is enabled only when the buffer contains a non-null value. The update is described by the sequence of assignments $\mathit{out} := x;\ x :=$ `null`. This leads to the following two output actions: $0 \xrightarrow{\mathit{out}\,!\,0}$ `null` and $1 \xrightarrow{\mathit{out}\,!\,1}$ `null`.

Internal Tasks

As a second example, consider the process `AsyncInc` of figure 4.2. The process does not have any input or output channels. It has two state variables x and y, both of type `nat` and initialized to 0. Since the process has no input and output channels, there are no input or output tasks.

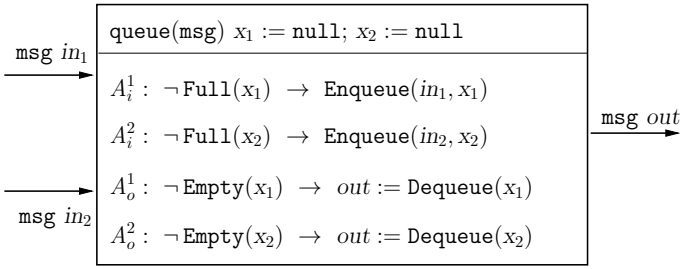

Figure 4.3: Asynchronous Process `Merge`

The internal computation of a process is described using *internal actions*. Such actions neither process inputs nor produce outputs but update internal state and are described using *internal tasks*. An internal task A has a Boolean guard condition *Guard* that describes the states in which the task is enabled and an update description *Update* that specifies how the task updates the state variables based on their old values. Given a state s, we evaluate the guard *Guard* to check whether the task is ready to be executed and, if so, execute the update description *Update* to compute the new values for the state variables leading to a state t. Thus, the task specifies the set of internal actions $s \xrightarrow{\varepsilon} t$ such that $s \in [\![Guard]\!]$ and $(s,t) \in [\![Update]\!]$. The label ε indicates that there is no observable communication during an internal action.

For the process `AsyncInc`, the state is updated by two internal tasks A_x and A_y. The task A_x is always enabled (that is, its guard condition is always satisfied) and increments the state variable x as specified by the update code $x := x + 1$. The task A_y is symmetric and increments the state variable y. The set of all internal tasks of a process is denoted $\mathcal{A}$ and equals $\{A_x, A_y\}$ for the process `AsyncInc`. A step of the process corresponds to executing one of these two tasks. Thus, the set of internal actions consists of $(i,j) \xrightarrow{\varepsilon} (i+1,j)$ and $(i,j) \xrightarrow{\varepsilon} (i,j+1)$ for every pair of natural numbers i and j.

Asynchronous Merge

As a third example, consider the process `Merge`, shown in figure 4.3, with two input channels in_1 and in_2, both of type `msg`. The process uses a buffer dedicated to each of the two input channels to store values received on that channel. We model a buffer using the type `queue`: `null` represents the empty queue, the operation $\texttt{Enqueue}(v,x)$ updates the queue x by adding the value v as its last element, the operation $\texttt{Dequeue}(x)$ returns the first element of the queue x while updating the queue by removing the first element, the operation $\texttt{Front}(x)$ returns the first element of the queue x without removing it from the queue, the operation $\texttt{Empty}(x)$ returns 1 if the queue x is empty and 0 otherwise, and the operation $\texttt{Full}(x)$ returns 1 if the queue x is full and 0 otherwise.

The input task A_i^1 captures how the values received on the input channel in_1 are processed: if the queue x_1 is not full, then the value of in_1 is enqueued in the queue x_1. This is captured by the guard condition $\neg\,\texttt{Full}(in_1)$ and the update code $\texttt{Enqueue}(in_1, x_1)$. Compared to the process `Buffer`, this captures a different style of synchronization: the environment, or the process sending values on the channel in_1, is blocked if the process `Merge` has its internal queue x_1 full. The input task A_i^2 corresponding to processing of the channel in_2 is similar.

The process `Merge` has two output tasks. The task A_o^1 dequeues an element from the queue x_1 and transmits it on the output channel *out*. This is possible when the queue x_1 is not empty. Hence, the task is described by the guard condition $\neg\,\texttt{Empty}(x_1)$ and the update code $out := \texttt{Dequeue}(x_1)$. The task A_o^2 is symmetric and corresponds to transferring the front element of the queue x_2 to the output channel. Note that both the output tasks are associated with the same channel, and thus $\mathcal{A}_{out} = \{A_o^1, A_o^2\}$. When the queues x_1 and x_2 are non-empty, both output tasks are enabled, and either of them can be executed.

The definition is summarized below.

Asynchronous Process

An *asynchronous process* P has:

- a finite set I of typed input channels defining the set of inputs of the form $x\,?\,v$ with $x \in I$ and a value v for x;
- a finite set O of typed output channels defining the set of outputs of the form $y\,!\,v$ with $y \in O$ and a value v for y;
- a finite set S of typed state variables defining the set Q_S of states;
- an initialization *Init* defining the set $[\![Init]\!] \subseteq Q_S$ of initial states;
- for each input channel x, a set $\mathcal{A}_x$ of input tasks, each described by a guard condition over S and an update from the read-set $S \cup \{x\}$ to the write-set S defining a set of input actions $s \xrightarrow{x\,?\,v} t$;
- for each output channel y, a set $\mathcal{A}_y$ of output tasks, each described by a guard condition over S and an update from the read-set S to the write-set $S \cup \{y\}$ defining a set of output actions $s \xrightarrow{y\,!\,v} t$; and
- a set $\mathcal{A}$ of internal tasks, each described by a guard condition over S and an update from the read-set S to the write-set S defining a set of internal actions $s \xrightarrow{\varepsilon} t$.

Exercise 4.1: We want to design an asynchronous adder process `AsyncAdd` with input channels x_1 and x_2 and an output channel y, all of type `nat`. If the ith input message arriving on the channel x_1 is v and the ith input message arriving

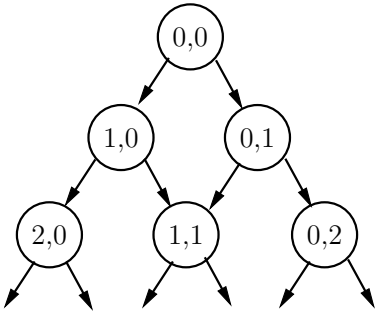

Figure 4.4: Executions of the Process `AsyncInc`

on the channel x_2 is w, then the ith value output by the process `AsyncAdd` on its output channel should be $v + w$. Describe all the components of the process `AsyncAdd`. ■

4.1.3 Executions

The operational semantics of an asynchronous process can be captured by defining its executions. An execution starts in an initial state. At every step, one of the tasks that is enabled in the current state is chosen and executed. This task may be an input task, an output task, or an internal task. Only one task is executed at every step, and the order in which different tasks are executed is totally unconstrained. Such a semantics for asynchronous interaction is called the *interleaving semantics*.

Figure 4.4 shows possible executions of the asynchronous process `AsyncInc` of figure 4.2. Each state is a pair (i, j) of natural numbers corresponding to the values of the variables x and y, respectively. The state $(0, 0)$ is the sole initial state, and each state has two possible transitions: one that increments the value of x corresponding to the execution of the internal task A_x, and one that increments the value of y corresponding to the execution of the task A_y. An execution is a (finite) path through the graph shown in figure 4.4 starting at the root. Note that for the process `AsyncInc`, every state of the form (i, j) is a reachable state. In particular, the state $(i, 0)$ is reachable via an execution that consists of executing the task A_x i times without ever executing the task A_y, corresponding to the left-most path in figure 4.4.

Formally, a finite *execution* of an asynchronous process P consists of a finite sequence of the form

$$s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \xrightarrow{l_3} s_3 \cdots s_{k-1} \xrightarrow{l_k} s_k$$

where for $0 \leq j \leq k$, each s_j is a state of P, s_0 is an initial state of P, and for $1 \leq j \leq k$, $s_{j-1} \xrightarrow{l_j} s_j$ is an input, an output, or an internal action P.

For instance, one possible execution of the process `Buffer` of figure 4.1 is:

$$\texttt{null} \xrightarrow{in\,?\,1} 1 \xrightarrow{out\,!\,1} \texttt{null} \xrightarrow{in\,?\,0} 0 \xrightarrow{in\,?\,1} 1 \xrightarrow{in\,?\,1} 1 \xrightarrow{out\,!\,1} \texttt{null}.$$

Note that the process `Buffer` may execute an unbounded number of input actions before it executes an output action, which issues the most recent input value received.

For the process `Merge` of figure 4.3, below is one possible execution, where the state lists the contents of the queues x_1 and x_2 in that order:

$$(\texttt{null}, \texttt{null}) \xrightarrow{in_1\,?\,0} ([0], \texttt{null}) \xrightarrow{in_1\,?\,2} ([02], \texttt{null}) \xrightarrow{in_2\,?\,5} ([02], [5]) \xrightarrow{out\,!\,5}$$
$$([02], \texttt{null}) \xrightarrow{in_2\,?\,3} ([02], [3]) \xrightarrow{out\,!\,0} ([2], [3]) \xrightarrow{out\,!\,3} ([2], \texttt{null}) \xrightarrow{in_1\,?\,0} ([20], \texttt{null}).$$

In a state such as $([02], [5])$ where both the buffers are non-empty, assuming that the two input buffers are also not full, all the four tasks are enabled. For every possible value v of type `msg`, the possible input actions are: $([02], [5]) \xrightarrow{in_1\,?\,v} ([02v], [5])$ and $([02], [5]) \xrightarrow{in_2\,?\,v} ([02], [5v])$, obtained by executing the input tasks A_i^1 and A_i^2, respectively; and the possible output actions are $([02], [5]) \xrightarrow{out\,!\,0} ([2], [5])$ and $([02], [5]) \xrightarrow{out\,!\,5} ([02], \texttt{null})$, obtained by executing the output tasks A_o^1 and A_o^2, respectively.

Note that the sequence of values output by the process represents a merge of the sequences of input values received on the two input channels. The relative order of values received on the input channel in_1 is preserved in the output sequence, and so is the relative order of values received on the channel in_2, but an input value received on the channel in_1 before a value received on the channel in_2 may appear on the output channel later.

In this example, each individual task is deterministic: for each task, given a state in which the task is enabled, the execution of the task results in a unique update of the variables in its write-set. However, the asynchronous execution model is inherently nondeterministic: at each step, one of the enabled tasks is chosen and executed, and the order in which the tasks execute affects the outputs.

Exercise 4.2: We want to design an asynchronous process `Split` that is the dual of `Merge`. The process `Split` has one input channel *in* and two output channels out_1 and out_2. The messages received on the input channel should be routed to one of the output channels in a nondeterministic manner so that all possible splittings of the input stream are feasible executions. Describe all the components of the desired process `Split`. ■

4.1.4 Extended-State Machines

In section 2.1.6, we explored the use of extended-state machines to specify the behavior of synchronous reactive components. Extended-state machines are

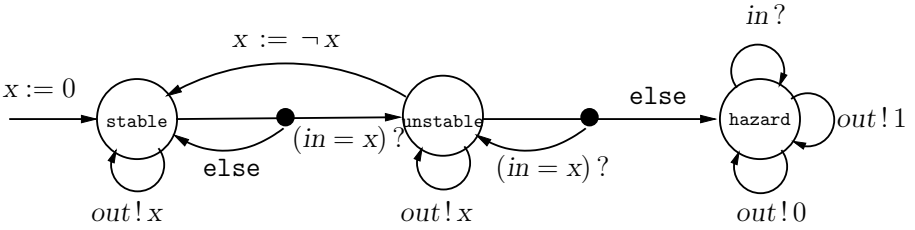

Figure 4.5: An Asynchronous Not Gate `AsyncNot`

used to describe the behavior of asynchronous processes also. In an extended-state machine description, there is an implicit state variable *mode* that ranges over a finite enumerated type. The behavior is described by a graph whose vertices correspond to the modes and whose edges correspond to mode-switches. In the asynchronous case, each mode-switch can access at most one input channel or at most one output channel, and each mode-switch corresponds to a task. We will illustrate the notation using the example of an asynchronous process modeling an *asynchronous Not* gate shown in figure 4.5.

In an asynchronous circuit, in contrast to synchronous circuits, there is no single global clock, and a change in the value of an output due to changes in the values of the inputs is delayed. An asynchronous logic gate is *stable* when its output is the desired function of the inputs and *unstable* otherwise. If the gate is stable and any of the inputs change in a way that violates the stability condition, then the gate turns unstable. The output of an asynchronous gate can change only if the gate is unstable, and when this happens, the gate becomes stable. The time it takes for the gate to update its output is assumed to be arbitrary so that a correctly designed asynchronous circuit is not dependent on concrete values of the delay parameters. If the gate is unstable and any of the inputs change without causing the stability condition to become true, then the gate remains unstable. However, if any of the inputs of an unstable gate change in a way that causes the stability condition to become true, then a hazard is encountered and the gate fails. If a gate has failed, then its output may change arbitrarily. Asynchronous gates and latches should be composed together to form an asynchronous circuit in a manner so as to ensure that no gate ever fails.

Figure 4.5 describes the asynchronous process `AsyncNot`. The process has an input channel *in* and an output channel *out* modeling the input and output wires of the gate. The extended-state machine has three modes `stable`, `unstable`, and `hazard` corresponding to the three modes of operation of the gate. The state variable x captures the value of the output, and this value is issued on the output channel *out*.

Initially, the gate is in the `stable` mode with the output x equal to 0. If the value received on the input channel *in* equals the current output, then this violates

the logic of the Not gate, making it unstable, and if the value of the input channel is the negation of the output x, then the gate continues to stay stable. This is expressed by a *conditional* mode-switch that has multiple targets: the switch out of `stable` corresponding to processing inputs has no guard (that is, it is always enabled), and then if the condition $(in = x)$ holds the target of the mode-switch is `unstable`; otherwise the target of the mode-switch is `stable`.

In the unstable mode, the gate can switch back to the stable mode, toggling the value of x. Processing of an input value in the unstable mode causes the gate to ignore the input by either keeping the mode unchanged (this is the case if the input value is equal to the current output, thereby maintaining the validity of the pending toggling of the output) or switching to the mode `hazard` (this is the case if the input value is the negation of the current output implying a meaningful change in input values in rapid succession without giving the gate a chance to update its output appropriately). Processing of inputs is again expressed by a conditional mode-switch with two possible targets.

In the mode `hazard`, the gate ignores input values (that is, processing an input value has no effect on the state) and issues both output values in a nondeterministic manner.

Each mode-switch corresponds to a single task, and at each step, exactly one mode switch of the machine is executed. In our example, the self-loop on the mode `stable` contributes an output task with the guard condition ($mode =$ `stable`) and the update code $out\,!\,x$. Each of the other three self-loops that involve the output channel contribute one output task each. The switch from the mode `unstable` to `stable` contributes the sole internal task with the guard condition $mode =$ `unstable` and the update code $x := \neg\, x$; $mode :=$ `stable`. The conditional mode-switch out of `stable` contributes an input task with the guard condition ($mode =$ `stable`) and the update code `if` $(in = x)$ `then` $mode :=$ `unstable`. The input task corresponding to the conditional mode-switch out of the mode `unstable` is similar. Finally, the self-loop on the mode `hazard` labeled with $in\,?$ contributes the input task with the guard condition ($mode =$ `hazard`) and empty update code (that is, the state stays unchanged). A possible execution of the process is shown below:

$$\begin{array}{l}(\texttt{stable},0) \xrightarrow{out\,!\,0} (\texttt{stable},0) \xrightarrow{in\,?\,0} (\texttt{unstable},0) \xrightarrow{in\,?\,0} (\texttt{unstable},0) \xrightarrow{\varepsilon} \\ (\texttt{stable},1) \xrightarrow{out\,!\,1} (\texttt{stable},1) \xrightarrow{out\,!\,1} (\texttt{stable},1) \xrightarrow{in\,?\,1} (\texttt{unstable},1) \xrightarrow{in\,?\,0} \\ (\texttt{hazard},1) \xrightarrow{out\,!\,0} (\texttt{hazard},1) \xrightarrow{out\,!\,1} (\texttt{hazard},1) \xrightarrow{in\,?\,0} (\texttt{hazard},1).\end{array}$$

Note that starting in the initial state, if the process is supplied the input $in\,?\,0$, followed by the input $in\,?\,1$, with no intervening output actions, then the resulting mode may be `hazard` or `unstable`. The latter is a possibility if in between the two input actions the process executes the internal action that toggles the state variable x. Note that the internal action of toggling x is decoupled from issuing the output on the output channel out. The only way to ensure that the

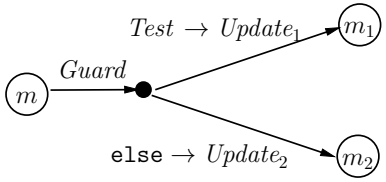

Figure 4.6: A Conditional Mode-switch in Extended-state Machines

gate does not enter the mode `hazard` is for its environment, after supplying the input $in\,?\,0$, to wait for the output $out\,!\,1$ before issuing the subsequent input $in\,?\,1$.

