Contents

© Rajeev Alur and Thomas A. Henzinger

Chapter 1

Reactive Modules

This chapter introduces a modeling language, Rml (Reactive Module Language), for describing the architecture and behavior of hardware and software systems. Modeling languages are to be used by hardware designers, software engineers, and CAD tools during the early stages of the design process for describing and, more importantly, analyzing blueprints of a system. Thus, unlike an implementation language such as a hardware description language or a programming language, a modeling language need not provide extensive mechanisms for structuring control flow and manipulating data. Rather, a modeling language must have the following four essential characteristics. First, it must facilitate high-level, partial system descriptions by supporting nondeterminism. Second, it must facilitate the description of interactions between systems and system components by supporting concurrency. Third, it must facilitate the rapid prototyping and simulation of system descriptions by supporting an execution model. Fourth, it must facilitate the formal analysis of system behaviors by supporting a precise mathematical semantics. The first and fourth characteristics distinguish Rml from many common implementation languages; the second and third characteristics distinguish Rml from many common requirements specification languages.

1.1 Definition of Reactive Modules

We model systems as reactive modules. Reactive modules resemble molecules, in that they interact with each other to form composite objects. Reactive modules are built from atoms, and atoms are built from variables —the elementary particles of systems. Our presentation proceeds bottom-up, from variables to atoms, modules, and the composition of modules.

1.1.1 Variables

We consider systems that are discrete, deadlock-free, and nondeterministic. A discrete system is a collection of variables that, over time, change their values in a sequence of rounds. The first round is an initialization round, and all subsequent rounds are update rounds. In the initialization round, the values of all variables are initialized, and in every update round, the values of all variables are updated. Deadlock-freedom means that in the initialization round, there is at least one option for initializing each variable, and in every update round, there is at least one option for updating each variable. Consequently, every update round can be followed by another update round. Nondeterminism means that in the initialization round, there may be several options for initializing a variable, and in an update round, there may be several options for updating a variable. Consequently, two exact copies of a system can, over time, exhibit very different behaviors.

Initial commands and update commands

We define the behavior of a variable using two guarded commands —an initial command and an update command. While unprimed symbols, such as x, refer to the value of a variable at the beginning of a round, primed symbols, such as x', refer to the value of the same variable at the end of a round. A value of x at the end of the initialization round is called an initial value of x. The possible initial values for a variable are defined by an initial command. For example, the initial command

init
$[\!] \; \mathit{true} \rightarrow x' := 0$
$[\!] \; \mathit{true} \rightarrow x' := 1$

asserts that 0 and 1 are the possible initial values of x. A value of x at the beginning of an update round is called a current value of x, and a value of x at the end of an update round is called a next value of x. In every update round, the possible next values for x may depend on the current value of x, and on the current values of other variables. The possible next values for a variable are defined by an update command. For example, the update command

$$
\begin{array}{l}
\textbf{update} \\
\quad [\!]\ y = 0 \rightarrow x' := x + 1 \\
\quad [\!]\ y \neq 0 \rightarrow x' := x - 1 \\
\quad [\!]\ \mathit{true} \quad \rightarrow x' := x
\end{array}
$$

asserts that in every update round, if the current value of y is 0, then the value of x is either incremented or stays unchanged, and if the current value of y is different from 0, then the value of x is either decremented or stays unchanged.

Example 1.1 [Scheduler] Consider a scheduler that, in every round, assigns a processor to one of two tasks. The nonnegative-integer variable task_1 indicates the amount of processor time, measured in rounds, which is necessary to complete the first task. Similarly, the nonnegative-integer variable task_2 indicates the amount of processor time which is necessary to complete the second task. The ternary variable proc has the value 0 if in the most recent round, the processor has been idle; proc has the value 1 if the processor has been assigned to the first task; and proc has the value 2 if the processor has been assigned to the second task. The initial command

$$
\begin{array}{l}
\textbf{init} \\
\quad [\!]\ \mathit{true} \rightarrow \mathit{proc}' := 0
\end{array}
$$

