Chapter 3

Asynchronous Models

3.1 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*. Even within a process, the processing of inputs is decoupled from producing outputs, modeling the assumption that any internal computation takes an unknown, but nonzero, amount of time. Asynchronous models make the design problem more challenging, but are easier to implement on multi-processor machines or on networked distributed platforms.

As an example, consider the `Buffer` process shown in Figure 3.1 that models the asynchronous variation of the synchronous reactive component `Delay` of Figure 1.1. 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 be empty, or contain a Boolean value. This is modeled by the variable x ranging over $\{\texttt{null}, 0, 1\}$. Initially, the buffer is empty, and this is expressed by the initialization expression $x = \texttt{null}$. The key difference between `Delay` and `Buffer` is in the specification of their dynamics. The process `Buffer` has two possible 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, the process can output the buffer state by writing it to the output channel *out* while resetting the buffer to empty.

3.1.1 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. As in the case of reactive components, a state of a process P is a valuation over the set S of its

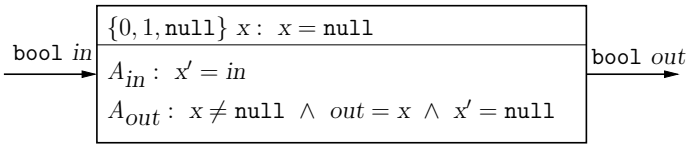

Figure 3.1: Asynchronous process `Buffer`

state variables, and the set of its states is Q_S. Initialization is expressed by a Boolean expression *Init* over the state variables S. The set of initial states contains the states satisfying this expression, that is, a state $q \in Q_S$ is an initial state of the process if q satisfies *Init*. As in the synchronous case, we require that there is at least one initial state, and there can be more than one initial state.

Recall that combinational synchronous components computed the outputs in response to inputs without maintaining any internal state. An asynchronous process, on the other hand, must maintain an internal state since processing of inputs is decoupled from production of outputs. Thus, the set S of state variables is always non-empty. As in the synchronous case, if all the input, output, and state variables range over finite types, we will call the process *finite-state*.

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 specifies an input channel x along with a value v of 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 then specifies an output channel y along with a value v of the type of y. We denote such an output by $y \,!\, v$. Such an output can be interpreted as *sending* the value v on the output channel y.

3.1.2 Input, Output, and Internal Actions

Processing of an input is called an *input transition*. During an input transition, the process can only update its state, and this step is decoupled from producing outputs. If the process in state s, on input $x \,?\, v$, can update its state to t, we write $s \xrightarrow{x \,?\, v} t$. An *input action* corresponding to an input channel x consists of all input transitions involving the channel x, and is specified by a Boolean expression, denoted A_x, over state variables S, input channel x, and primed state variables S' denoting the updated state. The set $\{A_x \mid x \in I\}$ of all input actions is denoted *InActs*.

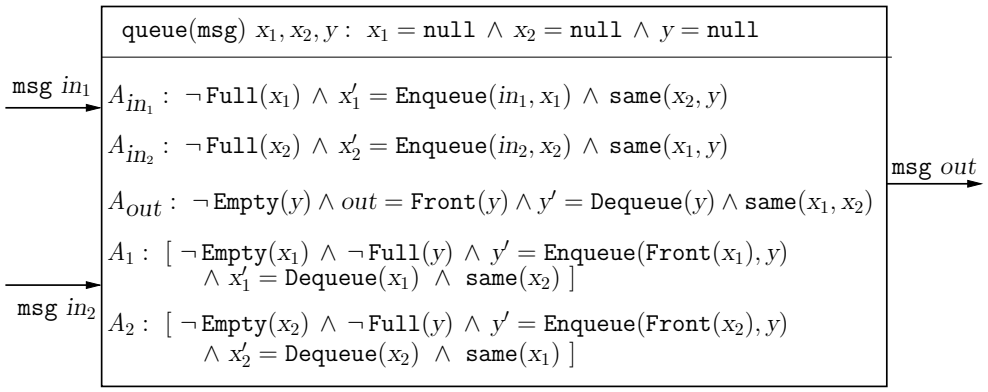

Figure 3.2: Asynchronous process `Merge`

For the process `Buffer` of Figure 3.1, the input action for the input channel *in* is A_{in} given by $x' = in$. This process has six input transitions: for $s \in \{0, 1, \texttt{null}\}$, $s \xrightarrow{in\,?\,0} 0$ and $s \xrightarrow{in\,?\,1} 1$. Note that if the process is supplied an input value when the buffer is non-empty, the old state is lost.

Producing an output is called an *output transition*. During such a transition, the process can update its state: if the process in state s can update its state to t while producing the output $y\,!\,v$, we write $s \xrightarrow{y\,!\,v} t$. The production of outputs corresponding to an output channel y is captured by a Boolean expression, denoted A_y, over state variables S, output channel y, and primed state variables S' denoting the updated state. This expression is called the *output action* of the process corresponding to the channel y, and the set $\{A_y \mid y \in O\}$ of all output actions is denoted *OutActs*.

For the process `Buffer` of Figure 3.1, the output action for the output channel *out* is A_{out} given by $x \neq \texttt{null} \ \wedge \ out = x \ \wedge \ x' = \texttt{null}$. The process has two output transitions: $0 \xrightarrow{out\,!\,0} \texttt{null}$ and $1 \xrightarrow{out\,!\,1} \texttt{null}$.

As a second example, consider the `Merge` process of Figure 3.2 with two input channels in_1 and in_2, both of type `msg`. The process uses a buffer dedicated to each of the input channels to store values received on that channel. We model a buffer using the type `queue`: `null` represents the empty queue, the operation $\texttt{Enqueue}(v, x)$ is used to add the value v at the end of the queue x, the operation $\texttt{Front}(x)$ returns the first element of the queue x, the operation $\texttt{Dequeue}(x)$ returns the queue x with its first element removed, 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 action A_{in_1} captures how the values received on the input channel in_1 are processed: if the queue x_1 is not full, the value of in_1 is enqueued

in x_1, leaving the other state variables x_2 and y unchanged. The expression $\texttt{same}(x_2, y)$ is an abbreviation for the expression $x_2' = x_2 \wedge y' = y$, stating that the new value is the same as the old value for both these variables. In general, for a list of variables $x_1, x_2, \ldots x_n$, $\texttt{same}(x_1, x_2, \ldots x_n)$ asserts that for each variable in the list, the primed version is equal to the unprimed version. If the queue x_1 is full, no input transition corresponding to the channel in_1 is possible. 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 action A_{in_2} corresponding to processing of the channel in_2 is similar.

The internal computation of the process, to be discussed in the next paragraph, transfers elements from the queues x_1 and x_2 to the queue y. The output action transmits elements of the queue y to the output channel *out*. Such an action is possible if the queue y is not empty. When possible, the output is set to the first element of the queue y, and it is removed from the queue using the `Dequeue` operation. The other two queues are left unchanged. This action is described by the expression A_{out}.