The execution semantics of processes specified using extended-state machines is intuitively simple and can be directly incorporated in the simulation and analysis tools for asynchronous processes. Alternatively, it is possible to translate the extended-state machine description to the task-based formal definition. The general form of a conditional mode-switch is shown in figure 4.6. The mode-switch can be executed if the current mode is m and the guard condition *Guard* holds. The update code corresponds to evaluating the condition *Test*, and if that holds, the code $Update_1$ is executed, and the mode variable is updated to m_1; otherwise the code $Update_2$ is executed, and the mode variable is updated to m_2. Such a mode-switch contributes a task with the guard condition

$$(mode = m) \ \wedge \ Guard$$

and update code

$$\texttt{if}\ Test\ \texttt{then}\ \{Update_1;\ mode := m_1\}\ \texttt{else}\ \{Update_2;\ mode := m_2\}.$$

The variables accessed in the two conditions *Guard* and *Test* and the two updates $Update_1$ and $Update_2$ should be such that the task can be classified as an internal task, an input task associated with a single input channel, or an output task associated with a single output channel. In particular, the key restriction is that the guard condition *Guard* should not refer to input values. That's why we cannot replace the conditional mode-switch out of the mode `stable` by two separate mode-switches, one from the mode `stable` to `unstable` with the guard condition $(in = x)$ and one self-loop with the negated guard condition $(in \neq x)$.

Exercise 4.3: Describe an asynchronous process `AsyncAnd` that models an asynchronous *And* gate with two Boolean input channels in_1 and in_2 and a Boolean output channel *out*. The process can be described as an extended-state machine with three modes as in the case of the process `AsyncNot` in figure 4.5 and with three Boolean state variables. ■

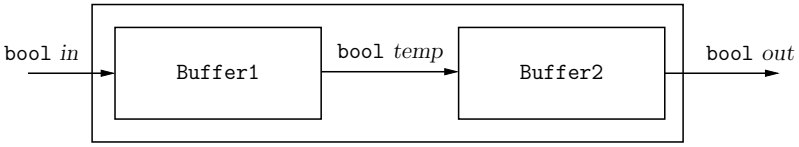

Figure 4.7: Block Diagram for `DoubleBuffer` from Two `Buffer` Processes

4.1.5 Operations on Processes

As discussed in chapter 2, block diagrams can be used to describe composition of synchronous components to form systems in a hierarchical manner. The same design methodology applies to asynchronous processes also. As an example, consider the block diagram of figure 4.7 that uses two instances of the asynchronous process `Buffer` to form a composite process `DoubleBuffer`. The block diagram is structurally identical to the block diagram of the synchronous component `DoubleDelay` of figure 2.15. As before, the meaning of such diagrams can be made precise using three operations: instantiation, parallel composition, and output hiding. A textual description of the process `DoubleBuffer` using these operations is:

$$(\texttt{Buffer}[\,out \mapsto temp\,] \mid \texttt{Buffer}[\,in \mapsto temp\,]) \setminus temp.$$

Input/Output Channel Renaming

The operation of input or output channel renaming is used to achieve the desired communication pattern. In figure 4.7, the asynchronous process `Buffer1` is obtained by renaming the output channel *out* of the process `Buffer` to *temp* and corresponds to the renaming expression $\texttt{Buffer}[\,out \mapsto temp\,]$. Analogously, the process `Buffer2` is obtained by renaming the input channel *in* of the process `Buffer` to *temp* and corresponds to the process $\texttt{Buffer}[\,in \mapsto temp\,]$. When these two processes are composed, the shared name *temp* ensures that the output issued by the process `Buffer1` is consumed by the process `Buffer2` as its input.

When composing processes, we assume that the names of state variables are private, and state variables are implicitly renamed to avoid name conflicts. In our example, we can assume that the state variable of `Buffer1` is called x_1 instead of x, and the state variable of `Buffer2` is called x_2.

The formal definition of the input/output channel renaming operation for processes is similar to the corresponding definition for synchronous components and corresponds to syntactic substitution of channel names throughout its description.

Parallel Composition

The parallel composition operation combines two processes into a single process whose behavior captures the interaction between the two processes running concurrently so that an output action of one is synchronized with an input action of another with the common channel name, and remaining actions are interleaved. To differentiate the asynchronous composition with the synchronous composition (which is denoted $\|$), we use $P_1 \mid P_2$ to denote the composition of two processes P_1 and P_2.

As in the synchronous case, two processes can be composed only if their variable declarations are mutually consistent: there are no name conflicts concerning state variables, and the two sets of output channels are disjoint. These requirements capture the assumption that only one process is responsible for controlling the value of any given variable. An input channel of one can be an input or output channel of the other. Note that the problem of mutually cyclic await-dependencies discussed for the synchronous case does not arise in the asynchronous interaction. If x is an output channel of process P_1 and an input channel of process P_2, and y is an output channel of P_1 and an input channel of P_2, then we can compose P_1 and P_2 without any complications. This is because production of an output is a separate step from processing an input for each of the processes, and hence there can be no dependencies among variables within the same step.

The set of input channels, output channels, and state variables of the composite process are defined as in the synchronous case. Each state variable of a component process is a state variable of the composite process. Each output channel of a component process is an output channel of the composite process. Each input channel of a component process that is not an output of the other is an input channel of the composite process.

The state of the composite process is of the form (s_1, s_2), where s_1 is a state of the process P_1 and s_2 is a state of the process P_2. The two processes initialize their states independently, and thus a composite state (s_1, s_2) is initial if both states s_1 and s_2 are initial states of processes P_1 and P_2, respectively.

Tasks of the Composite Process

When an input channel x is a common input channel to both the processes, both consume an input value on channel x simultaneously, and a possible input action of the composite process corresponding to such an input is obtained by executing input actions of the two processes together. That is, $(s_1, s_2) \xrightarrow{x\,?\,v} (t_1, t_2)$ is an input action of the composite process precisely when $s_1 \xrightarrow{x\,?\,v} t_1$ is an input action of P_1 and $s_2 \xrightarrow{x\,?\,v} t_2$ is an input action of P_2. Consider an input task A_1 of P_1 associated with the channel x, and suppose its guard is $Guard_1$ and update description is $Update_1$. Similarly, suppose an input task A_2 of P_2 associated with the channel x has guard $Guard_2$ and update description

$Update_2$. Then by combining the tasks A_1 and A_2, we obtain an input task A_{12} for the composite process associated with the channel x: its guard condition is $Guard_1 \wedge Guard_2$ and the update code is $Update_1; Update_2$. That is, the task A_{12} for processing input values on the channel x is enabled exactly when both the corresponding input tasks of the two component processes are enabled, and it updates the state variables of P_1 using the update code $Update_1$ and then updates the state variables of P_2 by executing the update code $Update_2$. The order in which the two update descriptions are executed does not really matter as they update disjoint sets of variables. When the processes P_1 and P_2 have multiple input tasks associated with the channel x, the composite process has tasks corresponding to all possible pairings of such tasks of the two processes.

If a channel x is an output channel of one process, say process P_1, and an input channel of the other process P_2, then the two processes synchronize using this channel: when P_1 executes an output action sending a value on the channel x, the receiver P_2 executes a matching input action. The resulting joint action is an output action for the composite. That is, $(s_1, s_2) \xrightarrow{x\,!\,v} (t_1, t_2)$ is an output action of the composite process precisely when $s_1 \xrightarrow{x\,!\,v} t_1$ is an output action of P_1 and $s_2 \xrightarrow{x\,?\,v} t_2$ is an input action of P_2. If the description of an output task A_1 of P_1 associated with channel x is $Guard_1 \rightarrow Update_1$ and the description of an input task A_2 of P_2 associated with channel x is $Guard_2 \rightarrow Update_2$, then the description for the task A_{12} of the composite process obtained by pairing these two tasks is: $Guard_1 \wedge Guard_2 \rightarrow Update_1; Update_2$. Thus, the task is enabled when the guard conditions of both the tasks are satisfied. The update description $Update_1$ updates the state variables of the process P_1 and computes an output value for the channel x. This value is then used by the update code $Update_2$ to update the state variables of P_2. It is worth emphasizing that the guard condition of an input task corresponding to a channel x refers only to the state variables: whether a process is willing to process an input on a channel x depends on its state but not on the value supplied on the channel x. Thus, in the synchronization between two processes P_1 and P_2 using the channel x, the willingness of both the processes to participate in the synchronization using the tasks A_1 and A_2 is captured by the condition $Guard_1 \wedge Guard_2$, which can be evaluated in a given composite state before the process P_1 has executed its update code to determine which output value is to be transmitted on the channel x. If P_1 has multiple output tasks associated with the channel x and/or P_2 has multiple input tasks associated with the channel x, the set of tasks associated with the channel x in the composite process is obtained by considering all possible pairings.

Now consider the case when P_1 has an input channel x that is not a channel of the other process P_2. In this case, to process an input value on the channel x, the composite process simply executes the input task of P_1 corresponding to the channel x, and the state of P_2 stays unchanged during such an input action. For every input action $s_1 \xrightarrow{x\,?\,v} t_1$ of process P_1 and every state s of process P_2, the composite process has an input action $(s_1, s) \xrightarrow{x\,?\,v} (t_1, s)$. For this purpose, we

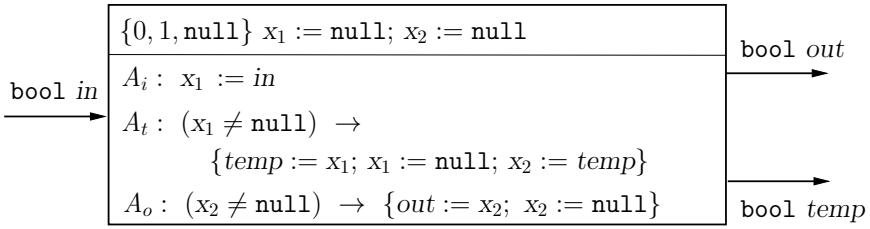

Figure 4.8: Asynchronous Parallel Composition of Two `Buffer` Processes

declare each input task of process P_1 associated with the channel x to be also an input task of the composite. Note that the guard condition and the update description of such a task stays unchanged and does not refer to the variables of P_2.

The same holds for output actions for a channel that involves only one process. If y is an output channel of the process P_1 and is not a channel of P_2, then each output task of P_1 associated with the output channel y is declared to be an output task of the composite process with the same guard condition and update description. Enabledness of such a task does not depend on the process P_2, and executing such a task leaves the state of P_2 unchanged. Thus, for every output action $s_1 \xrightarrow{y\,!\,v} t_1$ of process P_1 and every state s of process P_2, the composite process has an output action $(s_1, s) \xrightarrow{y\,!\,v} (t_1, s)$.

Finally, an internal action of the composite process is an internal action of exactly one of the two component processes, with the other process maintaining its state unchanged. Thus, every internal task of each of the two processes is declared to be an internal task of the composite with the same guard condition and update description.

The composition of processes `Buffer1` and `Buffer2` gives the process shown in figure 4.8. It has state variables $\{x_1, x_2\}$, output channels $\{temp, out\}$, and input channels $\{in\}$. For the composite process, the input task A_i is the same as the corresponding input task for the process `Buffer1`, and the output task A_o is the same as the corresponding output task for the process `Buffer2`. Since *temp* is a common channel, the corresponding output task A_t is obtained by composing the specifications of the output task of the process `Buffer1` responsible for producing outputs on the channel *temp*, with the input task of the process `Buffer2` responsible for processing the inputs on the channel *temp*. The guard condition for this task then is the conjunction of the guard conditions of the two contributing tasks, which turns out to be only $(x_1 \neq \texttt{null})$ since the input task of `Buffer2` is always enabled with guard condition 1. The update description executes the update code of the output task of `Buffer1` followed by the input task of `Buffer2`. The composite has no internal tasks. Thus, only the process `Buffer1` participates in the processing of the channel *in*, the two processes

synchronize on *temp*, and only the process `Buffer2` participates in producing the output on the channel *out*.

We now summarize the formal definition of parallel composition of asynchronous processes.

ASYNCHRONOUS PROCESS COMPOSITION

Let $P_1 = (I_1, O_1, S_1, \mathit{Init}_1, \{\mathcal{A}_x^1 \mid x \in I_1\}, \{\mathcal{A}_y^1 \mid y \in O_1\}, \mathcal{A}_1)$ and $P_2 = (I_2, O_2, S_2, \mathit{Init}_2, \{\mathcal{A}_x^2 \mid x \in I_2\}, \{\mathcal{A}_y^2 \mid y \in O_2\}, \mathcal{A}_2)$ be two asynchronous processes such that O_1 and O_2 are disjoint. Then the *parallel composition* $P_1 \,|\, P_2$ is the asynchronous process P defined by:

- the set S of state variables is $S_1 \cup S_2$;
- the set O of output channels is $O_1 \cup O_2$;
- the set I of input channels is $(I_1 \cup I_2) \setminus O$;
- the initialization is given by $\mathit{Init}_1; \mathit{Init}_2$;
- for each input channel $x \in I$, (1) if $x \notin I_2$, then the set of input tasks $\mathcal{A}_x$ is $\mathcal{A}_x^1$; (2) if $x \notin I_1$, then the set of input tasks $\mathcal{A}_x$ is $\mathcal{A}_x^2$; and (3) if $x \in I_1 \cap I_2$, then for each task $A_1 \in \mathcal{A}_x^1$ and $A_2 \in \mathcal{A}_x^2$, the set of input tasks $\mathcal{A}_x$ contains the task described by $\mathit{Guard}_1 \wedge \mathit{Guard}_2 \rightarrow \mathit{Update}_1; \mathit{Update}_2$, where $\mathit{Guard}_1 \rightarrow \mathit{Update}_1$ is the description of the task A_1 and $\mathit{Guard}_2 \rightarrow \mathit{Update}_2$ is the description of the task A_2;
- for each output channel $y \in O$, (1) if $y \in O_1 \setminus I_2$, then the set of output tasks $\mathcal{A}_y$ is $\mathcal{A}_y^1$; (2) if $y \in O_2 \setminus I_1$, then the set of output tasks $\mathcal{A}_y$ is $\mathcal{A}_y^2$; (3) if $y \in O_1 \cap I_2$, then for each task $A_1 \in \mathcal{A}_y^1$ and $A_2 \in \mathcal{A}_y^2$, the set of output tasks $\mathcal{A}_y$ contains the task described by $\mathit{Guard}_1 \wedge \mathit{Guard}_2 \rightarrow \mathit{Update}_1; \mathit{Update}_2$, where $\mathit{Guard}_1 \rightarrow \mathit{Update}_1$ is the description of the task A_1 and $\mathit{Guard}_2 \rightarrow \mathit{Update}_2$ is the description of the task A_2; and (4) if $y \in O_2 \cap I_1$, then for each task $A_1 \in \mathcal{A}_y^1$ and $A_2 \in \mathcal{A}_y^2$, the set of output tasks $\mathcal{A}_y$ contains the task described by $\mathit{Guard}_2 \wedge \mathit{Guard}_1 \rightarrow \mathit{Update}_2; \mathit{Update}_1$, where $\mathit{Guard}_1 \rightarrow \mathit{Update}_1$ is the description of the task A_1 and $\mathit{Guard}_2 \rightarrow \mathit{Update}_2$ is the description of the task A_2;
- the set $\mathcal{A}$ of internal tasks of the composite is $\mathcal{A}_1 \cup \mathcal{A}_2$.