asserts that initially the processor is idle. The update command

$$
\begin{array}{l}
\textbf{update} \\
\quad [\!]\ \mathit{task}_1 = 0 \wedge \mathit{task}_2 = 0 \rightarrow \mathit{proc}' := 0 \\
\quad [\!]\ \mathit{task}_1 > 0 \qquad\qquad\qquad \rightarrow \mathit{proc}' := 1 \\
\quad [\!]\ \mathit{task}_1 = 0 \wedge \mathit{task}_2 > 0 \rightarrow \mathit{proc}' := 2
\end{array}
$$

asserts that the scheduler always gives priority to the first task. ■

Simultaneous updates

Within a round, some variables are initialized or updated simultaneously, and some variables are initialized or updated sequentially. We insist that if two variables x and y are initialized or updated simultaneously in some round, then x and y are initialized and updated simultaneously in every round. A set of variables that are initialized simultaneously in the initialization round, and updated simultaneously in every update round, are said to form an atom. It is often convenient to name atoms, in which case the constituent variables of an atom are referred to as the variables that are controlled by the atom. The behaviors of all variables that are controlled by one atom are defined using a single initial command and a single update command. For example, for the atom controlling the two variables x and y, the initial command

$$
\begin{array}{l}
\textbf{init} \\
\quad [\!]\ \mathit{true} \rightarrow x' := 0;\ y' := 1 \\
\quad [\!]\ \mathit{true} \rightarrow x' := 1;\ y' := 0
\end{array}
$$

asserts that the possible initial values of x and y are 0 and 1, and that the initial values of x and y are different. For the same atom, the update command

$$
\begin{array}{l}
\textbf{update} \\
\quad [\!]\ x \leq y \rightarrow x' := x + 1;\ y' := y - 1 \\
\quad [\!]\ x \geq y \rightarrow x' := x - 1;\ y' := y + 1
\end{array}
$$

asserts that in every update round, depending on the current values of x and y, either x is incremented and y is decremented, or vice versa.

Example 1.2 [Scheduler] Consider a scheduler similar to Example 1.1, except that the scheduler alternates priorities between both tasks. If in a given round the processor is assigned to the first task, then in the next round the second task is given priority over the first task, and vice versa. The binary variable *prior* indicates which of the two tasks will be given priority in the upcoming round. The variables *proc* and *prior* form an atom; that is, the processor assignment and the priority information are updated simultaneously. Assuming that initially either task may be given priority, we have the initial command

$$
\begin{array}{l}
\textbf{init} \\
\quad [\!]\ \mathit{true} \rightarrow \mathit{proc}' := 0;\ \mathit{prior}' := 1 \\
\quad [\!]\ \mathit{true} \rightarrow \mathit{proc}' := 0;\ \mathit{prior}' := 2.
\end{array}
$$

Assuming that a task retains priority until it is given the processor, we have the update command

$$
\begin{array}{lll}
\textbf{update} & & \\
\quad [\!]\ \mathit{task}_1 = 0 \wedge \mathit{task}_2 = 0 & & \rightarrow \mathit{proc}' := 0 \\
\quad [\!]\ \mathit{prior} = 1 \wedge \mathit{task}_1 > 0 & & \rightarrow \mathit{proc}' := 1;\ \mathit{prior}' := 2 \\
\quad [\!]\ \mathit{prior} = 1 \wedge \mathit{task}_1 = 0 \wedge \mathit{task}_2 > 0 & & \rightarrow \mathit{proc}' := 2 \\
\quad [\!]\ \mathit{prior} = 2 \wedge \mathit{task}_2 > 0 & & \rightarrow \mathit{proc}' := 2;\ \mathit{prior}' := 1 \\
\quad [\!]\ \mathit{prior} = 2 \wedge \mathit{task}_2 = 0 \wedge \mathit{task}_1 > 0 & & \rightarrow \mathit{proc}' := 1.
\end{array}
$$

Note that in the first, third, and fifth guarded assignments, the value of the variable *prior* stays unchanged; that is, $\mathit{prior}' := \mathit{prior}$. ■