The internal computation of a process is described using *internal transitions*. Such transitions neither process inputs nor produce outputs, but update internal state. We write $s \xrightarrow{\varepsilon} t$ if the process in state s can update its state to t using an internal transition. The label ε indicates that there is no observable communication during this step. The process `Merge` has two kinds of internal transitions. Each kind is described by an expression, called an *internal action*, over the state variables S and the primed state variables S' denoting the updated state. The internal action A_1 denotes the computation step of dequeuing an element from the queue x_1 and enqueuing it into the queue y. This is possible when x_1 is not empty and y is not full. The queue x_2 is left unchanged. The action A_2 is symmetric, and corresponds to transferring the front element of the queue x_2 to the end of the queue y. Note that the two actions are independent of one another, and are executed at different speeds. This allows the process `Merge` to be composed of subprocesses that may be distributed or may be executed on different threads. The separation of the internal computation into two actions will also play a role in ensuring *fairness* discussed in Section 3.2.3. The set of all internal actions of a process is denoted *Acts*, equals $\{A_1, A_2\}$ for the process `Merge`.

We have now described all the elements of the definition of an asynchronous process. The definition is summarized below:

ASYNCHRONOUS PROCESS

An *asynchronous process* P has

- a finite set I of typed input channels, a finite set O of typed output channels, a finite set S of typed state variables, such that these three sets are pair-wise disjoint,
- a satisfiable Boolean expression *Init* over S,
- a set *InActs* of input actions, containing, for each input channel x, a Boolean expression A_x over $S \cup S' \cup \{x\}$,
- a set *OutActs* of output actions, containing, for each output channel y, a Boolean expression A_y over $S \cup S' \cup \{y\}$,
- a finite set *Acts* of internal actions, each of which is a Boolean expression over $S \cup S'$,

An *input* $x \,?\, v$ of P is an input channel $x \in I$ and a value v for x; an *output* $y \,!\, v$ of P is an output channel $y \in O$ and a value v for y; and a *state* $s \in Q_S$ of P is a valuation over S. A state $s \in Q_S$ is an *initial state* of P if s satisfies *Init*; for states s, t, input $x \,?\, v$, and output $y \,!\, v$, $s \xrightarrow{x \,?\, v} t$ is an *input transition* of P if $[s, t'][x \mapsto v]$ satisfies the input action A_x; $s \xrightarrow{y \,!\, v} t$ is an *output transition* of P if $[s, t'][y \mapsto v]$ satisfies the output action A_y; and $s \xrightarrow{\varepsilon} t$ is an *internal transition* of P if $[s, t']$ satisfies A for some internal action $A \in \mathit{Acts}$.

3.1.3 Executions

The operational semantics of a process can be captured by defining its executions. An execution starts in an initial state, and proceeds by executing either an input transition, or an output transition, or an internal transition, at every step. Only one action is executed at every step, and the order in which input, output, and internal actions are executed is totally unconstrained. Such a semantics for asynchronous interaction is called the *interleaving semantics*.

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 either an input, or output, or internal transition P.

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

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

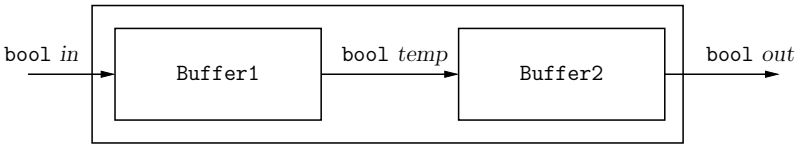

Figure 3.3: Block diagram for `DoubleBuffer` from two `Buffer` processes

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

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

$$(\texttt{null},\texttt{null},\texttt{null}) \xrightarrow{in_1\ ?\ 0} (0,\texttt{null},\texttt{null}) \xrightarrow{in_1\ ?\ 2} (02,\texttt{null},\texttt{null}) \xrightarrow{in_2\ ?\ 5}$$
$$(02,5,\texttt{null}) \xrightarrow{\varepsilon} (02,\texttt{null},5) \xrightarrow{in_2\ ?\ 3} (02,3,5) \xrightarrow{\varepsilon}$$
$$(2,3,50) \xrightarrow{out!5} (2,3,0) \xrightarrow{out!0} (2,3,\texttt{null}) \xrightarrow{\varepsilon} (\texttt{null},3,2)$$

Note that the sequence of values output by the process represents a merge of the sequences of input values supplied on the input channels. The relative order of values received on the input channel x_1 is preserved in the output sequence, and so is the relative order of values received on x_2, but an input value received on x_1 before a value received on x_2 may appear on the output later.

3.1.4 Operations on Processes

As discussed in Chapter 1, 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 3.3 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 1.11. As before, the meaning of such diagrams can be made precise using three operations, instantiation, parallel composition, and output hiding. The process `DoubleBuffer` is formally defined as

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

Input and output channel renaming

The operation of input or output channel renaming is used to ensure desired communication pattern. In Figure 3.3, the process `Buffer1` is obtained by renaming the output channel `out` of `Buffer` to `temp`, and is denoted $\texttt{Buffer}[\,out \mapsto$

$temp$]. Analogously, the process `Buffer2` is obtained by renaming the input channel `in` of `Buffer` to `temp`, and is denoted `Buffer`[$in \mapsto temp$]. When these two processes are composed, the shared name `temp` ensures that the output issued by the process `Buffer1` is processed by the process `Buffer2` as its input.

When composing processes, we assume that names of state variables are private, and 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 (Section 1.3.2). For example, for the process `Buffer1`, the set of input channels is $\{in\}$, the set of output channels is $\{temp\}$, the set of state variables is $\{x_1\}$, the initialization expression is $x_1 = \texttt{null}$, the input action A_{in} is given by $x_1' = in$, the output action A_{temp} is given by $x_1 \neq \texttt{null} \ \wedge \ temp = x_1 \ \wedge \ x_1' = \texttt{null}$, and there are no internal actions.

Parallel composition

The parallel composition operation combines two processes into a single process whose behavior captures the interaction between the two processes running concurrently in which output transition of one is synchronized with the input transition of another with the common channel name, and remaining actions are interleaved. To differentiate the asynchronous composition with the synchronous case, 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 output channels are disjoint. These requirements capture the assumption that only one process is responsible for controlling the value of any given variable. Input channels of one can be either input or output channels of the other.

The parallel composition of two compatible processes is defined below. The set of input, output, and state variables of the composite process are defined as in the synchronous case. The two processes initialize their states independently, and thus, a composite state is initial if initialization constraints of both processes are met. Thus the state of the composite 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. Such a state (s_1, s_2) is initial if both s_1 and s_2 are initial states.

When an input channel x is common to both processes, then both will process it simultaneously, and the corresponding input action is simply the logical conjunction of the two. That is, $(s_1, s_2) \xrightarrow{x\,?\,v} (t_1, t_2)$ is an input transition of the composite process precisely when $s_1 \xrightarrow{x\,?\,v} t_1$ is an input transition of P_1 and

$s_2 \xrightarrow{x\,?\,v} t_2$ is an input transition of P_2. If a channel x is an output channel of one, say P_1, and input channel of the other, then the two processes synchronize using this channel: when P_1 executes an output transition on x, the receiver P_2 executes the matching input transition. The action for x in the composite then is simply the conjunction of the output action for x of one process and the input action for x of the other process. The resulting transition is an output transition for the composite. That is, $(s_1, s_2) \xrightarrow{x\,!\,v} (t_1, t_2)$ is an output transition of the composite process precisely when $s_1 \xrightarrow{x\,!\,v} t_1$ is an output transition of P_1 and $s_2 \xrightarrow{x\,?\,v} t_2$ is an input transition of P_2. When one process, say P_1, processes an input corresponding to a channel x that is not a channel of the other process P_2, then the state of P_2 stays unchanged. For every input transition $s_1 \xrightarrow{x\,?\,v} t_1$ of the process P_1 and every state s of P_2, the composite has an input transition $(s_1, s) \xrightarrow{x\,?\,v} (t_1, s)$. The input action for the channel x in the composite then can be expressed as $A^1_x \wedge \texttt{same}(S_2)$, where A^1_x is the input action of P_1 for the channel x and S_2 is the set of state variables of P_2. Same holds for output transitions for a channel that involves only one process: the other keeps its state unchanged. Finally, an internal transition of the composite is an internal transition of exactly one of the two component processes, with the other keeping its state unchanged. This definition is formalized below.