Output Hiding

If y is an output channel of a process P, then the result of *hiding* y in P gives a process that behaves exactly like the process P, but y is no longer an output that is observable outside. This is achieved by removing y from the set of output channels and turning each output task associated with the channel y into an internal task by declaring y to be a *local* variable. Recall that a local variable

is an auxiliary variable used in the description of the update code of a task and is not stored in the state.

Let us revisit the process `Buffer1 | Buffer2`. If we hide the intermediate output channel *temp*, we get the desired composite process `DoubleBuffer`: the set of state variables is $\{x_1, x_2\}$, the set of output channels is $\{out\}$, the set of input channels is $\{in\}$, and the initialization is given by $x_1 := \texttt{null}$; $x_2 := \texttt{null}$. The input task A_i and the output task A_o are unchanged from `Buffer1 | Buffer2`. The process `DoubleBuffer` has one internal task described by

$(x_1 \neq \texttt{null}) \rightarrow$
$\{$ `local bool` *temp*;
temp $:= x_1$; $x_1 :=$ `null`;
$x_2 :=$ *temp* $\}$.

Exercise 4.4: Consider the asynchronous process

$$\texttt{Merge}[\, out \mapsto temp\,] \mid \texttt{Merge}[\, in_1 \mapsto temp\,][\, in_2 \mapsto in_3\,]$$

obtained by connecting two instances of the process `Merge`. Show the "compiled" version of this composite process similar to the description in figure 4.8. Explain the input/output behavior of this composite process. ■

4.1.6 Safety Requirements

In chapter 3, we studied how to specify and verify safety requirements of transition systems. The same techniques apply to asynchronous processes also. Given an asynchronous process P, we can define an associated transition system T as follows:

- the state variables S of P are the state variables of T;
- the initialization specification *Init* of P is also the initialization of T; and
- the transition description of T corresponds to choosing either an internal, an input, or an output task A of P such that the guard of A is satisfied, and executing the corresponding update description. For output tasks, the corresponding output channel is converted into a local variable; and for input tasks, the corresponding input channel is converted into a local variable whose value is chosen nondeterministically at the beginning.

Thus, $s \rightarrow t$ is a transition of T precisely when the process P has either an input or an output or an internal action from state s to t.

A property φ over state variables of an asynchronous process is an invariant of the system if all reachable states of the corresponding transition system satisfy the property φ. For instance, consider the process `AsyncInc` of figure 4.4 and the requirement that the values of the two variables x and y remain at most c apart for a given constant c. This corresponds to checking whether the property

$|x - y| \leq c$ is an invariant of the system. It turns out that this is not the case for the process `AsyncInc`, no matter how large the constant c is.

Concepts such as inductive invariants can be used to prove safety requirements of asynchronous processes. For instance, to show that a state property φ is an inductive invariant, we need to show that it (1) holds initially, and (2) is preserved by every transition. Since a transition corresponds to executing exactly one task, we need to show that φ is preserved by the execution of every task.

Safety monitors can be used to capture safety requirements that cannot be directly stated in terms of state variables. In the asynchronous setting, a safety monitor for a process with input variables I and output variables O is another asynchronous process with internal state and $I \cup O$ as its input variables. Such a monitor synchronizes with the observed system P on the input and output actions of P. The monitor is described by an extended-state machine, such that an execution that ends up in an "error" mode of the monitor indicates a violation of the desired safety requirement.

Enumerative and symbolic reachability algorithms discussed in sections 3.3 and 3.4 also apply to verification of asynchronous processes.

Exercise 4.5: Consider the process `AsyncNot` of figure 4.5. In this exercise, we want to design an asynchronous process `AsyncNotEnv` that interacts with `AsyncNot`. The process `AsyncNotEnv` has a Boolean input channel *out* and a Boolean output channel *in*. It first outputs the value 0 and then is able to receive inputs. It waits until the received input equals 1 and proceeds to output the value 1, and then waits until the received input equals 0. This cycle is then repeated. Model the desired asynchronous process `AsyncNotEnv` as an extended-state machine. Consider the asynchronous composition `AsyncNot | AsyncNotEnv` and argue that (`AsyncNot`.*mode* $\neq$ `hazard`) is an invariant of the composite process. ■

4.2 Asynchronous Design Primitives

4.2.1 Blocking vs. Non-blocking Synchronization

In the asynchronous model, exchange of information between two processes, and thus synchronization between them, occurs when the production of an output by one process is matched with the consumption of the corresponding input by another. Suppose x is an output channel of a process P_1 and an input channel of another process P_2. Let A_1 be an output task of P_1 corresponding to the channel x. Suppose the process P_1 is in a state s_1, in which this output task A_1 is enabled. Then the process P_1 is ready to send a value on its output channel x. Suppose s_2 is the current state of P_2. If some input task A_2 of P_2 associated with the channel x is enabled in the state s_2, then the process P_2 is willing to accept an input on the channel x, and the composite process can execute a synchronizing action on the channel x. However, if none of the input

tasks of P_2 associated with the channel x is enabled in the state s_2, then the process P_2 is not willing to accept an input on the channel x, and effectively the process P_1 is blocked from executing its output task. This is a form of *blocking* communication where the producer P_1 needs the cooperation of the receiver P_2 to produce an output on the channel x. A process that is willing to accept every input in every state does not prevent the producer from producing outputs and is said to be *non-blocking*.

In our model, a process is willing to process inputs on a channel x precisely when the guard condition of one of the input tasks associated with the channel x is satisfied. For a process to be non-blocking, we require that the *disjunction* of the guards of all the tasks corresponding to an input channel be a valid formula, that is, equivalent to the Boolean constant 1.

> Non-blocking Process
>
> An asynchronous process P is said to be *non-blocking* if for every input channel x and for every state s, some task in the set $\mathcal{A}_x$ of tasks associated with the channel x is enabled in the state s.

The process `Buffer` of figure 4.1 is non-blocking: its environment can always supply a value on the input channel *in* even though some of these values are effectively lost. However, the process `Merge` of figure 4.3 is blocking: an input on the channel in_1 cannot be processed if the queue x_1 is full, and thus the producer of outputs on the channel in_1 has to wait until this queue becomes non-full. The process `AsyncNot` of figure 4.5 is non-blocking: it always accepts inputs even though supplying inputs in rapid succession can lead it to the hazardous state.

The process `DoubleBuffer` obtained by composing two `Buffer` processes is non-blocking. In fact, it is easy to verify that all the operations defined in section 4.1.5 preserve the property of being non-blocking: if all the component processes in a block diagram are non-blocking, then so is the composite process corresponding to the block diagram.

In designing asynchronous systems, both styles of synchronization, non-blocking and blocking, are common. In the non-blocking designs, if a process P_1 sends an output value to another process P_2, then typically an explicit acknowledgment from the process P_2 back to process P_1 is needed for P_1 to be sure that its output was examined by P_2. In the implementation of blocking synchronization, the run-time system must somehow ensure that the receiver is willing to participate in the synchronizing action.

4.2.2 Deadlocks

Deadlock is a commonly occurring error in asynchronous designs. In a system composed of multiple processes, a deadlock refers to a situation in which each process is waiting for some other process to execute a task, but no task is enabled, and thus there is no continuation of the execution.

To illustrate how deadlocks can arise, consider the system consisting of two processes P_1 and P_2 shown in figure 4.9. The process P_1 can generate requests of one type that are serviced by the process P_2. For our purpose, the data values exchanged are not particularly relevant. Hence, we model a request by process P_1 as a message with the value $\texttt{req}_1$ sent on the channel x_1 from process P_1 to process P_2 and the corresponding response by process P_2 as a message with the value $\texttt{resp}_1$ sent on the channel x_2 from process P_2 to process P_1. Similarly, the process P_2 can generate requests of another kind, and each such request is modeled as a message with the value $\texttt{req}_2$ sent on the channel x_2 from process P_2 to process P_1. Each such request is serviced by process P_1, and the corresponding response is modeled as a message with the value $\texttt{resp}_2$ sent on the channel x_1 from process P_1 to process P_2.

The description of the process P_1 is shown in figure 4.9. The process has an internal queue y_1 that is used to store messages received on its input channel x_2. The description of its tasks uses a mixture of two styles of specifications we have seen so far: the input task is listed explicitly, whereas the computation of the process corresponding to the internal and output tasks is described using an extended-state machine. The input task A_2 is always enabled and simply enqueues each message received on the input channel into the queue y_1. Initially, the mode is `idle`. In this mode, if the process finds a message req_2 at the front of the queue y_1, then it dequeues this request and switches to the mode `busy`. The mode `busy` captures the internal state of the process P_1, where the computation needed to service an incoming request occurs. The corresponding response is then issued on the output channel (captured by the update $x_1\,!\,\texttt{resp}_2$), and the process returns to the `idle` mode. In the idle mode, the process can also generate a request on its own. This is modeled by the switch to the mode `wait` with the output action $x_1\,!\,\texttt{req}_1$. In the mode `wait`, the process P_1 is waiting for a response from the other process and is unwilling to process requests issued by P_2. Hence, the process can switch from the mode `wait` back to `idle` only when the first message in the queue y_1 is a response message $\texttt{resp}_1$; if so, it is removed from the queue.

The description of the process P_2 is symmetric. Now consider the composed process $P_1 \,|\, P_2$. Suppose the process P_1 issues the request $\texttt{req}_1$ on the channel x_1. This request gets stored in the internal queue y_2 of process P_2. This step is captured by the action shown below, where each state is described by listing the values of variables $P_1.\texttt{mode}$, y_1, $P_2.\texttt{mode}$, and y_2, in that order:

$$(\texttt{idle}, \texttt{null}, \texttt{idle}, \texttt{null}) \xrightarrow{x_1 ! req_1} (\texttt{wait}, \texttt{null}, \texttt{idle}, [\texttt{req}_1]).$$

At this point, two tasks are enabled: the internal task of the process P_2 corresponding to the mode-switch from the mode `idle` to mode `busy` and the output task of the process P_2 corresponding to the mode-switch from the mode `idle` to mode `wait`. If the former task executes first, then the computation will progress as intended. However, if the latter task executes first, then the corresponding

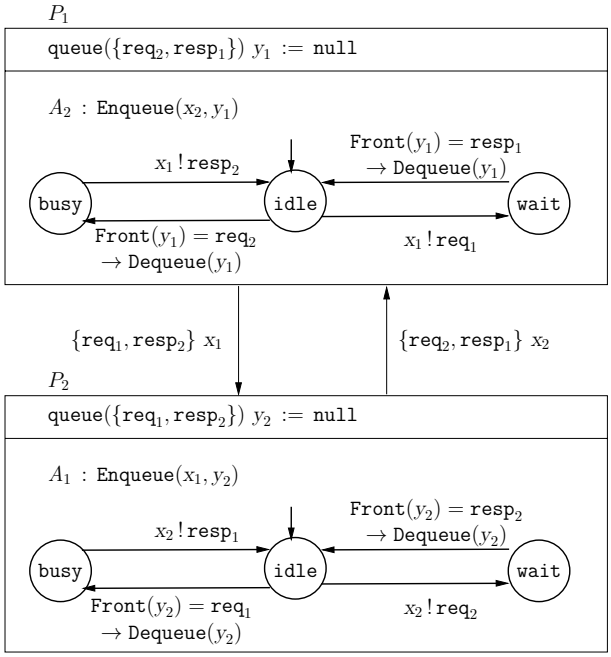

Figure 4.9: Illustrating Deadlocks

transition is:

$$(\texttt{wait}, \texttt{null}, \texttt{idle}, [\texttt{req}_1]) \xrightarrow{x_2!req_2} (\texttt{wait}, [\texttt{req}_2], \texttt{wait}, [\texttt{req}_1]).$$

In the resulting state, no task is enabled: the process P_1 is expecting a response from P_2 and vice versa. Such a state is a deadlock and should be considered a bug in the design.

In general, a state s of an asynchronous process P is a *deadlock* state if (1) no task is enabled in the state s, and (2) the state s does not correspond to a *successful* termination of the system. The latter condition is specific to the design problem; for instance, in the leader election problem, a state in which all processes have already made a decision to be a leader or a follower is considered to be a successful terminal state. Except for such successful terminal states, we expect the system to continue executing. Thus, *absence of deadlocks* is a generic safety requirement that is expected to be an invariant of all asynchronous designs.

4.2.3 Shared Memory

In a shared memory architecture, processes communicate by reading and writing shared variables and, more generally, shared objects. In this section, we will

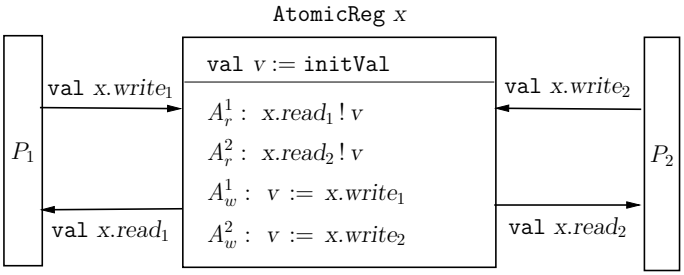

Figure 4.10: Atomic Register Supporting Read and Write Operations

illustrate how to model shared variables as asynchronous processes. In the asynchronous model, executions of different tasks are interleaved. A crucial design decision concerns how much computation can happen in one computation step of a task, that is, which operations are supported by shared objects as *atomic* operations that can be executed in a single step. We first discuss the model of atomic registers, where the only allowed operations are the most basic read and write operations.

Atomic Registers

Figure 4.10 shows the process `AtomicReg` that models a variable (or a register) x shared between two processes P_1 and P_2. The only atomic operations supported by this shared object are read and write, and such an object is called an *atomic register*. The description is parameterized by the set of values that the register can hold, denoted `val`, and the initial value of the register, denoted `initVal`.

The internal state variable v of the object holds its current value and is initialized to the value `initVal`. The channels $x.read_1$ and $x.read_2$ are used to model the read operations. The channel $x.read_1$ is an output channel for the atomic register and is an input channel for the process P_1. When the process P_1 wants to read the register, it executes the input action $y := x.read_1$, where y is a state variable of P_1, which is synchronized with the output action of the task A_r^1. Executing this action transmits the current state of the register x, and as a result, the updated value of the state variable y of P_1 is the current value of the register, whereas the state of the register stays unchanged. Thus, synchronization of the atomic register with the process P_1 on the channel $x.read_1$ transmits the value of the register to P_1. Note that the task A_r^1 of the process `AtomicReg` is always enabled, and thus whether the process P_1 can execute the task corresponding to reading the register depends solely on the guard condition of the task in P_1.

Analogously, the channels $x.write_1$ and $x.write_2$ are used to model the write operations. When the process P_1 wants to update the register by writing the value u, it executes the output action $x.write_1 \,!\, u$, which is synchronized with the input action of the task A_w^1 and updates the internal state of the register x to the

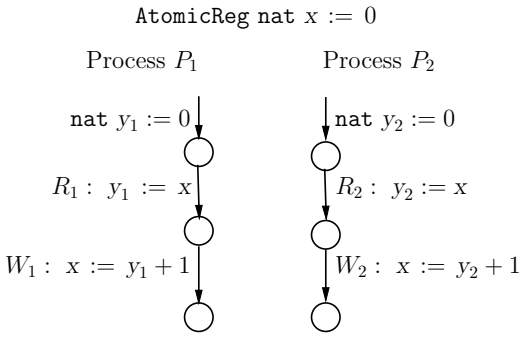

Figure 4.11: Data Race Example: Shared Counter

value received on the channel. If y is a state variable of the process P_1, then to update the shared register x with the current value of y, the process can execute the output statement $x.write_1 := y$. Since the task A_w^1 is always enabled, the enabledness of this joint activity depends solely on the guard condition of the corresponding task in P_1.