Sequential updates

We insist that if a variable x is initialized or updated before a variable z in some round, then x is initialized and updated before z in every round. If x is initialized before z in the initialization round, and x is updated before z in every update round, then the variable x is said to be awaited by the variable z. If z awaits x, then the possible initial values for z may depend on the initial value of x, and in every update round, the possible next values for z may depend on the next value of x. For example, assuming that z awaits both x and y, the initial command

$$
\begin{array}{l}
\textbf{init} \\
\quad [\!]\ \mathit{true} \rightarrow z' := x' + y'
\end{array}
$$

asserts that the initial value of z is equal to the sum of the initial values of x and y, and the update command

$$\begin{array}{l}\textbf{update}\\ \quad [\!]\ y' = 0 \to z' := x\\ \quad [\!]\ y' \neq 0 \to z' := x'\end{array}$$

asserts that in every update round, if the next value of y is 0, then the next value of z is equal to the current value of x, and otherwise the next value of z is equal to the next value of x.

Example 1.3 [Scheduler] In every update round of the scheduler example, the indicators $task_1$ and $task_2$ for pending work are updated after the processor is assigned to one of the two tasks; that is, both indicators await the processor assignment $proc$. The indicator $task_1$ is decremented in every round in which the processor is assigned to the first task, and the indicator $task_2$ is decremented in every round in which the processor is assigned to the second task. In addition, new work for a task may arrive in any round, and it arrives in blocks of 5 units. The nonnegative-integer variable new_1 indicates the amount of new work that has arrived in the most recent round for the first task, and the nonnegative-integer variable new_2 indicates the amount of new work that has arrived for the second task. The initial command and the update command for new_1 are identical, and we write

$$\begin{array}{l}\textbf{initupdate}\\ \quad [\!]\ \mathit{true} \to \mathit{new}'_1 := 0\\ \quad [\!]\ \mathit{true} \to \mathit{new}'_1 := 5\end{array}$$

to avoid duplication. The variable $task_1$ awaits both new_1 and $proc_1$, and its behavior is defined by the commands

$$\begin{array}{l}\textbf{init}\\ \quad [\!]\ \mathit{true} \to \mathit{task}'_1 := \mathit{new}'_1\\ \textbf{update}\\ \quad [\!]\ \mathit{proc}' = 1 \to \mathit{task}'_1 := \mathit{task}_1 + \mathit{new}'_1 - 1\\ \quad [\!]\ \mathit{proc}' \neq 1 \to \mathit{task}'_1 := \mathit{task}_1 + \mathit{new}'_1.\end{array}$$

The behaviors of the variables new_2 and $task_2$ are defined by similar commands. ■

1.1.2 Atoms

The initialization round and every update round consist of several subrounds, one for each atom. For an atom U, in the U-subround of the initialization round, the controlled variables of U are initialized simultaneously, as defined by the initial command of U. In the U-subround of an update round, the controlled variables of U are updated simultaneously, as defined by the update command of U. If the possible next values for some controlled variable of U depend on the current value of a variable x, then x is said to be read by the atom U. Read variables occur in the update command of U as unprimed symbols. If the possible initial values for some controlled variable of U depend on the initial value of x, or if the possible next values for some controlled variable of U depend on the next value of x, then the variable x is awaited by the atom U. Awaited variables occur in the initial and update commands of U as primed symbols. A variable can be both read and awaited by an atom, or read and controlled, but for obvious reasons, a variable cannot be awaited and controlled.

Atom

Let X be a finite set of typed variables. An X-atom U consists of an atom declaration and an atom body. The declaration of U consists of a nonempty set $\mathsf{ctr}X_U \subseteq X$ of controlled variables, a set $\mathsf{read}X_U \subseteq X$ of read variables, and a set $\mathsf{await}X_U \subseteq X \backslash \mathsf{ctr}X_U$ of awaited variables. The body of U consists of an initial command init_U and an update command update_U. The initial command init_U is a guarded command from $\mathsf{await}X'_U$ to $\mathsf{ctr}X'_U$. The update command update_U is a guarded command from $\mathsf{read}X_U \cup \mathsf{await}X'_U$ to $\mathsf{ctr}X'_U$.