PROCESS COMPOSITION

Let $P_1 = (I_1, O_1, S_1, \{A_x^1 \mid x \in I_1\}, \{A_y^1 \mid y \in O_1\}, Acts_1)$ and $P_2 = (I_2, O_2, S_2, \{A_x^2 \mid x \in I_2\}, \{A_y^2 \mid y \in O_2\}, Acts_2)$ be two compatible asynchronous processes. Then the *parallel composition* $P_1 \,|\, P_2$ is the asynchronous process P defined by:

- each state variable of a component process is a state variable of the composite: $S = S_1 \cup S_2$;
- each output channel of a component process is an output channel of the composite: $O = O_1 \cup O_2$;
- each input channel of a component process is an input channel of the composite, provided it is not an output channel of the other: $I = (I_1 \cup I_2) \setminus O$;
- the initialization expression for the composite is the conjunction of the component processes' initializations: $Init = Init_1 \wedge Init_2$;
- for each input/output channel $x \in I \cup O$ of the composite,
  1. if x is a channel common to both processes, that is, $x \in (I_1 \cup O_1) \cap (I_2 \cup O_2)$, then the action A_x of the composite is the conjunction $A_x^1 \wedge A_x^2$ of the corresponding actions of the two processes;
  2. if x is not a channel of P_1, that is, $x \notin (I_1 \cup O_1)$, then the action A_x of the composite is the conjunction $A_x^2 \wedge \texttt{same}(S_1)$; and
  3. if x is not a channel of P_2, that is, $x \notin (I_2 \cup O_2)$, then the action A_x of the composite is the conjunction $A_x^1 \wedge \texttt{same}(S_2)$;
- for each internal action of a component process, conjoined with an assertion requiring the state of the other process to stay unchanged, is an internal action of the composite:

$$Acts = \{A \wedge \texttt{same}(S_2) \mid A \in Acts_1\} \cup \{A \wedge \texttt{same}(S_1) \mid A \in Acts_2\}.$$

The composition of processes `Buffer1` and `Buffer2` gives the process with state variables $\{x_1, x_2\}$, output channels $\{temp, out\}$, input channels $\{in\}$, and initialization expression $x_1 = \texttt{null} \wedge x_2 = \texttt{null}$. For the composite process, the input action A_{in} is

$$x_1' = in \;\wedge\; x_2' = x_2;$$

the output action A_{temp} is

$$x_1 \neq \texttt{null} \;\wedge\; temp = x_1 \;\wedge\; x_1' = \texttt{null} \;\wedge\; x_2' = temp;$$

the output action A_{out} is

$$x_2 \neq \texttt{null} \;\wedge\; out = x_2 \;\wedge\; x_2' = \texttt{null} \;\wedge\; x_1' = x_1;$$

and it has no internal actions. Thus, only the process `Buffer1` participates in the processing of the channel *in*, the two processes synchronize on *temp*, and only `Buffer2` participates in producing the output on the channel *out*.

As in the synchronous case, the parallel composition operation is commutative and associative. Note that the problem of mutually cyclic awaits-dependencies discussed for the synchronous case does not arise in the asynchronous interaction. If x is an output channel of process P_1, and also an input channel of process P_2; and y is an output channel of P_1 and an input channel of P_2, 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 combinational loops.

Output hiding

If y is an output channel of a process P, the result of *hiding* y in P gives a process that behaves exactly like 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 the output action A_y corresponding to y into an internal action $\exists y.\, A_y$. Note that A_y is an expression that involves state variables, primed state variables, and the variable y. Existentially quantifying y gives an expression over only state variables and their primed versions, and is added to the set of internal actions of the process.

Let us revisit the process `Buffer1 | Buffer2`. If we hide the intermediate output *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 expression is $x_1 = \texttt{null} \wedge x_2 = \texttt{null}$. The input action A_{in} and the output action A_{out} are unchanged from `Buffer1 | Buffer2`. The process `DoubleBuffer` has one internal action given by

$$\exists\, temp\,.\, [\, x_1 \neq \texttt{null} \;\wedge\; temp = x_1 \;\wedge\; x_1' = \texttt{null} \;\wedge\; x_2' = temp\,],$$

which simplifies to

$$x_1 \neq \texttt{null} \;\wedge\; x_1' = \texttt{null} \;\wedge\; x_2' = x_1\,.$$

3.1.5 Safety Requirements

In Chapter 2, 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 expression *Init* of P is also the initialization expression for T; and

- if I and O are input and output variables of P and $Acts$ is the set of its actions, then the transition expression for T is given by

$$\bigvee_{x \in I} \exists x \,.\, A_x \;\vee\; \bigvee_{y \in O} \exists y \,.\, A_y \;\vee\; \bigvee_{A \in Acts} A.$$

The transition expression is simply the disjunction of all the input, output, and internal actions of P, where the input and output variables are existentially quantified. Thus, $s \rightarrow t$ is a transition of T precisely when the process has either an input or an output or an internal transition from state s to t. The disjunctive nature of the transition expression reflects the interleaving execution semantics of the asynchronous model.

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 (1) it holds initially, and (2) it is preserved by every transition. Since the transition expression is just a disjunction of expressions corresponding to different actions of the process, we need to show that it is preserved by every action (for instance, for every internal action A, $\varphi \wedge A \rightarrow \varphi'$ is valid).

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 transitions of P. Safety requirement is expressed as a property over the state variables of the monitor, and we want to establish that it is an invariant of the (asynchronous) parallel composition of the monitor and the system P.

Enumerative and symbolic reachability algorithms discussed in Sections 2.3 and 2.4 also apply to verification of asynchronous processes. We will discuss verification of liveness properties in Chapter 4.

3.2 Asynchronous Modeling Choices

3.2.1 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 processing of the corresponding input by another. Consider an output transition $s_1 \xrightarrow{x\,!\,v} t_1$ by a process P_1. Suppose x is an input channel for another process P_2, and suppose s_2 is the current state of P_2. If $s_2 \xrightarrow{x\,?\,v} t_2$ is an input transition of P_2, then in the composite system, there will be a synchronized output transition $(s_1, s_2) \xrightarrow{x\,!\,v} (t_1, t_2)$. However, if no such state t_2 exists, that is, the process P_2 is not willing to accept the input

x ? v in state s_2, then no synchronization is possible, and effectively, the process P_1 is blocked from executing its output transition. A process that is willing to accept every input in every state cannot cause the producer to wait, and is said to be *non-blocking*.

> NON-BLOCKING PROCESS
>
> For an asynchronous process P, its input x ? v is said to be *enabled* in a state s if there exists a state t such that $s \xrightarrow{x\ ?\ v} t$ is an input transition of P. The process P is said to be *non-blocking* if every input is enabled in every state.