The communication pattern for the process P_2 is analogous.

Data Races

Consider the asynchronous system shown in figure 4.11. It consists of three processes. The shared register x is an instantiation of the `AtomicReg` process where the type `val` is `nat` and the initial value `initVal` is 0. In our description, such a shared register is declared using the familiar syntax for declaring variables. We will also refer to the shared variable without explicitly mentioning the associated read/write channels. For example, a read reference to the shared register x by process P_1 is an abbreviation for its input channel $x.read_1$, and an update of the shared register x by process P_2 is an update of its output channel $x.write_2$.

The two asynchronous processes P_1 and P_2 communicate by reading and writing the shared register x. The process P_1 has a state variable y_1, which is initialized to 0. The process first reads the value of x by executing the statement $y_1 := x$ (task R_1). Execution of this statement involves synchronization of the processes P_1 and x on the channel $x.read_1$. The process P_1 then writes the value $y_1 + 1$ back to the shared register x by executing the statement $x := y_1 + 1$ (task W_1), which involves synchronization of the processes P_1 and x on the channel $x.write_1$.

The process P_2 is symmetric: it reads the value of the shared register x in its internal state variable y_2 (task R_2) and writes the incremented value back to the shared register (task W_2).

Interleaving	x	y_1	y_2
$R_1; R_2; W_1; W_2$	1	0	0
$R_1; W_1; R_2; W_2$	2	0	1
$R_1; R_2; W_2; W_1$	1	0	0
$R_2; R_1; W_2; W_1$	1	0	0
$R_2; W_2; R_1; W_1$	2	1	0
$R_2; R_1; W_1; W_2$	1	0	0

Figure 4.12: All Possible Executions of Shared Counter of Figure 4.11

A step of the composed system corresponds to executing one of the tasks of the two processes. Executions of the composed system resulting from all possible interleavings of the four tasks are shown in figure 4.12. With each such execution, we list the values of the variables x, y_1, and y_2 at the end of the execution. Observe that when all the tasks have been executed once, the final value of the shared register x can be either 1 or 2. If the desired intent of each process is to increment the value of x, then a final value of 1 corresponds to a lost increment, a potential bug. Such a bug is caused by the other process accessing the shared register in between the execution of read and write access statements of one process. This type of interference between concurrent accesses to shared objects by asynchronous processes is called a *data race*.

Mutual Exclusion Problem

In the illustrative example of a shared counter of figure 4.11, suppose the process P_1 wants to ensure that the value of the shared object x stays unchanged between the execution of the read and write statements by P_1. Note that the shared object does not support both reading and writing in a single atomic step: the process P_1 cannot use the statement $x := x + 1$ to increment the counter atomically as it would involve two distinct synchronizations on two separate channels, $x.read_1$ and $x.write_1$. To ensure that the process P_1 has an *exclusive* access to the shared object as it reads and then updates the shared register, we need to design a protocol to solve the classical coordination problem of *mutual exclusion*.

Suppose we have two or more asynchronous processes that need access to a critical shared resource, such as the shared counter of our illustrative example. At any time, only one process should be using the shared resource. The allocation of the resource is not governed by a central coordinator, but processes need to coordinate among themselves to ensure such a mutually exclusive access. We assume that processes can communicate using atomic registers. Initially, a process starts in the mode `Idle`. It accesses the shared resource in the mode `Crit`, classically known as the *critical section*. In our example, once the process enters the critical section, it can read the value of the shared counter, add one to it, and write the updated value back to the shared register. We want to design the

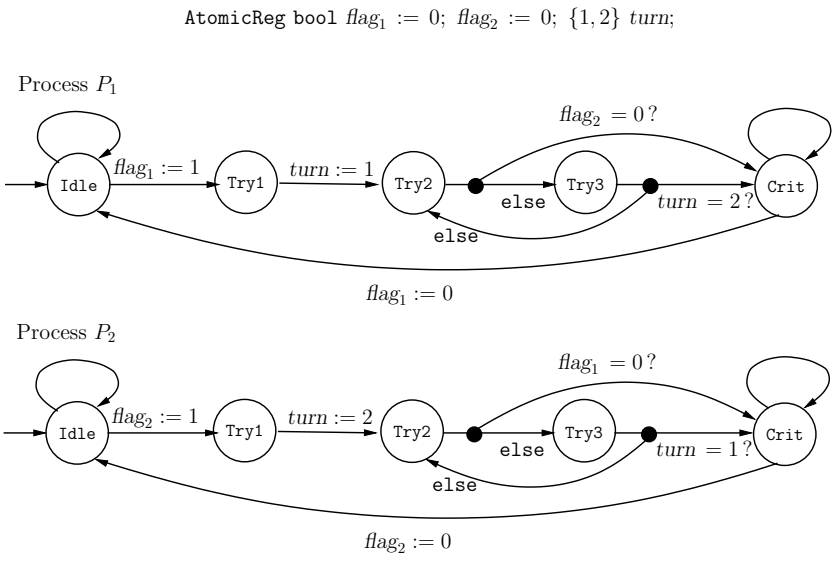

Figure 4.13: Peterson's Mutual Exclusion Protocol

entry code that the process should execute when it wants to switch from the mode `Idle` to the mode `Crit` and the exit code that the process should execute when it has finished its job in the critical section before returning to the idle mode. The safety requirement is *mutual exclusion*: no two processes should be in the critical section simultaneously. The other requirement is *deadlock freedom*: it should not be the case that one of the processes wants to enter the critical section but none is allowed to enter. Note that the safety requirement can be made precise using invariants, and formalization of the deadlock freedom as a liveness requirement is addressed in chapter 5.

Peterson's Mutual Exclusion Algorithm

Figure 4.13 shows Peterson's protocol, a classical solution to the mutual exclusion problem for the case of two processes. The processes communicate via three shared atomic registers: *turn*, $flag_1$, and $flag_2$. The process P_1 is initially in the mode `Idle`. The process has seven tasks, each corresponding to a mode-switch as described below.

1. The self-loop on the mode `Idle` indicates that the process may stay in this mode for arbitrarily many steps.

2. In the mode `Idle`, when the process needs access to the shared resource, it sets the Boolean register $flag_1$ to 1 and switches to the mode `Try1`.

3. In the mode `Try1`, the process updates the shared variable *turn* to its identifier, namely 1, and switches to the mode `Try2`.

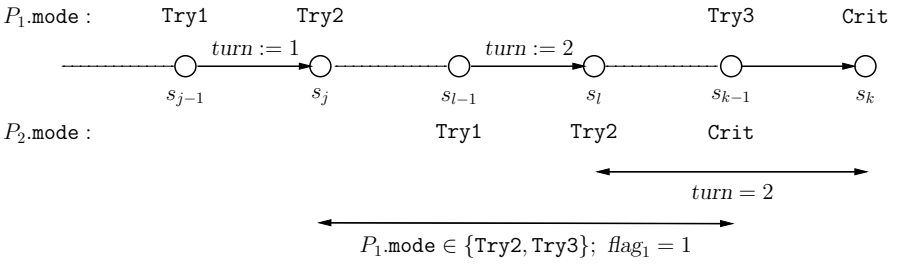

Figure 4.14: Analysis of Potential Counterexample Execution

4. In the mode `Try2`, the process reads the value of the variable $flag_2$. If $flag_2$ is 0, then it concludes that the other process does not need the resource and proceeds to the critical section. Otherwise, it switches to the mode `Try3`. This is formally modeled as a conditional mode-switch that involves the read channel between the process P_1 and shared register $flag_2$.

5. In the mode `Try3`, using a conditional switch, the process checks the value of the shared register *turn*. If *turn* equals 2, then it concludes that the process P_2 updated *turn* to 2 *after* the process P_1 updated *turn* to 1, and in this case, the process P_1 proceeds to the critical section. If it finds *turn* to be 1, then it concludes that the process P_2 updated *turn* to 2 *before* the process P_1 updated *turn* to 1 and returns to the mode `Try2` to check the value of $flag_2$ again.

6. The mode `Crit` corresponding to the critical section has a self-loop indicating that the process may spend an arbitrary number of steps in the critical section.

7. In the mode `Crit`, when the process no longer needs the shared resource, it updates the variable $flag_1$ back to 0 and returns to the initial mode `Idle`.

The process P_2 is symmetric.

We want to argue that Peterson's solution indeed satisfies the mutual exclusion requirement. First, observe that only the process P_1 writes to the shared register $flag_1$. The process sets $flag_1$ to 1 when it leaves the mode `Idle` and resets it to 0 when it returns to this mode. Hence, the value of $flag_1$ is 0 exactly in those states in which the mode of the process P_1 is `Idle`. Symmetrically, the value of the Boolean variable $flag_2$ is 0 exactly in those states in which the mode of the process P_2 is `Idle`.

To prove that there is no execution that leads both processes to be in the critical section simultaneously, let us assume to the contrary. Let $\rho = s_0, s_1 \dots s_k$ be a shortest execution such that the modes of both processes equal `Crit` in the

state s_k. In such an execution, the last step must correspond to some process, say P_1, switching its mode to `Crit` (if not, in the state s_{k-1}, both processes are already in their critical sections, and thus ρ is not the shortest counterexample demonstrating the violation of the desired requirement). The process P_1 can update its mode from `Try2` to `Crit` provided $flag_2$ is 0, or from `Try3` to `Crit` provided *turn* is 2. In the state s_{k-1}, the process P_2 is already in the critical section, and hence $flag_2$ must be 1, and thus only the latter case is possible. Suppose the transition from the state s_{j-1} to s_j is the latest write to *turn* by the process P_1, and the transition from the state s_{l-1} to s_l is the latest write to *turn* by the process P_2 in this execution. Since *turn* is 2 at the end of the execution, we conclude that $j < l$ (that is, the most recent update to *turn* must be by the process P_2). Figure 4.14 depicts the scenario corresponding to this execution. The mode of the process P_1 must be either `Try2` or `Try3` in all the states $s_j, s_{j+1}, \ldots s_{k-1}$, and hence the value of $flag_1$ must be 1 in all these states. We can conclude that the value of *turn* is 2 and $flag_1$ is 1 in all the states $s_l, s_{l+1}, \ldots s_{k-1}$. This implies that the switching conditions for the two possible ways for the process P_2 to enter its critical section (($flag_2 = 0$) from the mode `Try2`, and ($turn = 1$) from the mode `Try3`) are false during this interval. Since the mode of the process P_2 is `Try2` in the state s_l and `Crit` in the state s_k, we obtain a contradiction (that is, the postulated execution ρ witnessing a violation of the safety requirement cannot exist).

We can also show that Peterson's protocol does not deadlock: it cannot happen that one of the processes wants to enter the critical section but none is allowed to enter. If only one process, say P_1, wants to enter the critical section, then the other process P_2 is in the mode `Idle`, and the variable $flag_2$ is 0. In this case, the process P_1 will succeed when it checks the value of $flag_2$ in the mode `Try2`. If both processes are trying to enter the critical section, then it cannot happen that both get stuck in the cycle between the modes `Try2` and `Try3`: once both are past their updates to the variable *turn*, its value does not change, and depending on its value, one of them must succeed in the test in the mode `Try3`.

Test&Set Registers

Figure 4.15 shows the process `Test&SetReg` that models a shared object that stores a Boolean value but supports the primitive operations of *test&set* and *reset*. The test&set operation sets the shared register to 1 while returning the old value, and the reset operation updates the shared register to 0. The state variable v, initialized to 0, stores the current value of the register. When the process P_1 wants to execute the test&set operation, it executes an input action on the channel $x.t\&s_1$ and synchronizes with the output action executed by the register. The output task A^1_{ts0} is enabled when the value of the register is 0 and it transmits 0 while updating the state to 1. The output task A^1_{ts1} is enabled when the value of the register is 1, and it transmits 1 while keeping the state unchanged. Note that the transmission of the current value and its update happen

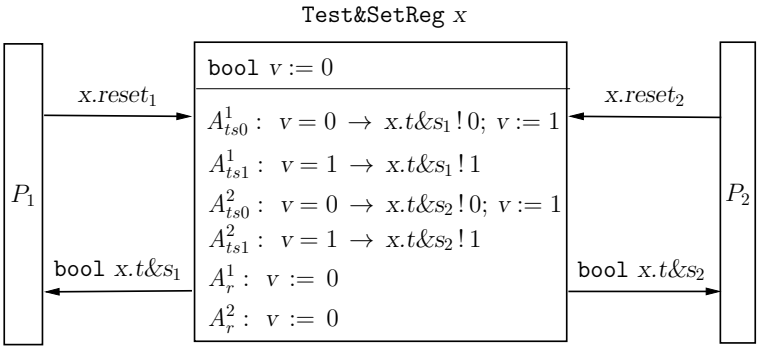

Figure 4.15: Boolean Register Supporting *test&set* and *reset* Operations

atomically within a single step, and this results in a more powerful communication scheme. If both processes P_1 and P_2 are attempting to synchronize with the `Test&SetReg` process, then the first process to synchronize will receive the value 0, and this will set the state of the register to 1, causing the subsequent process to receive the response 1.

Whenever the process P_1 wants to reset the register, it executes the output action for the channel $x.reset_1$, which gets synchronized with the execution of the input task A_r^1 which updates the register to 0. Note that no value needs to be associated with the reset operation.

In the presence of `Test&Set` registers, it is easy to implement a solution to the mutual exclusion problem. Figure 4.16 shows a solution using a single shared `Test&Set` register *free*. When the value of this shared object is 0, the critical section is unoccupied. When a process wants to enter the critical section, it simply executes the *test&set* operation on the shared register. If the operation returns 0, then the process proceeds to the critical section, and if the operation returns 1, the process tries again. Upon leaving the critical section, the process resets the value of the shared register to 0. Note that both processes are identical. It is easy to verify that the protocol satisfies the mutual exclusion requirement and is also deadlock-free.

The specification of shared objects such as `AtomicReg` and `Test&SetReg` can be generalized so that the object is shared among multiple processes instead of two.