Remark 1.1 [Atom variables] All variables of an X-atom are taken from the underlying set X of variables. If X and Y are two sets of variables with $X \subseteq Y$, then every X-atom is also a Y-atom. ■

An important special case of atoms are the deterministic atoms. For a deterministic atom, the initial values of all controlled variables are uniquely determined by the initial values of the awaited variables, and in every update round, the next values of all controlled variables are uniquely determined by the current values of the read variables and the next values of the awaited variables.

Deterministic atom

The atom U is deterministic if both the initial command init_U and the update command update_U are deterministic. Otherwise, U is a nondeterministic atom.

The consistency of atoms

We insist on two consistency requirements for atoms, which ensure that a collection of variables that are controlled by several atoms can be initialized and updated unambiguously. First, we require that no variable be controlled by more than one atom. This prevents the assignment of multiple, inconsistent values to a variable. Second, we require that there be no circular await dependencies between variables. The await dependencies between the controlled variables and the awaited variables of an atom constrain the possible temporal orderings of the subrounds within a round. We allow only await dependencies that permit, within every round, at least one possible ordering of the subrounds. Consider two variables, x and y, which are controlled, respectively, by the two atoms U_x and U_y. If x is an awaited variable of U_y, then the initial value of y may depend on the initial value of x, and the next value of y may depend on the next value of x. Therefore x must be initialized and updated before y; that is, the U_x-subround must go before the U_y-subround in the initialization round and in every update round. It follows that y must not be an awaited variable of U_x. More generally, the atom U_1 precedes the atom U_n if there is a chain $U_2, \ldots, U_{n-1}$ of atoms such that for all $2 \leq i \leq n$, some controlled variable of U_{i-1} is an awaited variable of U_i. If U precedes V, then the U-subround must go before the V-subround in every round. Therefore it must not happen that U precedes V and V precedes U.

Atom consistency

Let X be a finite set of typed variables, let $\mathcal{U}$ be a set of X-atoms, and let x and y be two variables in X. The variable y awaits the variable x, written $x \prec_{\mathcal{U}} y$, if some atom in $\mathcal{U}$ controls y and awaits x. The set $\mathcal{U}$ is consistent if (1) no variable is controlled by more than one atom in $\mathcal{U}$, and (2) the transitive closure $\prec_{\mathcal{U}}^{+}$ of the await relation on the variables in X is asymmetric. Given two atoms U and V in $\mathcal{U}$, the atom U precedes the atom V, written $U \lll_{\mathcal{U}} V$, if there is a variable x controlled by U and a variable y controlled by V such that $x \prec_{\mathcal{U}}^{+} y$.

Proposition 1.1 [Partial order of atoms] If $\mathcal{U}$ is a consistent set of atoms, then the precedence relation $\lll_{\mathcal{U}}$ is a strict partial order on $\mathcal{U}$.

Exercise 1.1 {`T2`} [Proof of Proposition 1.1] (a) Prove Proposition 1.1. (b) Show that the definition of consistency cannot be relaxed; that is, prove that if the precedence relation $\lll_{\mathcal{U}}$ of a set $\mathcal{U}$ of atoms is asymmetric, then condition (2) for the consistency of $\mathcal{U}$ is satisfied. ■

The execution of atoms

For a consistent set $\mathcal{U}$ of atoms, the linearizations of the partial order $\lll_{\mathcal{U}}$ determine the possible sequences of subrounds within a round. These linearizations are called the execution orders for $\mathcal{U}$. Every consistent set of atoms has at least one execution order, and possibly several.

Execution order

Let X be a finite set of typed variables, and let $\mathcal{U}$ be a consistent set of X-atoms. An execution order for $\mathcal{U}$ is a sequence $U_1, \ldots, U_n$ of the atoms in $\mathcal{U}$ which does not violate the precedence relation $\lll_{\mathcal{U}}$; that is, for all $1 \leq i, j \leq n$, if $U_i \lll_{\mathcal{U}} U_j$, then $i < j$.