The process `Buffer` of Figure 3.1 is non-blocking: its environment can always supply a value on the input channel *in*, even though some of these values are effectively lost. On the other hand, the process `Merge` of Figure 3.2 is non-blocking: an input on the channel $\texttt{in}_1$ cannot be processed if the queue x_1 is full, and thus, the producer of outputs on in_1 has to wait till this queue becomes non-full.

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 3.1.4 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 P_2 back to P_1 is needed for P_1 to be sure that its output was properly processed by P_2. In the implementation of blocking synchronization, the runtime system must somehow ensure that the receiver is willing to participate in the synchronizing action.

3.2.2 Atomicity of Actions

In the asynchronous model, actions of different processes, as well as different actions of the same process, are interleaved. A crucial design decision concerns how much computation can happen in a single action. For example, in our example of the process `Merge`, transferring of an element from one of the input queues x_1 and x_2 to the output queue y was separate from the output action, but the transfer action itself, say A_1, involved checking whether the queue x_1 is non-empty, checking whether the queue y has space, dequeuing an element from x_1, and enqueuing that element in y. All the computation within a single action happens atomically, without interference from other actions. Obviously, finer the atomicity, harder is the design problem, but closer is the design to a potential implementation.

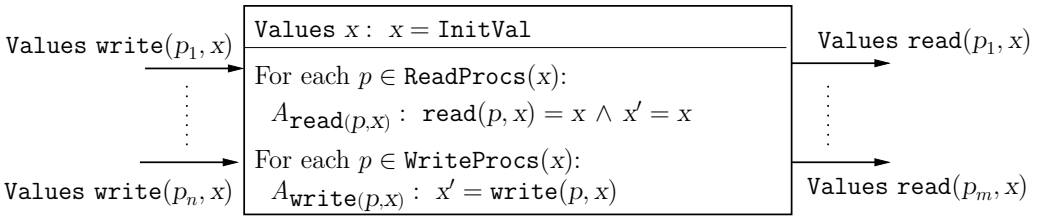

Figure 3.4: Atomic register supporting read and write operations

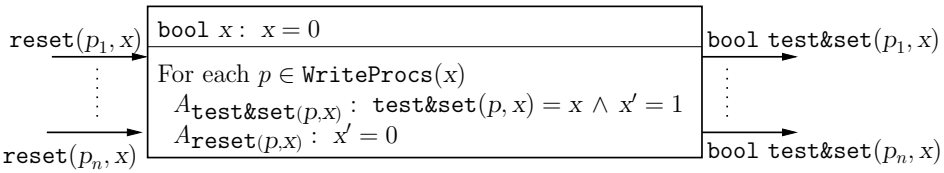

Figure 3.5: Boolean register supporting `test&set` and `reset` operations

The role of atomicity can be illustrated by which operations are supported by shared objects in shared memory systems. In a shared memory architecture, processes communicate by reading and writing shared variables.

Figure 3.4 shows the process `AtomicReg` that models a shared object x. The only atomic operations supported by this shared object x are read and write, and such an object is called an *atomic register*. The description is parameterized by (1) the set of values that the register can hold, denoted `Values`, (2) the initial value of the register, denoted `InitVal`, (3) the set of readers of x, denoted $\texttt{ReadProcs}(x)$, and (4) the set of writers of x, denoted $\texttt{WriteProcs}(x)$. For every shared atomic register x, and a process p that reads x, the synchronizing channel $\texttt{read}(p, x)$ denotes the action of reading x by p, which is an input action for the process p and an output action for the atomic register process. Similarly, $\texttt{write}(p, x)$ denotes the action of writing x by the process p, which is an output action for p and an input action for the atomic register.

The internal state x of the object holds the current value, and this initialized to `InitVal`. The output transition for the object corresponds to reading by one of the reader processes: the output action $\texttt{read}(p, x)\,!\,v$ is possible if the state of the object is v. When this transition is synchronized with an input transition $\texttt{read}(p, x)\ ?\ v$ by the reader process p, the value of x is communicated to p. This transition leaves the state of the object unchanged. Analogously, when a writer process p wants to produce its output $\texttt{write}(p, x)\,!\,v$, this transition is synchronized with the input transition $\texttt{write}(p, x)\ ?\ v$ by the shared object, and this updates the internal state x to v.

Figure 3.5 shows the process `Test&SetReg` that models a shared object x that

takes a Boolean value, but supports the primitive operations of `test&set` and `reset`. Initially the value of such a register is 0. The output `test&set`$(p, x)\,!\,v$, for some process p that can write to x, is possible if the state of the object is v. When this action is synchronized with an input transition `test&set`$(p, x)\ ?\ v$ by the process p, the value of x is communicated to p. Unlike the `read` action of the atomic register, the `test&set` action sets the internal state to 1. Thus, if multiple processes are attempting to synchronize with the `Test&SetReg` process with the internal state 0, the first synchronization will have the associated value 0, and this will set the state to 1 atomically, causing all subsequent transitions to have the associated value 1. When a process wants to change the value of the register back to 0, it executes the action `reset`(p, x), which is synchronized with the matching input transition by the register process, and this updates the internal state x to 0. Note that no value needs to be associated with the `reset` transition.

Two-process Consensus

To see how the choice of atomic primitives supported by shared objects impacts the ability 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 presence of unpredictable delays is central to many distributed computing problems.

More specifically, we have two asynchronous processes, say P_a and P_b, each of which has an initial Boolean value, denoted, v_a and v_b, unknown to the other. The processes want to arrive at Boolean decision values d_a and d_b, respectively, so that the following two requirements are met: (1) the decision values d_a and d_b of the two processes are identical, and (2) the decision value must be equal to one of the initial values v_a or v_b. The first requirement, called *agreement*, captures that the two 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 rules out input-oblivious solutions, such as "both decide on 0 no matter what the initial preferences are."

Suppose we want to design the processes P_a and P_b 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 an action shared by one of the processes and a single shared object. The third requirement for the consensus problem is called *wait-freedom*: at any point, if actions involving only one of the processes are repeatedly executed, it reaches a decision. This ensures that a process can

decide on its own without having to wait indefinitely for the other. Perhaps this requirement can be understood by considering the following protocol for solving consensus. We use two shared atomic registers x_a and x_b, each of which can take values in the set $\{\texttt{null}, 0, 1\}$, and is initially `null`. The process P_a first writes its preference v_a to x_a, then waits till x_b becomes non-null by reading it repeatedly. Symmetrically, the process P_b first writes its preference v_b to x_b, then waits till x_a becomes non-null by reading it repeatedly. When either process learns the preference of the other, it know both v_a and v_b, and can decide on, say, the logical-disjunction of the two (that is, decide on 0 if both preferences are 0, and 1 otherwise). This protocol satisfies the requirements of agreement and validity, but not wait-freedom. The reason is that if, say P_b, has not yet executed its write to v_b, then P_a will repeatedly read v_b, and won't be able to reach a decision.