Exercise 4.6: Consider the transition system corresponding to Peterson's mutual exclusion protocol. The set of state variables for this system contains the variables P_1.`mode`, P_2.`mode`, *turn*, $flag_1$, and $flag_2$. Draw the reachable subgraph of this transition system. How many states are reachable? ■

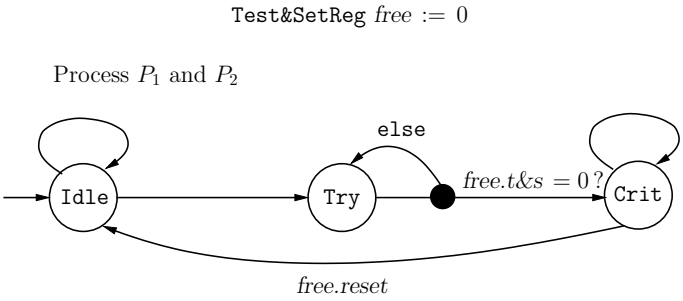

Figure 4.16: Mutual Exclusion Using `Test&Set` Register

Exercise 4.7: In an attempt to "optimize" the two-process mutual exclusion protocol of figure 4.13, someone proposes that the shared register *turn* is not necessary. Consider the modified solution of figure 4.17 that uses only the Boolean shared registers $flag_1$ and $flag_2$. Does this solution satisfy the mutual exclusion requirement? If your answer is yes, then give an informal argument of correctness or else show a counterexample execution. Is this revised protocol a satisfactory solution to the mutual exclusion problem? ■

Exercise 4.8: In Peterson's mutual exclusion protocol (see figure 4.13), the process P_1, when it wants to enter the critical section, first sets the register $flag_1$ to 1 and then sets the register *turn* to 1. Suppose we switch the order in which these two steps are executed. That is, consider a modified version of Peterson's protocol in which the process P_1, when it wants to enter the critical section, first sets the register *turn* to 1 and then sets the register $flag_1$ to 1; symmetrically, the process P_2, when it wants to enter the critical section, first sets the register *turn* to 2 and then sets the register $flag_2$ to 1. Everything else stays the same. Does the modified protocol satisfy the requirement of mutual exclusion? If yes, give a brief justification; if no, show a counterexample. ■

Exercise 4.9*: Consider two asynchronous processes P_1 and P_2 that communicate using a shared atomic register x of type `nat` with initial value 1. The process P_1 reads the shared register and stores the value in its internal state variable u_1, reads it again and stores the value in another state variable v_1, updates the shared register value with the sum of u_1 and v_1, and repeats this sequence of read, read, and write. The process P_2 is symmetric: it reads the shared register and stores the value in its internal state variable u_2, reads it again and stores the value in another state variable v_2, updates the shared register value with the sum of u_2 and v_2, and repeats this sequence. Let us say that a value n is reachable if there is an execution of the system in which the value of the shared register x is n at the end of the execution. Which values are reachable? Hint: try to find executions that demonstrate the reachability of values 5, 6, 7, and 8. ■

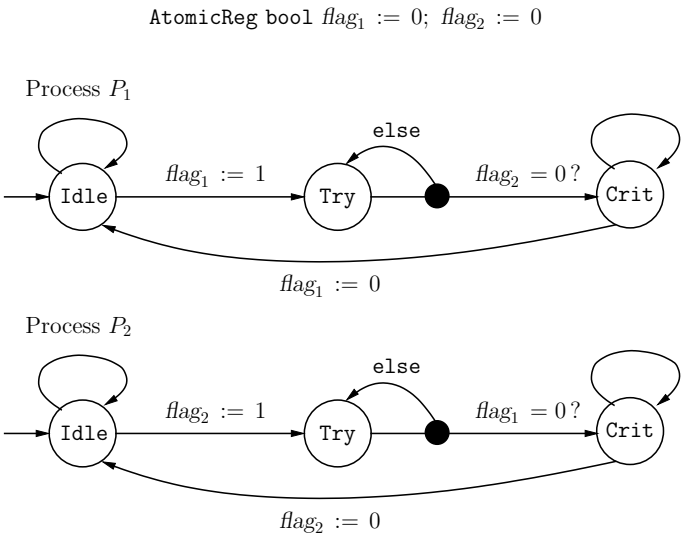

Figure 4.17: Modified Peterson's Mutual Exclusion Protocol

4.2.4 Fairness Assumptions *

The execution of a process in the asynchronous model is obtained by interleaving executions of different tasks. At every step of the execution, if multiple tasks can be executed, there is a choice. For example, for the `Buffer` process (figure 4.1), at every step, one can obtain the next state by executing either the input task A_i or the output task A_o (provided the state is non-null). We do not want to make assumptions about the relative frequency at which the two tasks are executed, but we would like to rule out an execution where the output task is *never* executed. Similarly, for the `Merge` process (figure 4.3), while the exact order in which the values arriving on the two input channels are merged is arbitrary by design, it is natural to assume that all these values eventually appear on the output channel. For the shared objects such as `AtomicReg` and `Test&SetReg`, if multiple processes are competing to write to them, the asynchronous model of computation allows them to succeed in an arbitrary order, possibly one process executing multiple writes before another process gets to execute a single write. However, if a process is denied a chance to write successfully forever, then no meaningful computation can occur. Hence, we would like to assume that a read or write operation by a process on a shared register is not delayed forever.

Infinite Executions

The standard mathematical framework for capturing the informal assumption that *execution of a task can be delayed arbitrarily long, but not forever*, requires us to consider *infinite* executions. An infinite execution, also called an

ω-execution, of a process P starts in one of the initial states and has an infinite sequence of states such that every state in this sequence is obtained from the previous one by executing one of the actions of the process.

Let us revisit the process `AsyncInc` of figure 4.2: it has two tasks A_x and A_y that are always enabled and increment the variables x and y, respectively. Consider the following ω-execution of the process `AsyncInc`:

$$(0,0) \xrightarrow{A_x} (1,0) \xrightarrow{A_x} (2,0) \xrightarrow{A_x} (3,0) \xrightarrow{A_x} (4,0) \cdots$$

where we have labeled each internal action by the task that was executed. In this ω-execution, the next state is always obtained by executing the internal task A_x. We will say that such an ω-execution is *unfair* to the task A_y: at every step the task A_y is enabled, but it is never executed. It is reasonable to assume that no implementation produces such unfair executions. When we state requirements, we will only require that all fair executions should satisfy the requirements. Consider the process `AsyncInc` and a correctness requirement that *the value of y should not always be zero*. Even though the ω-execution in which the task A_y is never executed violates this requirement, we still want to conclude that the process `AsyncInc` meets this requirement since in all fair executions y is guaranteed to be incremented.

Consider a finite execution of the process `AsyncInc`, say consisting of 1000 steps. Even if all the actions in this execution correspond to incrementing x, and thus the task A_y is enabled at every step without being executed, this is considered a plausible or valid execution of the process. If the desired correctness requirement states that "the value of y should be non-zero when the value x is 1000," we want to conclude that the `AsyncInc` is buggy since the finite execution consisting of executing the task A_x 1000 times demonstrates the reachability of the state $(1000, 0)$ that violates the requirement. If we put a *concrete quantitative bound* on the number of steps for which it is acceptable to ignore an enabled task, but not beyond this bound, then no matter what specific number we choose, it would seem to be an arbitrary assumption about the implementation of the process. That's why fairness is defined to be an assumption about infinite executions: every finite prefix of the unfair execution of `AsyncInc` illustrated above is considered legal, but the infinite execution is unfair. In a sense that can be made mathematically precise, fairness is a property of *limits* of finite executions. As we will study in chapter 5, even though infinite executions is an abstract mathematical concept and seems difficult to reason about at first glance, effective analysis algorithms exist for reasoning about such executions since such reasoning can typically be reduced to analyzing cycles in the graph of reachable states.

Consider an ω-execution of the process `AsyncInc` in which the tasks A_x and A_y are executed in an alternate manner, say 1000 times, but after that only the task A_x is executed indefinitely. In such an infinite execution, the value of y is "stuck" at 1000, but x keeps increasing in an unbounded manner. The

fairness assumption with respect to the execution of the task A_y rules out this execution also. For an ω-execution of the process `AsyncInc` to be fair with respect to the task A_y, it must contain infinitely many actions that increment y. Symmetrically, an ω-execution is considered fair with respect to the task A_x only if it contains infinitely many actions that increment x. In the infinite tree of figure 4.4, a fair ω-execution is an infinite path through the tree that zigzags along left and right branches in an arbitrary manner but is guaranteed to take both left and right branches repeatedly. In particular, for every number n, along every fair execution, we are guaranteed that the value of x will exceed n and the value of y will also exceed n.

For the process `Buffer`, an ω-execution where only the input task A_i is executed at every step is unfair to the output task A_o, and we want to rule out such an execution by assuming fairness with respect to the task A_o. For an ω-execution of `Buffer` to be fair with respect to its output task, it must contain infinitely many output actions. Again, when we state requirements such as *a message is eventually delivered*, we will demand that all fair executions should satisfy the requirements. For the process `Buffer`, we don't make any fairness assumptions about the execution of the input task A_i. Notice that for this particular process, every infinite execution must contain infinitely many input actions: this is because every time `Buffer` executes an output action, the buffer becomes empty, and the next output cannot be produced until another input is received.

Now consider the following infinite execution of the process `Merge` of figure 4.3. It receives a value on the input channel in_1. Then it repeatedly executes the loop in which it receives a value on the input channel in_2 and transfers it to the output channel by executing the task A_o^2. That is, the infinite sequence of tasks it executes is A_i^1, followed by the periodic execution of $A_i^2; A_o^2$. This clearly starves the output task A_o^1 that can transfer the element from the queue x_1 to the output channel, which is enabled at every step but never executed. We again want to rule out such an ω-execution as unfair.

Before we define the notion of fair ω-executions precisely, note that we can require a task to be executed only when it is enabled. An unfair ω-execution is one in which, after a certain point, a task is always enabled but never executed.

Consider another infinite execution of the process `Merge`: it repeatedly executes the loop in which it receives a value on the input channel in_2 and transfers it to the output channel by executing the task A_o^2. We consider this to be a fair execution. The input task A_i^1 is never executed, but this is a plausible scenario, and a bug revealed in such an execution may be a real bug. Demanding repeated execution of an input task would mean that we are making implicit assumptions about the environment. Thus, fairness is assumed only for the tasks that the process controls. If the process `Merge` is composed with another process P whose output channel is in_1, then the fairness with respect to an output task of P corresponding to in_1 can force actions involving the channel in_1. The ω-execution that repeatedly executes A_i^2 and A_o^2 in a loop is also (vacuously) fair

`nat` $x := 0;\ y := 0$
$A_x :\ x := x + 1$
$A_y :\ \texttt{even}(x) \ \rightarrow\ y := y + 1$

Figure 4.18: Asynchronous Process `AsyncEvenInc`

with respect to the output task A_o^1. This is because the queue x_1 is always empty, and thus the task A_o^1 is never enabled.

Strong Fairness

The notion of fairness we have discussed so far corresponds to what is known as *weak fairness*. Weak fairness for a task assumes that if the task is continuously enabled, then it is eventually executed. A stronger assumption is *strong fairness*, which demands that a task that is repeatedly enabled should eventually be executed.

An an illustrative example, consider the process `AsyncEvenInc` shown in figure 4.18. Similar to the process `AsyncInc` of figure 4.2, it has two state variables x and y that are incremented by the internal tasks A_x and A_y, respectively, but now the task A_y is enabled only when the value of x is an even number. Consider the following ω-execution of the process `AsyncEvenInc`:

$$(0,0) \xrightarrow{A_x} (1,0) \xrightarrow{A_x} (2,0) \xrightarrow{A_x} (3,0) \xrightarrow{A_x} (4,0) \cdots$$

During this execution, the status of the task A_y switches between enabled and disabled. As a result, this execution satisfies the condition "if continuously enabled then eventually taken" for both the tasks, and it is weakly fair with respect to both the tasks. However, this execution is not strongly fair with respect to the task A_y: the task A_y is enabled infinitely often but never taken. If we assume that the implementation platform ensures only weak fairness, then such an execution is a possible execution, and it is not guaranteed that y gets incremented. If we assume that the implementation platform ensures strong fairness, then such an execution is not a possible execution, and it is guaranteed that y gets incremented.

Modeling an Unreliable FIFO Link

As another illustrative example, consider the unreliable FIFO buffer modeled by the process `UnrelFIFO` shown in figure 4.19. The input task A_i simply transfers the input message to the internal queue x. The transfer of messages from the queue x to the output channel is done by three tasks. The task A_o^1 transfers a message from the queue x to the output channel correctly dequeuing a message

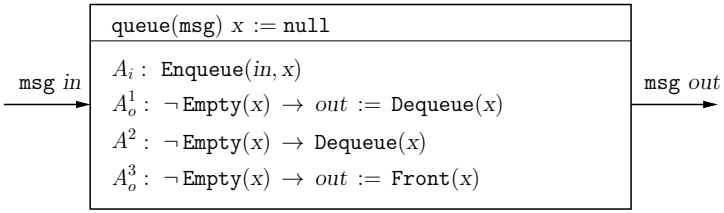

Figure 4.19: Asynchronous Process `UnrelFIFO` for Unreliable Link

and sending it on the output channel. The (internal) task A^2 models a loss of message and simply removes a message from the queue x without transferring it. The task A_o^3 models duplication of messages: it transmits the message at the front of the queue x to the output channel without removing it from the queue. The process thus models a communication link that may lose some messages and may duplicate others. However, it preserves the order and does not reorder messages. The fairness assumptions should ensure that an input message will eventually appear on the output channel.

Consider the following execution of the process `UnrelFIFO`. A message arrives on the channel *in* and is enqueued in the queue x. This message is removed by the task A^2. Since the execution of this task models loss of a message, it does not transmit it on the output channel. Suppose the tasks A_i and A^2 are repeated forever in an alternating manner. This ω-execution is weakly fair with respect to the task A_o^1 that models the correct transfer of messages. This is because every time the input task enqueues the input message in the queue x, the task A_o^1 is enabled, but every time the internal task A^2 removes this message, the task A_o^1 is disabled. Since it does not stay continuously enabled, the weak fairness assumption does not ensure its eventual execution. However, this infinite execution is not strongly fair with respect to the task A_o^1: the task is repeatedly enabled but is never executed. Thus, to capture the informal assumption that repeated attempts to transfer a message will eventually succeed, we should restrict attention to ω-executions that are strongly fair with respect to the task A_o^1.

Fairness Specification

The specification of the process `UnrelFIFO` also highlights that we do not have to assume fairness with respect to all the tasks. In particular, an infinite execution in which the task A_o^3 that duplicates a message or the task A^2 that loses a message is never executed is an acceptable and realistic execution. Losing or duplicating a message is not an active task to be executed and does not need to be executed repeatedly. While the correct functioning of the system could rely on fairness with respect to the task A_o^1, it should not rely on fairness with respect to A^2: a protocol that works correctly *only when* the underlying network

repeatedly loses messages should not be considered correct.

This suggests that the description of an asynchronous process should annotate its output and internal tasks: for some strong fairness is assumed, for some weak fairness is assumed, and some do not have any fairness assumption.

> FAIRNESS ASSUMPTION
>
> An ω-execution of an asynchronous process P consists of an infinite sequence of the form $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \xrightarrow{l_3} s_3 \cdots$ where each s_j is a state of P, s_0 is an initial state of P, and for each $j > 0$, $s_{j-1} \xrightarrow{l_j} s_j$ is an input, an output, or an internal action of P. A task A is *taken* at step j if the transition $s_{j-1} \xrightarrow{l_j} s_j$ corresponds to the execution of the task A. The ω-execution is *weakly fair* with respect to an internal or an output task A, if for all positions j, if the task A is enabled in the state s_j, then there exists a later position $l > j$ such that the task A is either taken at step l or not enabled in state s_l. The ω-execution is *strongly fair* with respect to an output or an internal task A, if for infinitely many indices j, the task A is enabled in state s_j, then for infinitely many indices l, the task A is taken at step l. A *fairness assumption* for an asynchronous process P consists of a subset *SF* of its internal and output tasks demanding strong fairness and a subset *WF* of its internal and output tasks demanding weak fairness. Given such a specification, a *fair ω-execution* of P is an ω-execution that is strongly fair with respect to every task in *SF* and is weakly fair with respect to every task in *WF*.

In the formal definition above, the weak fairness assumption is *if enabled then eventually either taken or disabled*, which is equivalent to *repeatedly disabled or repeatedly taken*. Similarly, the strong fairness assumption is *if repeatedly enabled then repeatedly taken*, which is equivalent to *continuously disabled or repeatedly taken*. Note that any execution that is strongly fair with respect to a task is also weakly fair with respect to that task, but the converse may not hold. In chapter 5, we will specify fairness assumptions using temporal logic, which can help in gaining more insight into the subtle distinction between weak and strong assumptions.