A system is closed if it controls the behavior of all its variables. We model a closed system with the set X of variables as a consistent set $\mathcal{U}$ of X-atoms so that each variable in X is controlled by some atom in $\mathcal{U}$. Such a set $\mathcal{U}$ of atoms is executed by carrying out, in the initialization round, all initial commands of $\mathcal{U}$ in some execution order, and by carrying out, in every update round, all update commands of $\mathcal{U}$ in some execution order. The outcome of the execution is called an initialized trajectory of $\mathcal{U}$: it gives, for each variable in X, a sequence of values, one for every round. If $\mathcal{U}$ contains some nondeterministic atoms, then for any given number of rounds, there can be many initialized trajectories. By contrast, the fact that $\mathcal{U}$ may have several execution orders does by itself not give rise to multiple trajectories. In particular, if all atoms in $\mathcal{U}$ are deterministic, then for any given number of rounds, there is a unique initialized trajectory.

Exercise 1.2 {T2} [Execution of atoms] Let X be a finite set of typed variables, and let $\mathcal{U}$ be a consistent set of X-atoms so that each variable in X is controlled by some atom in $\mathcal{U}$. Show that the possible outcomes of executing $\mathcal{U}$ do not depend on the execution orders that are chosen during the execution of $\mathcal{U}$; that is, prove that every initialized trajectory of $\mathcal{U}$ can be obtained by choosing an arbitrary execution order for $\mathcal{U}$ and maintaining the chosen execution order in every round. ■

Example 1.4 [Scheduler] The scheduler from Example 1.3 is a closed system with six variables, new_1, new_2, $task_1$, $task_2$, $proc$, and $prior$, which are arranged in the five atoms $A1$–$A5$ shown in Figure 1.1. In Rml, each atom is written as an atom name followed by an atom declaration and an atom body. In atom declarations, the keywords **reads** or **awaits** are omitted if the sets of read or awaited variables are empty. The atoms $A3$ and $A4$ are deterministic; the atoms $A1$, $A2$, and $A5$ are nondeterministic. The set $\{A1, \ldots, A5\}$ of atoms is consistent, because $new_1 \prec task_1$, $proc \prec task_1$, $new_2 \prec task_2$, and $proc \prec task_2$ are the only await dependencies. There are many execution orders, including $A1, A2, A5, A3, A4$ and $A5, A2, A4, A1, A3$. Figure 1.2 shows two, arbitrarily chosen, initialized trajectories with 15 update rounds each. The first of these two trajectories is depicted graphically, in the form of a so-called timing diagram, in Figure 1.3. The vertical dotted lines of the timing diagram represent boundaries between rounds. Note, for instance, that the variable $task_1$ changes its initial value, in the first update round, only after both new_1 and $proc$ have changed their initial values. ■

Combinational and sequential atoms

In the initialization round, the possible initial values for the controlled variables of an atom depend, in some way, on the initial values of the awaited variables. If in every update round, the possible next values for the controlled variables depend in the same way on the next values of the awaited variables, then the

atom $A1$ **controls** new_1
  initupdate
    $[\!]\ true \rightarrow new_1' := 0$
    $[\!]\ true \rightarrow new_1' := 5$

atom $A2$ **controls** new_2
  initupdate
    $[\!]\ true \rightarrow new_2' := 0$
    $[\!]\ true \rightarrow new_2' := 5$

atom $A3$ **controls** $task_1$ **reads** $task_1$ **awaits** $new_1, proc$
  init
    $[\!]\ true \rightarrow task_1' := new_1'$
  update
    $[\!]\ proc' = 1 \rightarrow task_1' := task_1 + new_1' - 1$
    $[\!]\ proc' \neq 1 \rightarrow task_1' := task_1 + new_1'$

atom $A4$ **controls** $task_2$ **reads** $task_2$ **awaits** $new_2, proc$
  init
    $[\!]\ true \rightarrow task_2' := new_2'$
  update
    $[\!]\ proc' = 2 \rightarrow task_2' := task_2 + new_2' - 1$
    $[\!]\ proc' \neq 2 \rightarrow task_2' := task_2 + new_2'$

atom $A5$ **controls** $proc, prior$ **reads** $task_1, task_2, prior$
  init
    $[\!]\ true \rightarrow proc' := 0;\ prior' := 1$
    $[\!]\ true \rightarrow proc' := 0;\ prior' := 2$
  update
    $[\!]\ task_1 = 0 \wedge task_2 = 0 \rightarrow proc' := 0$
    $[\!]\ prior = 1 \wedge task_1 > 0 \rightarrow proc' := 1;\ prior' := 2$
    $[\!]\ prior = 1 \wedge task_1 = 0 \wedge task_2 > 0 \rightarrow proc' := 2$
    $[\!]\ prior = 2 \wedge task_2 > 0 \rightarrow proc' := 2;\ prior' := 1$
    $[\!]\ prior = 2 \wedge task_2 = 0 \wedge task_1 > 0 \rightarrow proc' := 1$