It is possible to solve consensus using a test&set register. Consider the following protocol that uses two Boolean atomic registers x_a and x_b and a test&set register x. The initial values of x_a and x_b do not matter, and x is initially 0. The process P_a executes the following actions, with P_b following a symmetric protocol. The process P_a first sets x_a to its own preference v_a. Then it executes a `test&set` operation on x. If the value returned is 0 (implying the x was 0 before executing the atomic `test&set`), P_a goes ahead and decides on v_a. If the value returned is 1, then P_a concludes that the other process P_b has already executed `test&set` successfully, and hence, the register x_b must contain the preference v_b. The process P_a then proceeds to read x_b 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. The semantics of `test&set` ensures agreement. Each process executes a fixed number of actions, and thus, can decide without waiting for the other.

The key to the above solution was the primitive of `test&set` that updates the register and returns its old value in a single atomic step. If we are required to use only atomic registers, 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 the three requirements of agreement, validity, and wait-freedom.

Theorem 3.1 [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 the 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_a and P_b, 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 transition of one of the two processes, or a read transition involving one of the processes and one shared register, or a write transition 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 0 and 1 decisions are still possible: there is an execution starting in s in which both decide 0, and there is another execution starting in s in which both decide 1. A state is called 0-*committed* if in all executions starting in s the processes decide 0, and 1-*committed* if in all executions starting in s the processes decide 1.

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

As a first step towards the proof we first establish the following:

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

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

Now consider an initial state s in which the preferences v_a and v_b are different, say, 0 and 1, respectively. We claim that this state must be uncommitted. If not, suppose it is 0-committed. In state s, P_b has preference 1. Now consider another initial state t which is same as s except P_a also starts with the same preference 1. The states s and t are P_b-equivalent. By Lemma 1, we can conclude that the state t cannot be 1-committed. But that's a contradiction to the consistency requirement: in state t both preferences are 1, and thus every execution starting in t must lead to the decision 1.

We have now established that there is an uncommitted initial state. Since every execution starting at this state is finite, and the last state of every execution is committed (a unique decision has been reached when both processes stop), it follows that there must be a reachable uncommitted state whose all successors are committed. Let this state be s.

The state s is uncommitted, that is, both decisions are still possible, but executing one more step by any of the processes commits the protocol to the eventual

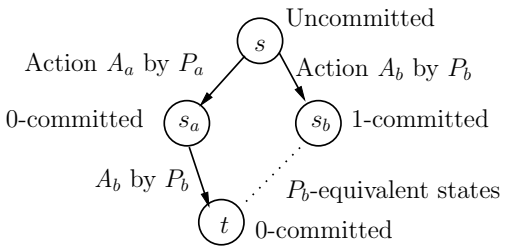

Figure 3.6: Impossibility result for consensus using atomic registers

decision. Without loss of generality, we can assume that there exist transitions $s \rightarrow s_a$ by process P_a using action A_a and $s \rightarrow s_b$ by process P_b using action A_b, such that every execution starting in s_a causes the decision to be 0, and every execution starting in s_b causes the decision to be 1 (see Figure 3.6). Each action can be an internal action, or a reading of a shared object, or writing of a shared object. We can consider all possible types of actions for A_a and A_b, and arrive at a violation of one of the requirements for the protocol.

Suppose the action A_a by P_a reads a shared atomic register. This action does not modify the state of any of the shared objects, and does not modify the internal state of P_b. Thus, states s and s_a are P_b-equivalent. Since the two states look the same to P_b, it can execute the action A_b in state s_a also, and let the resulting state be t (see Figure 3.6). States t and s_b are P_b-equivalent. But s_b is 1-committed, while t is 0-committed, a contradiction to Lemma 1.

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

Consider the case when both the processes write to the same atomic register, say, x. That is, in state s, the process P_a writes some value m_a to x leading to state s_a, and the process P_a writes some value m_b to the same register x leading to state s_b. Note that in state s_a, even though the value of the register x is different from its value in state s, the internal state of P_b is the same, and a write action is not influenced by the current value. Thus, in state s_a, the process P_b can write the same value m_b to the register x leading to the state t (see Figure 3.6). The writing of the value m_a by P_a to x has been lost, and did not influence what P_b was about to do in state s. In states s_b and t, the internal states of P_b are identical, and so are the states of all the shared registers. Thus, states s_b and t are P_b-equivalent, s_b is 1-committed, and t is 0-committed: a contradiction to Lemma 1. ■

3.2.3 Fairness

The execution of a process in the asynchronous model is obtained by interleaving different actions. At every step of the execution, if multiple actions can be executed, there is a choice. For example, for the `Buffer` process, at every step one can obtain the next state either by executing its input action or by executing its output action (provided the state is non-null). An execution where the output action is never executed is not particularly interesting. Similarly, for the `Merge` process, there are a total of five actions, all of which may be possible in a given state, and some action may get ignored forever. While the exact order in which the values arriving on the two input channels are merged is arbitrary by design, it is natural to require that all 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, but if a process is denied a chance to write successfully forever, no meaningful computation can occur.

The standard mathematical framework for capturing the informal requirement that "execution of an action should not be delayed 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. Consider the following ω-execution of the process `Buffer`:

$$\texttt{null} \xrightarrow{in?1} 1 \xrightarrow{in?1} 1 \xrightarrow{in?1} 1 \xrightarrow{in?1} 1 \xrightarrow{in?1} 1 \cdots$$

In this ω-execution, the state is 1 at every step, and the next state is always obtained by executing the input action. We will say that such an ω-execution is *unfair* to the output action: at every step there is a possible output transition, but it is never executed. An ω-execution of `Buffer` in which the input and output actions alternate for, say, first 1000 steps, but after that only the input action is executed indefinitely, is also considered unfair. For an ω-execution of `Buffer` to be fair with respect to its output action, it must contain infinitely many output transitions. Notice that for this particular process, every infinite execution must contain infinitely many input transitions: this is because every time `Buffer` executes an output transition, 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`. 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, transfers it to the channel y by executing the internal action A_2, and executes the output action sending this value on the output channel *out*. That is, the infinite sequence of actions it executes is A_{in_1} followed by the periodic execution of $A_{in_2}; A_2; A_{out}$. This

clearly starves the action $A_1$of transferring the element from the queue x_1 to y, which is possible at every step, but never executed. We 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 an action, whether corresponding to an output channel, or one of the internal actions, to be executed only when it is possible. This is captured by defining when an action is *enabled* in a state. The output action A_y of an asynchronous process P is enabled in a state s of P if there exists an output transition of P of the form $s \xrightarrow{y\,!\,v} t$. The internal action A of an asynchronous process P is enabled in a state s of P if there exists a state t such that $[s, t']$ satisfies A. An unfair execution is the one in which, after a certain point, an action 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, transfers it to the channel y by executing the internal action A_2, and executes the output action sending this value on the output channel *out*. We consider this to be a fair execution. The input action A_{in_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 action would amount to make implicit assumptions about the environment. Thus, fairness is required only for the actions that the process controls. If the `Merge` process is composed with another process P whose output channel is in_1, then the fairness with respect to the output action of P can force transitions involving the channel in_1. The ω-execution is also fair with respect to the internal action A_1. This is because the queue x_1 is always empty, and thus, the action A_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 requires that a choice that is continuously enabled is eventually executed. A stronger requirement is *strong fairness* which requires that a choice that is repeatedly enabled should be eventually executed. Suppose we have two internal actions such that executing one disables the other, but both get enabled by the arrival of a new input. In such a scenario, the ω-execution in which only first type of action is executed all the time is considered fair by the requirement of weak fairness (since from no point onwards, the second choice is always enabled), but would be ruled out by strong fairness.