Fairness Assumptions for Mutual Exclusion

To illustrate how to augment a process description with fairness assumptions, let us revisit the solutions to the mutual exclusion problem. First, let us consider Peterson's protocol described in figure 4.13. Each mode-switch corresponds to a task, and let us examine all the tasks of process P_1 (the assumptions for the tasks of the process P_2 are symmetric). There are no fairness assumptions regarding the mode-switches out of the mode `Idle`. This means that the protocol does not assume how long a process waits in the mode `Idle` and does not rely on whether a process requests to enter the critical section repeatedly. The mode-switch out

of the mode `Try1` represents an output action by the process P_1, and we assume weak fairness for this task. This rules out an infinite execution in which the process P_1 is waiting in the mode `Try1` to execute the statement $turn := 1$ while only the process P_2 is being executed repeatedly. The conditional mode-switches out of the modes `Try2` and `Try3` correspond to testing the values of the shared registers. Such read actions are output actions of the corresponding shared register process `AtomicReg` shown in figure 4.10. We assume weak fairness for the output tasks A^1_r of the registers $flag_2$ and $turn$. This ensures that the process P_1 cannot just stay in the mode `Try2` waiting to read $flag_2$ while only the process P_2 is executed repeatedly. There are no fairness assumptions about the self-loop on the mode `Crit`, as we don't want to rely on the process P_1 staying in the critical section for a specific duration. But we do want to assume that it eventually does leave the critical section (otherwise the process P_2 will be blocked forever), and hence we assume weak fairness for the mode-switch from the mode `Crit` to `Idle`. Note that in each case, weak fairness suffices since each task, once enabled, stays enabled until it is executed.

Now let us consider the protocol of figure 4.16. As in the case of Peterson's protocol, we don't make any fairness assumptions about the self-loops on the modes `Idle` and `Crit` and the switch from the mode `Idle` to `Crit`. Weak fairness is assumed for the switch from the mode `Crit` to `Idle` to capture the assumption that a process eventually does exit its critical section. Regarding the conditional switch out of the mode `Try` that tests the value of the shared register *free*, note that this corresponds to output actions of the shared register, and thus the fairness assumptions should be added to the description of the process corresponding to *free* (see the description of the `Test&SetReg` process in figure 4.15). We assume *strong fairness* for the four output tasks A^1_{ts0}, A^1_{ts1}, A^2_{ts0}, and A^2_{ts1}. Note that each task has a guard, and as the value of the shared register changes, each of the tasks can switch between being enabled and being disabled. Strong fairness with respect to the task A^1_{ts0} ensures that if it is the case repeatedly that the process P_1 is in the mode `Try` and the register is 0, then the task A^1_{ts0} must eventually be executed, resulting in a synchronization that returns the value 0 to the process P_1. Thus, it cannot happen that the process P_1 waits in the mode `Try` indefinitely while the process P_2 enters and exits the critical section repeatedly. Note that weak fairness for the task A^1_{ts0} will not rule out such a scenario.

Correctness under Fairness Assumptions

When proving liveness requirements of an asynchronous process with fairness assumption, we can restrict attention only to fair ω-executions. For example, for the process `Merge`, weak fairness is assumed for the output tasks A^1_o and A^2_o. With such a fairness assumption, if it processes the input $in_1\,?\,v$ at any step, then it is guaranteed that at some future step it will produce the output $out\,!\,v$. This is because in every fair execution, if the ith action processes the input $in_1\,?\,v$, then the message v will be added to the queue x_1. Once the queue x_1

has a message, the output task A_o^1 stays enabled at least until this message has been transmitted on the output channel. The weak-fairness assumption for the output task A_o^1 ensures that it will be executed: if the queue x_1 contains multiple messages ahead of v, then they will all be eventually transferred, finally along with v itself. The desired requirement does not hold for unfair executions, but such executions are merely an artifact of modeling definitions and not indicative of a violation in a real implementation.

For the mutual exclusion protocols of figures 4.13 and 4.16, in every fair execution, if the process P_1 wants to enter the critical section, then it will eventually enter the critical section.

The choice of fairness assumptions clearly affect the requirements that an asynchronous process satisfies. Let us illustrate this by considering different requirements for the processes `AsyncInc` and `AsyncEvenInc`:

- The requirement φ_1 states that "the value of x eventually exceeds 10." The process `AsyncInc` does not satisfy this requirement in absence of fairness assumptions but does satisfy this requirement if weak fairness for the task A_x is assumed. Similarly, the process `AsyncEvenInc` does not satisfy this requirement in absence of fairness assumptions but does satisfy this requirement if weak fairness for the task A_x is assumed.

- Consider the requirement φ_2 that "the value of y eventually exceeds 10." The process `AsyncInc` does not satisfy this requirement in absence of fairness assumptions but does satisfy this requirement if weak fairness for the task A_y is assumed. The process `AsyncEvenInc`, in contrast, does not satisfy this requirement in absence of fairness assumptions, and does not satisfy this requirement if only weak fairness is assumed, but it does satisfy this requirement if strong fairness for the task A_y is assumed.

- The requirement φ_3 states that "the value of y eventually exceeds the value of x." The process `AsyncInc` does not satisfy this requirement. The requirement is still not satisfied even if we assume fairness for the two tasks. In particular, the infinite execution where we first execute the task A_x, then execute the task A_y, and repeat this pattern is fair to both the tasks, and along this execution, the condition $y \leq x$ holds in every state (and thus the requirement φ_3 is violated). By a similar argument, the process `AsyncEvenInc` does not satisfy the requirement φ_3 with or without any form of fairness assumptions.

Fairness assumptions only ensure eventual execution of tasks and cannot enforce any specific pattern in relative frequencies of executions of different tasks. If such a pattern is required, as is the case in the requirement φ_3, then the coordination logic within the system must be modified to meet this requirement.

Exercise 4.10: Let us revisit the asynchronous process `Split` that you designed in exercise 4.2. Suppose we want to capture the assumption that the

distribution of messages among the two output channels should be, while unspecified, fair in the sense that if infinitely many messages arrive on the input channel *in*, then both output channels out_1 and out_2 should have infinitely many messages transmitted. How would you add fairness assumptions to your design to capture this? If you are using strong fairness, then argue that weak fairness would not be enough (that is, describe an infinite execution that is weakly fair but the split of messages is not fair as desired). ■

Exercise 4.11: By modifying the description of the process `UnrelFIFO` of figure 4.19, construct a precise specification of the process `VeryUnrelFIFO`, which, in addition to losing and duplicating messages, can also reorder messages. What would be natural fairness assumptions for the modified process? ■

Exercise 4.12: Consider the modified version of Peterson's mutual exclusion protocol shown in figure 4.17. What fairness assumptions should be added to this description? With these fairness assumptions, does the protocol satisfy the requirement that *if a process wants to enter the critical section, then it eventually will enter the critical section?* ■

Exercise 4.13: Consider an asynchronous process P with two variables x and y, both of type `nat`, with x initialized to 0 and y initialized to 2. The behavior of the process is described by two tasks. The task A_1 is always enabled, and its update code $x := x + 1$. The task A_2 is always enabled, and its update code is $y := x + y$. Answer each of the questions below with a brief justification. When adding fairness assumptions, clearly specify whether you are using strong fairness or weak fairness and for which task.

1. Is it guaranteed that the value of x eventually exceeds 5? If not, is there a suitable fairness assumption for the two tasks under which this guarantee holds?

2. Is it guaranteed that the value of y eventually exceeds 5? If not, is there a suitable fairness assumption for the two tasks under which this guarantee holds?

3. Is it guaranteed that at some step in the execution the values of x and y become equal? If not, is there a suitable fairness assumption for the two tasks under which this guarantee holds?

■

4.3 Asynchronous Coordination Protocols

In a network of processes communicating asynchronously, in each step a single process executes a computation step, and such a step can either receive an input value on an incoming channel or send an output value on an outgoing channel. As a result, algorithms for solving coordination problems cannot

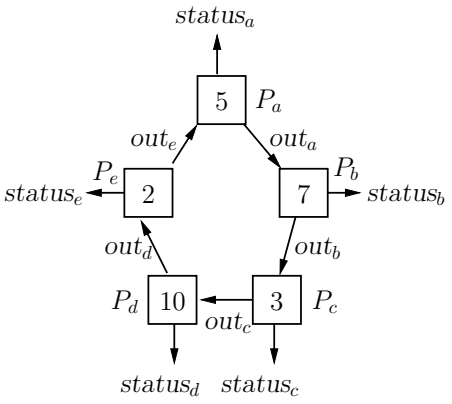

Figure 4.20: An Asynchronous Network with Ring Topology

proceed in lock-step rounds as in the synchronous case. We illustrate some of the design challenges using three classical problems: electing a leader in a ring of processes, implementing reliable communication using unreliable links, and reaching consensus among two processes using shared objects.

4.3.1 Leader Election

Let us revisit the coordination problem of leader election discussed in section 2.4.3, now in the asynchronous setting. Let us assume that the underlying network connects the nodes in a unidirectional ring (see figure 4.20 for an example ring with five nodes). Each node has a unique identifier, and the protocol consists of a strategy for nodes to exchange messages so that eventually a single node declares itself to be the leader, with the remaining nodes declaring themselves to be followers.

We model each network node as an asynchronous process P. The input channel *in* receives identifiers sent by the unique predecessor of P in the ring, and the output channel *out* sends identifiers to the unique successor of P in the ring. An internal queue x is used to store messages received on the channel *in*, and the queue y is used to store messages to be sent, which get delivered by the output task on the channel *out* one by one. When the process concludes that it is either the leader or one of the followers, the decision is issued on the output channel *status*. The description of the process is parameterized by the identifier of the corresponding network node, denoted `myID`. We will assume that each identifier is a positive number. To form a ring, we create multiple instances of the process P and compose them together using the asynchronous composition operation. For example, the system corresponding to the ring of five processes

in figure 4.20 is $P_a \mid P_b \mid P_c \mid P_d \mid P_e$, where

$$\begin{aligned}
P_a &= P[status \mapsto status_a][in \mapsto out_e][out \mapsto out_a][\texttt{myID} \mapsto 5],\\
P_b &= P[status \mapsto status_b][in \mapsto out_a][out \mapsto out_b][\texttt{myID} \mapsto 7],\\
P_c &= P[status \mapsto status_c][in \mapsto out_b][out \mapsto out_c][\texttt{myID} \mapsto 3],\\
P_d &= P[status \mapsto status_d][in \mapsto out_c][out \mapsto out_d][\texttt{myID} \mapsto 10],\\
P_e &= P[status \mapsto status_e][in \mapsto out_d][out \mapsto out_e][\texttt{myID} \mapsto 2].
\end{aligned}$$

Our goal is to complete the description of the process P so that when multiple instances of this process are composed to form a ring, the following requirements are met: (1) every process eventually terminates, that is, there is no infinite execution of the protocol; and (2) in every terminating execution, exactly one process has output the value `leader` on its output channel *status*, and the remaining processes have output the value `follower` on their output channels *status*.

Recall that in the synchronous solution, if N is the total number of nodes in the network, then assuming the network to be strongly connected, a node could infer that its identifier has reached all the nodes in the network within N rounds. In the asynchronous case, no such inference can be drawn, as nodes are executing at independent speeds, and there is no concept of a *round* that involves all the processes. A node can infer that the message it sent to its successor on the output channel *out* has propagated to all the processes in the ring only when it receives an appropriate input message from its predecessor. Consequently, a process does not need to know the number of processes.

One possible solution to the asynchronous leader election in a ring is obtained by adopting the flooding algorithm of section 2.4.3 that elects the process with the highest identifier. We describe a more interesting algorithm that reduces the number of messages that are exchanged. If the ring contains N processes, then the algorithm to be discussed will generate only about $N \log N$ messages, as opposed to N^2 messages that the flooding algorithm can generate. As it turns out, $N \log N$ is also a *lower bound* on the number of messages that have to be exchanged in order to elect a leader among processes communicating asynchronously over a ring network.

The algorithm is shown in figure 4.21. The input task A_i is always enabled and simply stores each input message in the internal queue x. The output task A_o outputs pending messages from the queue y to the output channel *out* and can be executed at any time provided the queue y is not empty.

The core computation of the process is described as an extended-state machine. Initially, the mode is undecided. Once a decision is reached, the process switches to the leader or the follower mode, and during this switch, the decision is output on the channel *status*. In the follower mode, the process simply relays messages from its input queue x to its output queue y, and this is captured by the internal task A_7.

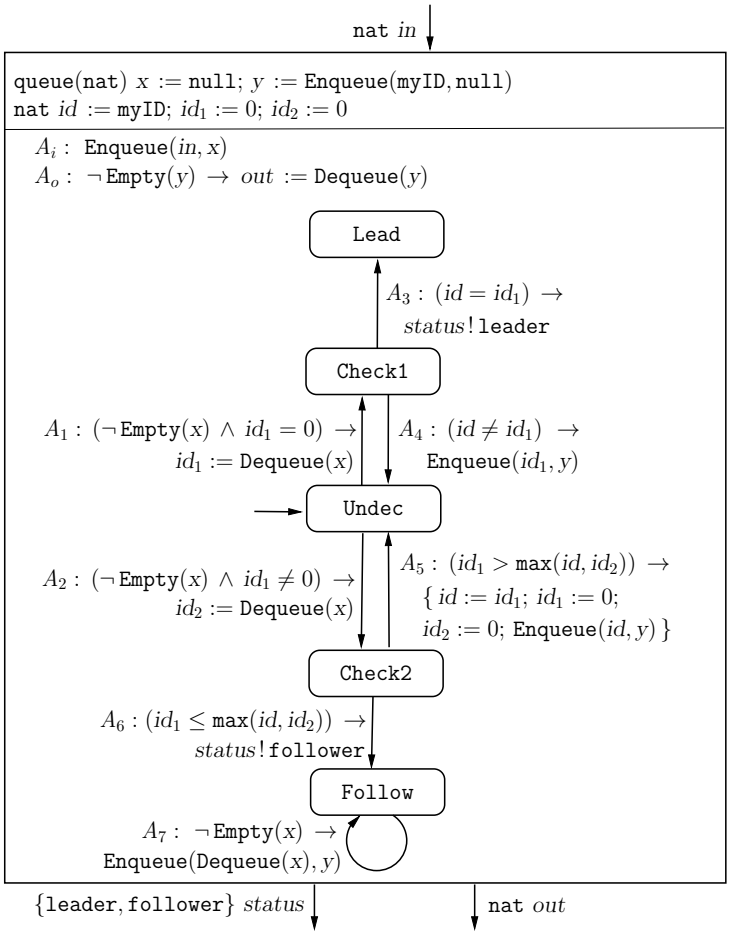

Figure 4.21: Asynchronous Leader Election in a Ring

The execution of the algorithm progresses in phases, and in each phase, the number of undecided processes decreases at least by a factor of 2, until only one process remains undecided, which then becomes the leader.

Initially, each process sends its identifier to two successive processes along the ring. To achieve this, each process first sends its identifier, as well as the first input message it receives, on the output channel. When a process receives two messages on the input channel, it knows its own identifier, captured by the variable *id*, the identifier of its predecessor, captured by the variable id_1, and the identifier of the predecessor's predecessor, captured by the variable id_2.

Initially, the variable *id* is set to `myID`, the unique positive number associated with the process. The variables id_1 and id_2 are set to 0. The identifier is

enqueued in the outgoing queue to be transmitted to the next process. Then the process waits until there is a message to be processed in the incoming queue x. When the value of id_1 is 0, the next message to be processed, which is at the front of the queue x, is the value from the predecessor, and id_1 is set to this value by dequeuing x, and the process switches to the mode `Check1` (see task A_1). If the value of id_1 is non-zero, the next input message to be processed is the identifier of the predecessor's predecessor. The task A_2 dequeues this message, stores it in the variable id_2, and switches to the mode `Check2`.

In the mode `Check1`, the process checks the value of the predecessor's identifier stored in id_1. When this value equals the current value of id, the process has won the election. In this case, the process outputs the value `leader` on the channel *status*, and the mode is updated to `Lead` (see task A_3). Otherwise, when the predecessor's identifier is different from the current value of id, this value is enqueued in the outgoing queue y to be sent to the successor process, and the mode switches to `Undec` (see task A_4).

Once the process has received the values of both id_1 and id_2, in the mode `Check2`, it compares these two identifiers with the value of id. If id_1 is the highest among these three identifiers, then the process continues to remain undecided, adopting the value of id_1 as its own identifier, and initiates a new phase starting in the mode `Undec` (see task A_5). If id_1 is not the highest among these three identifiers, then the process outputs the decision `follower` on the channel `status` and switches to the follower mode `Follow` (see task A_6). Subsequently, this process will only relay messages without examining them.

Note that every process is repeating the same computation. Suppose for a process P, $id = m_0$, $id_1 = m_1$, and $id_2 = m_2$. The process P will continue to stay undecided if both $m_1 > m_0$ and $m_1 > m_2$. Consider the predecessor P' of P. Then for the process P', its own identifier, that is, the value of its id variable is m_1, and the identifier of its predecessor, that is, the value of its id_1 variable, is m_2. This guarantees that *if P decides to stay undecided adopting m_1 as its identifier, P' will become a follower.* Consequently, the number of processes that continue to stay undecided is at most half of the current number of undecided processes. Furthermore, the number of processes that continue to stay undecided is at least 1: among all the undecided processes, the successor of the process with the highest identifier is guaranteed to stay undecided.

For the example network shown in figure 4.20, for the process P_c with the original identifier 3, the values of id, id_1, and id_2 in the first phase will be 3, 7, 5, respectively, and it will continue to the next phase as an undecided process, with 7 as its identifier. For the process P_b with the original identifier 7, the values of id, id_1, and id_2 in the first phase will be 7, 5, and 2, respectively, and it will become a follower. After the first phase, only processes P_c and P_e will be undecided, with modified identifiers 7 and 10, respectively.

When a process continues to stay undecided, it repeats the protocol again. It sends its current identifier (which was adopted from its predecessor in the preceding round) and the next input message on its output channel. After receiving

two input messages, it examines the relative ordering of its identifier and the identifiers of its two (undecided) predecessors, making decisions as before. That is, in every subsequent phase, the current ring with the reduced number of undecided processes repeats the same protocol, thereby again reducing the number of undecided processes by at least half. The presence of follower processes does not influence the logical argument since they are simply relaying messages.

When an undecided process receives an input message that is equal to its current identifier, it can conclude that it is the only undecided process and proceeds to declaring itself as the leader. Note that even though this identifier is guaranteed to be the highest among all the original identifiers, it is not the original identifier of this leader process.

Continuing our example from figure 4.20, during the second phase, for the process P_c, the values of id, id_1, and id_2 will be 7, 10, and 7, respectively, and it will continue to the next phase as the only undecided process adopting the identifier 10. In the third phase, the first message it sends will come back to it as the predecessor's identifier, with all other processes simply relaying this message. This will cause the process P_c to declare itself as the leader.

The formal correctness argument is complicated by the fact that the phases are not synchronized, and at any given time, neighboring processes may be executing different phases. In each phase, each process sends at most two messages. If the ring contains N processes, then each phase contributes at most $2N$ messages, and the number of phases is at most $\log N$, leading to an overall bound of $2N \log N$ messages.

In this protocol, no process ever sends messages repeatedly. Thus, no infinite execution is possible, and as a result, correctness does not require any fairness assumptions.

Exercise 4.14: For the leader election protocol of figure 4.21, consider a ring with 16 nodes where the identifiers of the processes in order are: 25, 3, 6, 15, 19, 8, 7, 14, 4, 22, 21, 18, 24, 1, 10, 23. Which process will be elected as the leader? ■

Exercise 4.15*: For the leader election protocol of figure 4.21, describe the best- and worst-case scenarios: (a) describe the scenario in which only one node will stay active after the first phase, and (b) describe the scenario in which the protocol will need $\log N$ phases before the election. ■

4.3.2 Reliable Transmission

Given an unreliable communication medium, how can we implement a reliable FIFO link that delivers each message exactly once in the order received? More specifically, we want to design processes P_s and P_r so that the composite system shown in figure 4.22 acts as a reliable FIFO buffer with respect to its input and output channels using two instances of the unreliable communication link

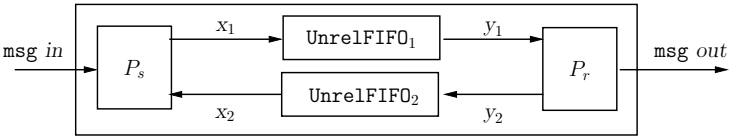

Figure 4.22: The Block Diagram for Reliable Communication

`UnrelFIFO` (see figure 4.19). The process P_s acts as an interface for the sender, and the process P_r acts as an interface for the receiver. The unreliable link `UnrelFIFO`$_1$ transfers messages from the process P_s to P_r, and the unreliable link `UnrelFIFO`$_2$ transfers messages from the process P_r to P_s.

Alternating Bit Protocol

To deliver a message that the process P_s receives on its input channel *in*, it may need to send the message repeatedly to the process P_r since the link `UnrelFIFO`$_1$ may lose messages, and the process P_r needs to send an explicit acknowledgment back to the process P_s notifying successful delivery. The acknowledgment also needs to be sent repeatedly to ensure eventual successful delivery to account for lost messages. A key design challenge is to match messages with acknowledgments, in the presence of potential duplication of messages as well as duplication of acknowledgments. One classical solution for this purpose is the *alternating bit protocol* that synchronizes the sender and the receiver processes using a Boolean tag bit that alternates.

The sender interface process P_s is shown in figure 4.23. It maintains a queue x of messages that it receives on its input channel *in*, and it is processed by the input task A_i. The state variable *tag* is a Boolean variable that is initially 1. When the process P_s sends the message at the front of its internal queue x to the receiver process P_r using the unreliable FIFO link on the channel x_1, it augments the message with the current value of *tag* and does not remove the message from the queue x. The output task A_1 may get executed repeatedly. When the sender process P_s gets an acknowledgment on the channel x_2 in the form of a tag bit from the receiver, it checks whether the received tag matches its own tag; if this check succeeds, it removes the message from its queue x and toggles the tag. The processing of acknowledgment tags is modeled by the input task A_2. The toggling of the tag will cause the next message in the queue x to be sent, possibly repeatedly, on the output channel x_1 augmented with this updated tag. Note that the task A_2 is always enabled, but it does not modify the state if the incoming acknowledgment tag does not match the expected tag. The fairness assumption consists of weak fairness for the output task: once a message is enqueued in the queue x, the task A_1 stays enabled and should eventually be executed sending the first message on the channel x_1.

The receiver process P_r is shown in figure 4.24. The messages it receives are

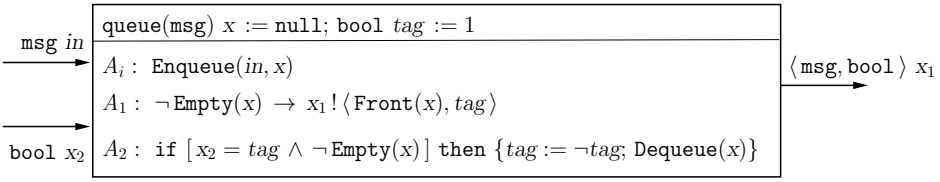

Figure 4.23: The Sender Process for the Alternating-bit Protocol

stored in the internal queue y. The process also maintains a Boolean-valued tag state variable, which is initially 0. Note that the initial values of the tag bits of the sender P_s and the receiver P_r are complements of each other: initially and at every step, the sender P_s expects the incoming tag from the receiver to be the same as its internal tag, whereas the receiver expects the incoming tag from the sender to be the complement of its internal tag. When the receiver process P_r receives a message on the input channel y_1, it checks whether the tag of the incoming message is the complement of its own tag. If so, the incoming message is considered a new message, and it is added to the queue y. This is captured by the input task A_1, where the primitives `First` and `Second` are used to retrieve the two fields of the incoming message. Note again that the task A_1 is always enabled, and if the incoming tag is not what it expects, the incoming message is simply ignored. Messages in the queue y are transmitted on the output channel by the output task A_o. The receiver P_r also repeatedly sends the current value of its tag to the sender process P_s as an acknowledgment on the channel y_2 (captured by the output task A_2). To ensure eventual delivery on both the output channels, the fairness assumption consists of weak fairness for both the output tasks A_2 and A_o. Since these tasks are not disabled by competing actions, we don't need strong fairness.

The following scenario describes how the protocol executes. Suppose the process P_s receives a message, say m_1, on its input channel *in*. Then it will repeatedly send the message $(m_1, 1)$ to the process P_r using the unreliable channel. Each such message may be lost or duplicated. Meanwhile, the process P_r can repeatedly send the tag bit 0 to the process P_s, but P_s will ignore all such acknowledgments. The first time the message $(m_1, 1)$ is successfully delivered to the process P_r on the channel y_1, the process P_r will change its tag to 1 and enqueue m_1 in its output queue y. The message m_1 will eventually be transmitted on the output channel *out*. Additional copies of the message $(m_1, 1)$ received on the channel y_1 will be ignored by the process P_r since its tag is now 1: it will recognize the next message as a fresh message only when the message is tagged with 0. The process P_r will repeatedly send the tag 1 to P_s as an acknowledgment on the channel y_2. Each such message again may be lost or duplicated, but eventually the process P_s will receive the tag 1 on the channel x_2. At this point, the process P_s will remove the message m_1 from its internal queue x and toggle its *tag* variable to 0. If additional messages are received on

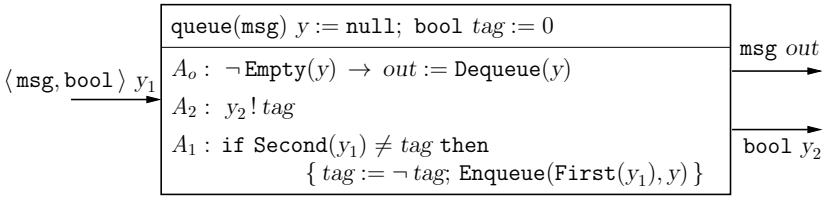

Figure 4.24: The Receiver Process for the Alternating-bit Protocol

the channel *in* during this period, then they get enqueued in the queue x, and if m_2 is the next pending message, then the process P_s will start sending the message $(m_2, 0)$ to P_r on the channel x_1. If the process P_s receives additional tag messages 1 on the channel x_2, then it will ignore them. The message m_2 will be dequeued by the process P_s from its queue x only when it receives the tag 0.

Exercise 4.16: Suppose we know that the communication link from the receiver back to the sender is reliable. How would you modify the alternating-bit protocol to take advantage of this? That is, design simplified versions of the processes P_s and P_r so that the composite system shown in figure 4.22 acts a reliable FIFO buffer when the process $\texttt{UnrelFIFO}_2$ is replaced by the process `Buffer`. ■

Exercise 4.17*: Consider the description of the process `VeryUnrelFIFO` designed in exercise 4.11 of an unreliable link that may lose messages, duplicate messages, and reorder messages. First, show that the alternating-bit protocol does not work correctly if we replace each instance of `UnrelFIFO` with a corresponding instance of the process `VeryUnrelFIFO`. How would you modify the processes P_s and P_r so that reliable communication is guaranteed even in the presence of this added complication of reordering? Argue that the modified protocol works correctly. Hint: a Boolean-valued tag is not enough, and messages need to be tagged with a counter variable of type `nat`. ■

4.3.3 Wait-Free Consensus *

To see how the choice of atomic primitives supported by shared objects impacts the ability to solve distributed coordination problems, let us consider the classical problem of *wait-free two-processes consensus*. Each process starts with an initial preference that is known only to itself. The processes want to communicate and arrive at a consensus decision value. This problem has been posed in many different forms, for instance, requiring two Byzantine Generals in charge of collaborating armies separated by the enemy army to exchange messengers to arrive at a mutually agreed time of attack. The core coordination problem of reaching agreement in the presence of unpredictable delays is central to many distributed computing problems.

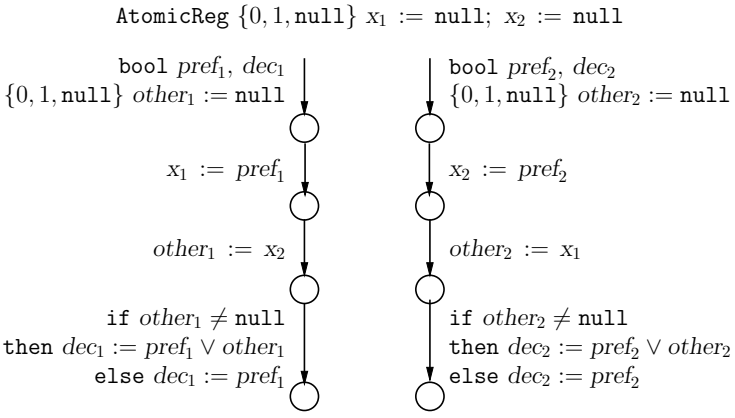

Figure 4.25: First Solution to Two-Process Consensus Using Atomic Registers

Problem Description

More specifically, we have two asynchronous processes, say P_1 and P_2, each of which has an initial Boolean value, denoted v_1 and v_2, respectively, unknown to the other. The processes want to arrive at Boolean decision values d_1 and d_2, respectively, so that the following three requirements are met: (1) the decision values d_1 and d_2 of the two processes are identical, (2) the decision value must be equal to one of the initial values v_1 or v_2, and (3) at any time, if tasks involving only one of the processes are repeatedly executed, this process should reach a decision. The first requirement, called *agreement*, means that the two processes should come to a common decision even if they start with different preferences. The second requirement, called *validity*, says that if both prefer the same value, then they must decide on that value. This requirement rules out input-oblivious solutions such as: *both decide on 0 no matter what the initial preferences are.* The third requirement, called *wait freedom*, ensures that a process can decide on its own without having to wait indefinitely for the other.

Suppose we want to design the processes P_1 and P_2 so that they communicate using shared objects such as atomic registers and test&set registers. In the composite system, every action then will be either an internal action of one of the processes or will be a primitive operation by one of the processes involving one of the shared objects.

Incorrect Solutions Using Atomic Registers

Correctness requirements for the problem and challenges in designing a correct solution can be best illustrated using protocols that meet only some of the requirements. As a first attempt, consider the solution shown in figure 4.25. The solution uses two shared atomic registers x_1 and x_2, each of which is an

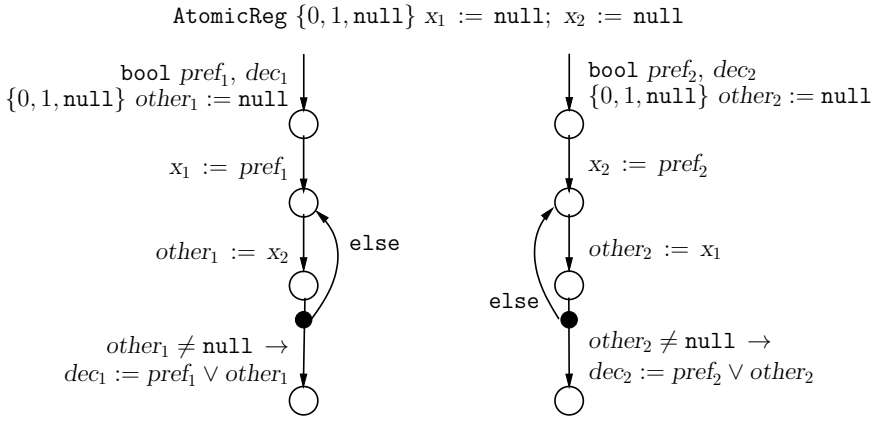

Figure 4.26: Second Solution to Two-Process Consensus Using Atomic Registers

instance of the `AtomicReg` process shown in figure 4.10. Here, the set `val` of values is $\{0, 1, \texttt{null}\}$, and the initial value `initVal` for each of the shared registers is `null`.

The initial preference of the process P_1 is stored in its state variable $pref_1$, and its objective is to set the decision variable dec_1 to the decision value on termination. The process P_1 first writes its preference to the shared register x_1, and analogously, the process P_2 writes its preference (captured by the initial value of its state variable $pref_2$) to the shared register x_2. The process P_1, after writing its preference to x_1, reads the shared register x_2 into its internal state variable $other_1$. Since the execution of the two processes proceeds asynchronously, when the process P_1 executes the action of reading x_2, there is no guarantee that the process P_2 has already written its preference to x_2. To account for this possibility, the process P_1 checks whether the value it reads is `null`. If it is `null`, then it decides on its own preference; otherwise, it knows the preference of the process P_2 and decides on the logical disjunction of the two preferences. The process P_2 is symmetric.