As a motivating example, consider the unreliable FIFO buffer modeled by the process `UnrelFIFO` shown in Figure 3.7. The input action A_{in} simply transfers the input message to the internal queue x, and the output action A_{out} transmits the first message from the internal queue y on the output channel. The transfer of messages from the queue x to y is done by three internal actions. The action A_1 transfers a message from x to y correctly dequeuing a message from x and enqueuing it in y. The action A_2 models a loss of message and simply removes

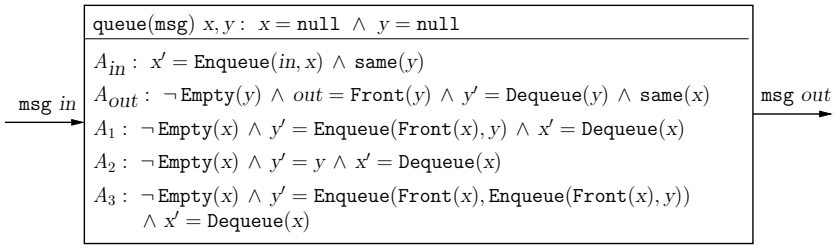

Figure 3.7: Asynchronous process `UnrelFIFO` for unreliable link

the message from x without adding it to y. The action A_3 models a duplication of messages: for every message it dequeues from x, it enqueues two copies of this message to y. The process thus models a communication link that may lose some messages, and may duplicate some messages. 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 action A_2. Since this action models loss of a message, it does not enqueue it in y. The actions A_{in} and A_1 are repeated forever in an alternating manner resulting in an execution where the queue y always stays empty. This ω-execution is weakly fair with respect to the action A_1 that models correct transfer of messages. This is because every time the input action enqueues the input message in x, the action A_1 is enabled, but every time the internal action A_2 removes this message, the action A_1 is disabled. Since it does not stay continuously enabled, weak fairness does not ensure its eventual execution. On the other hand, this infinite execution is not strongly-fair with respect to A_2: the action is repeatedly enabled, but 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 A_2.

Fairness Specification

The specification of the process `UnrelFIFO` also highlights that we need not demand fairness with respect to all actions. In particular, an infinite execution in which the duplication action A_3 or the lossy action A_2 is never executed is an acceptable and realistic execution. Losing or duplicating a message is not an active action to be executed, and does not need to be executed repeatedly. In particular, while the correct functioning of the system could rely on fairness with respect to A_1, it should not rely on fairness with respect to A_2: imagine a protocol that works correctly only when the underlying network loses messages. As argued above, we need strong fairness for the action A_1, but for the output

action, only weak fairness suffices. First of all, we do need some fairness assumption for the output action A_{out}, otherwise a message waiting to be delivered in the output queue y may never be transmitted on the output channel *out*. Then observe that once the queue y is non-empty, the action A_{out} stays enabled at least until it gets executed.

This suggests that the specification of an asynchronous process should annotate its output and internal actions: for some strong fairness is required, for some weak fairness is required, and some do not need any fairness. This is formalized in the definition below:

FAIRNESS SPECIFICATION

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, or output, or internal transition of P. An output action A_y is said to be *taken* at step j, if the label l_j involves the output channel y; and an internal action A is said to be *taken* at step j, if the label l_j is ε and $[s_{j-1}, s'_j]$ satisfies A. The ω-execution is *weakly-fair* with respect an output or an internal action, if either (1) for infinitely many indices j, the action is not enabled in the state s_j, or (2) for infinitely many indices j, the action is taken at step j. The ω-execution is *strongly-fair* with respect an output or an internal action, if either (1) there exists k such that for every $j \geq k$, the action is not enabled in the state s_j, or (2) for infinitely many indices j, the action is taken at step j. A *fairness specification* for an asynchronous process P a subset $\mathit{SFActs} \subseteq \mathit{OutActs} \cup \mathit{Acts}$ of actions for strong fairness and a subset $\mathit{WFActs} \subseteq \mathit{OutActs} \cup \mathit{Acts}$ of actions for weak fairness. Given such a specification, a fair ω-execution of P is an ω-execution that is strongly-fair with respect to every action in *SFActs* and is weakly-fair with respect to every action in *WFActs*.

In the formal definition above, weak fairness requirement is "repeatedly disabled or repeatedly taken," which is equivalent to "if continuously enabled then eventually taken." Similarly, strong fairness is "continuously disabled or repeatedly taken," which is equivalent to "if repeatedly enabled then repeatedly taken." Note that any execution that is strongly-fair with respect to an action is also weakly-fair with respect to that action, but converse may not hold.

When proving liveness requirements of an asynchronous process with fairness specification, we can restrict attention only to fair ω-executions. For example, for the `Merge` process, fairness specification requires weak-fairness for A_{out}, and strong-fairness for the internal actions A_1 and A_2. With such a fairness requirement, if it processes the input $in_1 ? v$ at any point in time, it is guaranteed that at some future time, it will produce the output $out ! v$. This is because in every fair execution if the i-th transition processes the input $in_1 ? v$, then v will be added to the queue x_1. The weak-fairness for the output action ensures that

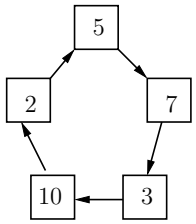

Figure 3.8: An asynchronous network with ring topology

if the outgoing queue has messages, they will eventually be transferred, and thus the output queue y won't stay full all the time. hence, the internal action A_1, with its strong-fairness requirement, cannot be ignored forever: once there is an element in the queue x_1, every time y is non-full, it is enabled, so it will be repeatedly enabled until it gets a chance to transfer an element from x_1. If the queue x_1 contains multiple elements in front of v, they will all be eventually transferred, finally along with v itself, to the queue y. Once the element v is enqueued in y, by a similar argument concerning fairness with respect to the action $Aout$, the output action will be executed repeatedly, thereby eventually transmitting v. 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.

3.3 Asynchronous Network 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 proceed in lock-step rounds as in the synchronous case. Furthermore, algorithms may have to be designed to account for potentially unreliable network links that may lose, duplicate, or reorder messages. We illustrate some of the design challenges using two classical algorithms, one for electing a leader in a ring of processes, and the alternating bit protocol for implementing reliable communication using unreliable links.

3.3.1 Leader Election

Let us revisit the coordination problem of leader election discussed in Section 1.4.2, now in the asynchronous setting. Now we assume that the underlying network connects the nodes in a unidirectional ring (see Figure 3.8 for an example ring). Each node has a unique identifier, and the algorithm should exchange messages so that eventually a single node declares itself to be the leader.