The protocol, however, is buggy. Suppose the initial preferences of P_1 and P_2 are 0 and 1, respectively. Consider the execution in which we first execute only the tasks involving the process P_1 until it finishes and then execute all the actions of the process P_2. In this scenario, when the process P_1 reads x_2, its value is still `null`, and hence P_1 decides on its own preference, 0. However, when the process P_2 reads x_1, it receives the value 0, and it decides on the disjunction of the two initial preferences, namely, 1. Thus, the protocol violates the *agreement* requirement. Observe that the protocol does meet the requirements of validity (if both initial preferences are 0, both will decide 0; and if both initial preferences are 1, both will decide 1 no matter in which order the two processes execute

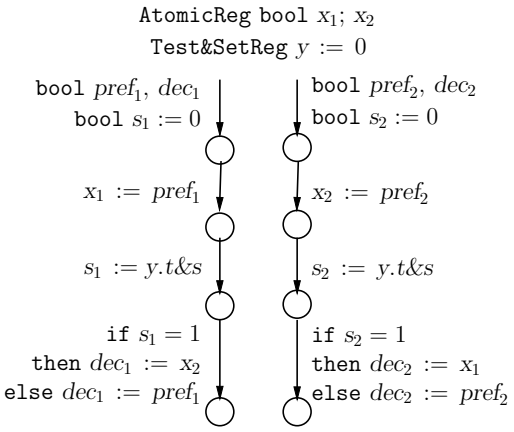

Figure 4.27: Solution to Two-Process Consensus Using a Test&Set Register

their respective actions) as well as wait freedom (each process executes exactly three actions before terminating and enabling of each of these actions does not depend on the other process).

We can try to "fix" the protocol by requiring each process to wait until it knows the preference of the other process. Figure 4.26 shows the revised protocol: after the process P_1 reads the shared register x_2, if its value is `null`, it loops back and reads x_2 again. In the revised version, the requirement of agreement is satisfied since both processes decide only after they know both the preferences. The requirement of validity also holds. However, the requirement of wait freedom is violated. The reason is that if, say the process P_2, has not yet executed its write to x_2, then the process P_1 will repeatedly read x_2 and will not be able to reach a decision on its own.

Solution Using Test&Set Registers

It is possible to solve consensus using a single test&set register. Consider the following protocol that uses two Boolean atomic registers x_1 and x_2 and a test&set register y (see figure 4.27). The initial values of the registers x_1 and x_2 do not matter, and y is initially 0. The process P_1 executes the following sequence of actions, with the process P_2 following a symmetric protocol. The process P_1 first writes its own preference to the atomic register x_1. Then it executes a *test&set* operation on the register y. If the value returned (stored in the state variable s_1) is 0 (implying that the register y was 0 before the process P_1's *test&set* operation was executed), then P_1 goes ahead and decides on its own preference. If the value received by the *test&set* operation on y is 1, then the process P_1 concludes that the other process P_2 had already executed its *test&set* operation successfully, and hence the register x_2 must contain the preference of the process P_2. The process P_1 then proceeds to read x_2 and decides on the

value it contains. In summary, each process publishes its preference in a shared atomic register, executes *test&set* to resolve contention, and, based on the result of this test, decides whose preference to adopt. Each process executes a fixed number of actions and thus can decide without waiting for the other.

Impossibility of Solving Consensus Using Atomic Registers

The key to the correct solution of figure 4.27 is the use of the atomic operation *test&set* that updates the register and returns its old value without interference from other processes. If we are required to use only atomic registers, where the operations of reading and writing a register are decoupled, then no matter how many shared registers the protocol employs and how many values each such register can hold, there is no solution that satisfies all three requirements of agreement, validity, and wait-freedom.

For the protocol of figure 4.27, consider the initial state in which $pref_1 = 0$ and $pref_2 = 1$. Such a state is called *uncommitted* in the sense both decisions are still feasible: starting from such a state, there is a possible execution in which both processes end up deciding on 0, and there is another possible execution that results in both processes deciding on 1. The first step of the proof below is to establish that such an uncommitted state must exist in every protocol that correctly solves the consensus problem. For the protocol of figure 4.27, starting with $pref_1 = 0$ and $pref_2 = 1$, consider the state after both the processes have taken one step each and have written their respective preferences to the variables x_1 and x_2. This state is still uncommitted, but if the next step is by P_1, then the final decision is guaranteed to be 0 (irrespective of how the execution proceeds subsequently), while if the next step is by P_2, then the final decision is guaranteed to be 1. A key part of the proof is to establish that in every consensus protocol, there must be such an uncommitted reachable state such that the next step is the deciding factor for the final decision. For the protocol of figure 4.27, this critical step involves the *test&set* operation on a shared register. The proof concludes by showing that if all that a process can do in one step is either read or write an atomic register, then such a step cannot be the critical deciding factor, and thus the problem cannot be solved using only atomic registers.

Theorem 4.1 [Impossibility of Consensus using Atomic Registers] *There is no protocol for two-process consensus such that (1) the processes communicate using only atomic registers as shared objects, and (2) the protocol satisfies all three requirements of agreement, validity, and wait-freedom.*

Proof. Suppose there exists a solution to the two-process consensus problem using only atomic registers. Consider the transition system T that corresponds to the system obtained by composing the two processes P_1 and P_2 and all the atomic registers that the protocol uses. A state s of T consists of the internal states of the two processes and the states of all the shared atomic registers. A single transition of T is either an internal action of one of the two processes or

a read action involving one of the processes and one shared register, or a write action involving one of the processes and one shared register.

Starting from a given state s, many executions are possible, but each one is finite and ends in a state where both processes have decided. Let us call a state s *uncommitted* if both decisions 0 and 1 are still possible: there is an execution starting in the state s in which both processes decide 0, and there is another execution starting in the state s in which both processes decide 1. A state is called 0-*committed* if in all executions starting in the state s both processes decide 0 and 1-*committed* if in all executions starting in the state s both processes decide 1.

Let us call two states s and t P_2-*indistinguishable* if the internal state of the process P_2 is the same in both states s and t, and the state of each of the shared registers is also the same in both states s and t. That is, the states s and t look the same from the perspective of the process P_2: if P_2 can execute an action in the state s, then it can execute the same action in the state t.

As a first step toward the proof, we first establish the following:

> Lemma 1. If two states s and t are P_2-indistinguishable, then it cannot be the case that the state s is 0-committed and the state t is 1-committed.

The wait-freedom requirement means that starting in any state, if we execute actions involving only one of the two processes, then it must reach a decision. Consider two states s and t that are P_2-indistinguishable such that the state s is 0-committed. In the state s, if we let only the process P_2 execute actions, then it will eventually reach a decision, and this must be 0 by the assumption that all executions starting in the state s lead to the decision 0. Now consider what happens if we let only the process P_2 take steps starting in the state t. Since the states s and t look the same as far as the process P_2 can tell, it can execute the same sequence of actions and reach the same decision 0. Thus, the state t cannot be 1-committed.

The next step in the proof is the following lemma:

> Lemma 2: There exists an uncommitted initial state.

Consider an initial state s in which the preferences v_1 and v_2 of the two processes are different, say 0 and 1, respectively. We claim that this state must be uncommitted. If not, suppose it is 0-committed. In the state s, the process P_2 has preference 1. Neither the initial values of the shared registers nor the initial values of the state variables belonging to P_2 reflect the initial preference of the process P_1. Consider the initial state t in which the initial values for the shared registers and the state variables of P_2 are identical to those in the state s but the initial state of the process P_1 is chosen so that its initial preference is 1. That is, the only difference in the states s and t is in the initial preference of the process P_1. The states s and t are P_2-indistinguishable by construction. By

Lemma 1, we can conclude that the state t cannot be 1-committed. But this is a contradiction to the *validity* requirement: in the state t, both preferences are 1, and thus every execution starting in the state t must lead to the decision 1 (otherwise the protocol does not satisfy the validity requirement).

We now proceed to establish that:

> Lemma 3: There exists an uncommitted reachable state s such that all successor states of the state s are committed.

The proof of the lemma is by contradiction. First observe that the protocol cannot terminate in an uncommitted state since both processes are required to reach a common decision upon termination. Then if Lemma 3 does not hold, we can assume that every reachable uncommitted state has an uncommitted successor state. Consider an uncommitted initial state s_0 guaranteed by Lemma 2. Clearly, the state s_0 is reachable, and by assumption, it must have an uncommitted successor, say state s_1. We can repeat this argument again: at every step j, we have a reachable uncommitted state s_j, and by assumption, we can find an uncommitted successor state s_{j+1} extending the execution by one more step. This means that there is an infinite execution in which processes have not reached a decision, a violation of the correctness requirement to reach agreement. It follows that Lemma 3 must hold.

Consider a state s promised by Lemma 3. The state s is uncommitted, that is, both decisions are still possible, but executing one more step by either of the processes commits the protocol to the eventual decision. Without loss of generality, we can assume that there exist actions $s \to s_1$ by the process P_1 using the task A_1 and $s \to s_2$ by the process P_2 using the task A_2, such that every execution starting in the state s_1 ends up with the decision 0, and every execution starting in the state s_2 results in the decision 1 (see figure 4.28). Each action can be an internal action, a reading of a shared register, or a writing of a shared register. To complete the proof, we consider all possible types of tasks for A_1 and A_2 and arrive at a contradiction in each case.

Suppose the task A_1 of the process P_1 corresponds to a reading of a shared atomic register. The execution of such a task does not modify the state of any of the shared objects and does not modify the internal state of the process P_2. Thus, the states s and s_1 are P_2-indistinguishable. Since the two states look the same to the process P_2, it can execute the task A_2 in the state s_1 also, and let the resulting state be t (see figure 4.28). The states t and s_2 are P_2-indistinguishable. But the state s_2 is 1-committed, while the state t is 0-committed, a contradiction to Lemma 1.

The cases when one of the tasks is an internal task and when the task A_2 involves a read action are similar. The interesting remaining case is when both the tasks A_1 and A_2 execute write actions. There are two sub-cases: they both write to the same register and they write to different registers. We will consider the former, leaving the latter as an exercise.

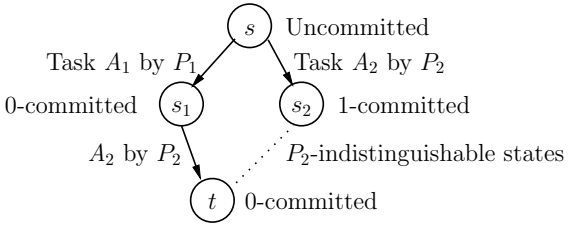

Figure 4.28: Impossibility Result for Consensus Using Atomic Registers

Consider the case when both the processes write to the same atomic register, say x. That is, in the state s, the process P_1 writes some value m_1 to the register x leading to the state s_1, and the process P_2 writes some value m_2 to the same register x leading to the 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. A key observation is that the execution of the task corresponding to writing a register is not influenced by the current value of the register. Thus, in the state s_1, the process P_2 can write the same value m_2 to the register x leading to the state t (see figure 4.28). The writing of the value m_1 by the process P_1 to x has been effectively lost and did not influence what the process P_2 was about to do in the state s. In the states s_2 and t, the internal states of the process P_2 are identical, and so are the states of all the shared registers. Thus, the states s_2 and t are P_2-indistinguishable, the state s_2 is 1-committed, and the state t is 0-committed: a contradiction to Lemma 1. ■

Exercise 4.18: Complete the proof of Theorem 4.1 by considering the remaining case where the task A_1 writes to a shared register x, and the task A_2 writes to a different shared register y. ■

Exercise 4.19: Consider the following solution to the two-process consensus problem in the asynchronous model. The processes use a shared atomic register x and a shared test-and-set register y. The possible values for the register x are *null*, 0, and 1, and the initial value is *null*. The possible values for the register y are 0 and 1, and its initial value is 0. Each process executes the following sequence of steps:

1. Write its initial preference to the register x.
2. Execute a test-and-set operation on the register y.
3. If step (2) returns 0, then decide on its own initial preference.
4. If step (2) returns 1, then read the register x and decide on the value read.

Consider the three requirements for consensus: validity, agreement, and wait-freedom. Which of these requirements are satisfied by this protocol? Justify your answer. ■

Exercise 4.20 *: Consider the generalization of the consensus problem to multiple processes in which each process starts with an initial preference bit and wants to decide on a common Boolean value. The protocol must satisfy the requirements of agreement (all decide on the same value), validity (the decision value must be a preference of one the processes), and wait freedom (if a process takes steps all by itself, then it should reach a decision in finitely many steps without having to wait for the others). Assume that the description of atomic registers and test&set registers described in section 4.2.3 is suitably generalized so that a register can be accessed by multiple processes. Explain why the strategy described in the two-process protocol based on a single `Test&SetReg` register to resolve contention (see figure 4.27) does not generalize to three processes. Try to design a solution to the three-process consensus problem using two `Test&SetReg` registers and show that your attempts fail (note: when the number of processes is three (or more), there is no solution to the consensus problem using only atomic and test&set registers). ■

Exercise 4.21 *: Consider the shared object `StickyBit` that supports read and write operations as in the case of an atomic register, with some modifications. The internal state of a `StickyBit` process can be `null`, 0, or 1 and is initially `null`. The read operation outputs the current value. The write operation has a Boolean (0 or 1) input value associated with it: if the current state is `null`, then the state is updated to the value of write, but if not (that is, if the state is already 0 or 1), then the value stays unchanged. Describe a protocol for solving two-process consensus using a single `StickyBit` object (you may use any number of additional atomic registers as you need). Can you solve consensus for three (or more generally, n) processes using multiple `StickyBit` and `AtomicReg` objects? ■

Exercise 4.22 *: This exercise describes a classical puzzle that requires design of an asynchronous coordination strategy. There are N prisoners who get together initially to decide on a strategy. Then each prisoner is taken to her own isolated cell. A prison guard goes to a cell and takes its prisoner to a room where there is a switch. The switch can either be up or down. The prisoner is allowed to inspect the state of the switch and then has the option of flicking the switch. The prisoner is then taken back to her cell. The prison guard repeats this process infinitely often. The order in which he brings the prisoners to the cell is arbitrary, in particular, there is no bound on how many times one prisoner visits the room with the switch before some other prisoner gets to visit. However, the prison guard guarantees fairness: every prisoner will visit the room infinitely often. At any time, any prisoner can exclaim "I have concluded that every prisoner has visited the room with the switch at least once." Upon such a declaration, if the statement is indeed correct, all prisoners are set free; if the statement is not correct, all prisoners are immediately executed. What strategy should the prisoners use to ensure their eventual freedom? Note that the initial state of the switch is unknown to the prisoners, but as a warm-up, you may consider the same problem but with a known initial state of the switch. ■

Bibliographic Notes

There is a rich history of formal models and distributed algorithms for asynchronous concurrent processes dating back to Dijkstra [Dij65]. The formal model described here is based on the model of I/O automata [LT87, Lyn96].

Different notions of fairness are discussed in [Fra86], and the literature on the model of fair transition systems contains many examples of specification and verification of asynchronous systems with weak and strong notions of fairness [MP91].

The coordination problems of mutual exclusion, consensus, leader election, and reliable communication in the presence of unreliable channels have been core research problems in both distributed computing and formal verification for decades (see for instance, [CM88], [Lyn96], and [Lam02]). The specific results we have discussed include Peterson's two-process mutual exclusion protocol [Pet81], leader election in a ring using $O(N \,.\texttt{log}\, N)$ messages [Pet82], alternating bit protocol for reliable communication [BSW69], and impossibility of consensus using atomic registers [FLP85, Her91].