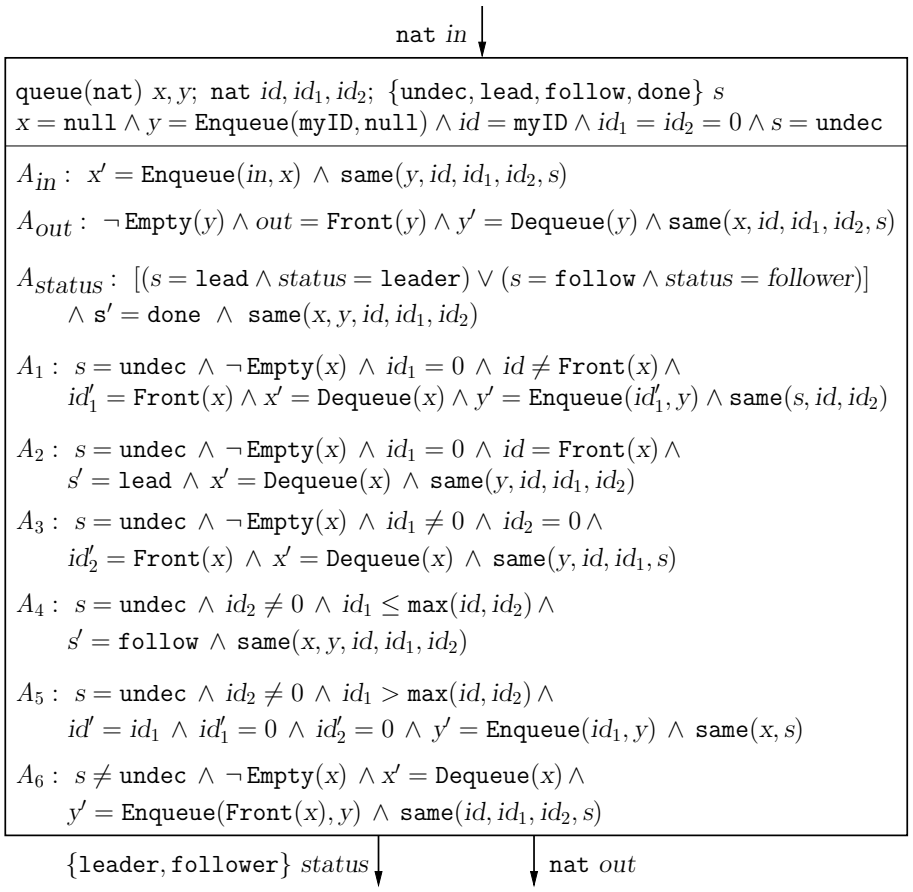

Figure 3.9: Asynchronous leader election in a ring

We model each network node as an asynchronous process P. The input channel *in* receives identifiers sent by the unique process, the predecessor of P, whose output is connected to this input, and the output channel *out* sends identifiers to the unique process, the successor of P, whose input is connected to this output. An internal queue x is used to store messages received, and the queue y is used to store message to be sent, which get delivered by the output action one by one. The decision is modeled by an output channel *status*: when the process concludes that it is either the leader or one of the followers, the decision is issued on this output. Our goal is to complete the specification of P so that when multiple instances of this process are composed to form a ring, the following requirements are met: (1) in a fair ω-execution, eventually every process outputs a value on its output channel *status*, and (2) exactly one process outputs the value `leader` on its output channel *status*.

As usual, the description of a process is parameterized by the identifier of the network node. We will assume that each identifier is a positive number. 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. A node can infer that the output it sent to its successor has propagated to everyone only when it receives a corresponding input from its predecessor. As a result, the protocol 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 1.4.2 that elects the process with the highest identifier. We describe a more interesting algorithm that reduces the number of messages that are sent. If the ring contains N processes then this algorithm will generate only $O(N \log N)$ messages, as opposed to $O(N^2)$ messages using the flooding algorithm. As it turns out, $O(N \log N)$ is also a lower bound on the number of messages that have to exchanged for this problem.

The algorithm is shown in Figure 3.9. The input action A_{in} simply transfers input messages to the internal queue x, and the output action A_{out} outputs pending messages from the queue y to the output channel *out*.

Each process has a mode, captured by the state variable s. Initially, s is undecided. Once a decision is reached, the process simply relays messages from its input queue x to its output queue y, and this is captured by the internal action A_6. The decision is reflected in the value of the mode variable s: when it is `lead` or `follow`, the output action A_{status} for the channel *status* outputs the decision, and changes the mode to `done`. This ensures that an output value is issued on the channel *status* just once. Note though, that the process will continue to relay messages in the mode `done` using the action A_6.

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

Initially, each process sends its identifier two steps along the ring. To achieve this, each process 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, *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 sent. When 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. When this value equals the current *id*, the process has won the election; this is modeled by the action A_2. Otherwise, the action A_2 dequeues the message, sets id_1 to this value, and sends

the message to its successor. If id_1 is non-zero, but id_2 is 0, the next input message to be processed is the message from the predecessor's predecessor. The action A_3 dequeues this message, and stores it in the variable id_2.

Once the process has received the values of id_1 and id_2, it compares these with id. If id_1 is the highest among these three identifiers, then the process continues to remain active, adopting the value of id_1 as its own identifier. This is modeled by the action A_5 which resets the variables, and initiates the new phase. If id_1 is not the highest among these three identifiers, then the process switches to follower mode by setting the mode variable s. This is captured by the action A_4. Subsequently, this process will only relay messages without examining them, and output `follower` on the channel `status`.

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 active if both $m_1 > m_0$ and $m_2 > m_0$. Consider the predecessor P' of P. Then, for the process P', its own identifier, that is, the value of its id variable will be 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 active adopting m_1 as its identifier, P' will become inactive. It is easy to see that the number of processes that continue to stay active is at least 1 (the successor of the process with the highest identifier is guaranteed to be active) and at most half of the current number of active processes (if a process stays active its predecessor is guaranteed to become inactive).

For the example network shown in Figure 3.8, for the process 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 active process, with 7 as its identifier. For the process 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 inactive. After the first phase, only processes 3 and 2 will be active, with modified identifiers 7 and 10, respectively.

When a process decides to stay active, 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 2 input messages, it examines the relative ordering of its identifier and the identifiers of its two predecessors, making decisions as before. That is, in every subsequent phase, the current ring with the reduced number of active processes repeats the same protocol, thereby again reducing the number of active processes by at least half. The presence of inactive processes does not influence the logical argument since they are simply relaying messages.

When an active process receives an input message that is equal to its current identifier, then it can conclude that it is the only active process, and proceed to declaring itself as the leader (action A_2). 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.

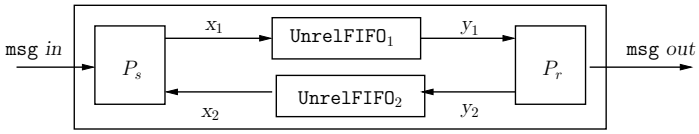

Figure 3.10: The block diagram for reliable communication

Continuing our example from Figure 3.8, during the second phase, for process 3, 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 active 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.

The precise correctness argument is complicated by the fact that the phases are not synchronized, and it any given time, neighboring processes may be executing different phases. Fairness assumption ensures that all messages will eventually be processed causing each process to finish its current phase, and proceed to the next phase. In each phase, each process sends at most 2 messages. If the ring contains N processes, then each phase contributes at most $2N$ messages, and the number of phases is at most $\texttt{log}\, N$, leading to an overall bound of $2N\, \texttt{log}\, N$ 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 specification.

3.3.2 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 3.10 acts as a reliable FIFO buffer with respect to its input and output channels using two instances of the unreliable communication link `UnrelFIFO` (see Figure 3.7). 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 $\texttt{UnrelFIFO}_1$ transfers messages from P_s to P_r, and the unreliable link $\texttt{UnrelFIFO}_2$ transfers messages from P_r to P_s.

To deliver a message that P_s receives on its input channel *in*, it may need to send it repeatedly to P_r since the link $\texttt{UnrelFIFO}_1$ may lose messages, and P_r needs to send an explicit acknowledgment back to P_s notifying successful delivery. The acknowledgment also needs to be sent repeatedly to ensure eventual successful delivery in presence of lost messages. A key design challenge is to match messages with acknowledgments, and identification of duplicates. 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.

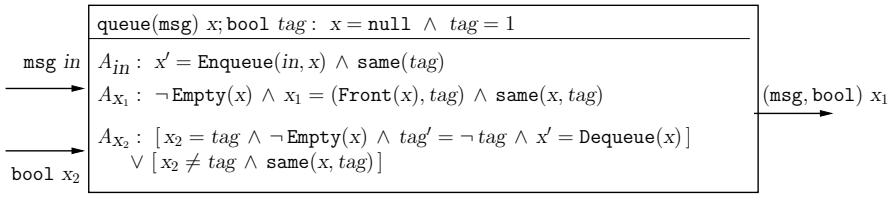

Figure 3.11: The sender process for the alternating-bit protocol

The sender interface P_s is shown in Figure 3.11. It maintains a queue x of messages that it receives on its input channel *in*, and this is captured by the action A_{in}. The *tag* is a Boolean variable that is initially 1. When P_s sends a message to the receiver process P_r using the unreliable FIFO medium on the channel x_1, it augments the message with the current value of *tag*, and does not remove the message from the queue. This action A_{X_1} may get executed repeatedly. When it gets an acknowledgment, in the form of a tag bit from the receiver, it checks if the received tag matches its own tag, and only if this check succeeds, it removes the message from its queue, and toggles the tag. The processing of acknowledgment tags is modeled by the action A_{X_2}. The toggling of the tag will cause the next message in the queue x to be sent, possibly repeatedly, on the channel x_1 with this updated tag. The fairness specification demands weak-fairness for the output action: once a message is enqueued in x, the action A_{X_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 3.12. The messages it receives are stored in the queue y, and it also maintains a tag, which is initially 0 (complement of the initial tag value of the sender). When it receives a message on the channel y_1, it checks if the tag of the incoming message matches its own tag. If so, the incoming message is a new message, and it is added to the queue y. This is captured by the action A_{Y_1}, where the primitives `First` and `Second` are used to retrieve the two fields of the incoming message. Messages in the queue y are transmitted on the output channel using the action A_{out}. The receiver repeatedly sends the current value of its tag to the sender as an acknowledgment on the channel y_2. To ensure eventual delivery on both the output channels, the fairness specification demands weak-fairness for both the output actions A_{Y_1} and A_{out}. Since these actions are not disabled by competing actions, we don't need strong fairness.

This is how the protocol executes. Suppose the process P_s receives a message, say m_1, on its input channel *in*. Then, it will send repeatedly the message $(m_1, 1)$ to P_r using the unreliable channel. Each such message may be lost or duplicated. Meanwhile, P_r is repeatedly sending the tag bit 0 to P_s, but P_s will ignore all such messages. The first time the message $(m_1, 0)$ is successfully received by P_r, it will change its tag to 1, and enqueue m_1 to its output queue,

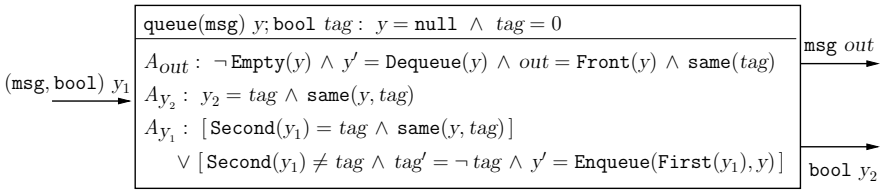

Figure 3.12: The receiver process for the alternating-bit protocol

and it will eventually be delivered on the output channel *out*. Additional copies of $(m_1, 0)$ will be ignored by 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. It will repeatedly send the tag 1 to P_s as an acknowledgment, each such message again may be lost or duplicated, but eventually, P_s will receive the tag 1. At this point, P_s will remove the message m_1 from its input queue. If it had received additional messages on the channel *in* during this period, they would have been enqueued in x, and if m_2 is next pending message, then P_s will start sending $(m_2, 1)$ to P_r. If P_s receives additional tag messages 1, it will ignore them. The message m_2 will dequeued only when P_s receives the tag 0.

3.4 Exercises

1. We want to design an asynchronous adder process `AsyncAdd` with input channels x_1 and x_2, and output channel y, all of type `nat`. If the i-th input message arriving on the channel x_1 is v and the i-th input message arriving on the channel x_2 is w, then the i-th value output by the process `AsyncAdd` on its output channel should be $v + w$. Describe all the components of the process `AsyncAdd`.

2. We want to design an asynchronous process `Spit` that is the dual of `Merge`. The process `Split` has one input channel `in` and two output channels $\texttt{out}_1$ and $\texttt{out}_2$. 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`. Suppose we want to capture the requirement that the distribution of messages among the two output channels should be, while unspecified, fair in the sense that if infinitely many messages arrive then both output channels should have infinitely many messages output. How would you add fairness specification to your design to capture this requirement? If you are using strong fairness, 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).

3. By modifying the specification of the process `UnrelFIFO` of Figure 3.7, construct a precise specification of the process `VeryunrelFIFO` which, in

addition to losing and duplicating messages, can also reorder messages. What would be a natural fairness specification for the modified process?

4. Consider the generalization of two process consensus 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), consistency (the decision value must be a preference of one the processes), and wait-freedom (if a process takes steps all by itself, it should reach a decision in finitely many steps without having to wait for others). Describe why the strategy described in the 2-process protocol using a `test&set` register does not generalize to 3 processes? Prove that there is no solution to the consensus problem, when the number of processes is 3 (or more), using only atomic registers and test&set registers as shared objects.

5. Consider the shared object `StickyBit` that supports read and write operations as in case of an atomic register, with some modification. The internal state of a `StickyBit` can be `null`, 0, or 1. The read operation outputs the current value. The write operation has a Boolean (0 or 1) 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, state is already 0 or 1), the value stays unchanged. Describe a protocol for solving 2-process consensus using a single `StickyBit` (you may use any number of additional atomic registers as you need). Can you solve consensus for 3 (or more generally, n) processes using multiple `StickyBit` and `AtomicReg` objects?

6. For the leader election protocol of Section 3.3.1, consider a ring with 16 nodes where the identifiers of the processes in the order in which they are connected 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?

7. For the leader election protocol of Section 3.3.1, describe the best-case and worst-case scenarios: (a) describe the scenario in which only one node will stay active after the first round, and (b) describe the scenario in which the protocol will need `log` N phases before the election.

8. Consider the alternating-bit protocol from Section 3.3.2. Suppose we know that the unreliable FIFO communication links may lose messages, but does not duplicate it. How would you modify the protocol to take advantage of this?

9. Consider the specification of the process `VeryunrelFIFO` designed in Exercise 3 above, of an unreliable link that may lose messages, duplicate messages, and reorder messages. First, show that the alternating-bit protocol from Section 3.3.2 does not work correctly if we replace the instance of `UnrelFIFO` with instances of `VeryunrelFIFO`. How would you modify the processes P_s and P_r so that reliable communication is guaranteed even

in presence of this added complication of reordering? Argue that the modified protocol works correctly. Hint: a Boolean-values tag is not enough, and messages need to be tagged with a counter variable of type `nat`.