Figure 1.1: The five scheduler atoms

new$_1$	5	0	0	0	0	0	0	0	0	0	5	0	0	0	5
new$_2$	0	0	0	0	5	0	0	0	0	0	0	5	0	0	0
task$_1$	5	4	3	2	2	1	1	0	0	0	4	4	3	3	7
task$_2$	0	0	0	0	4	4	3	3	2	1	1	5	5	4	4
proc	0	1	1	1	2	1	2	1	2	2	1	2	1	2	1
prior	1	2	2	2	1	2	1	2	1	1	2	1	2	1	2

new$_1$	5	0	0	0	0	0	0	0	0	0	0	0	5	0	0
new$_2$	5	0	0	0	0	0	0	0	0	0	0	0	5	0	0
task$_1$	5	5	4	4	3	3	2	2	1	1	0	0	4	4	3
task$_2$	5	4	4	3	3	2	2	1	1	0	0	0	5	4	4
proc	0	2	1	2	1	2	1	2	1	2	0	0	1	2	1
prior	2	1	2	1	2	1	2	1	2	1	1	1	2	1	2

Figure 1.2: Two initialized trajectories of the scheduler

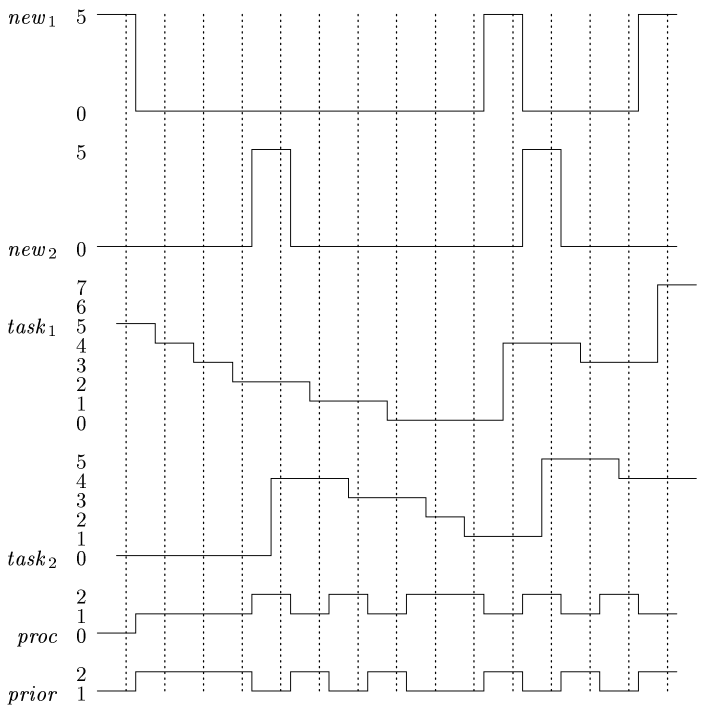

Figure 1.3: The timing diagram for the first trajectory from above

atom is called combinational. Thus, a combinational atom is a (generally nondeterministic) function that, given the values of the awaited variables at the end of an initialization or update round, computes the possible values for the controlled variables at the end of the round. In particular, for combinational atoms, the possible next values of some controlled variable x cannot depend on the current value of any variable, including x itself. By contrast, atoms that distinguish between initial and update rounds are called sequential.

> Combinational vs. sequential atom
>
> An atom U is combinational if (1) the set $\mathsf{read}X_U$ of read variables is empty, and (2) the initial command init_U and update command update_U are identical. Otherwise, U is a sequential atom.

Example 1.5 [Zero-delay vs. unit-delay copying] Given two variables y and x of the same type, we want y to duplicate the behavior of x. The combinational atom

atom *CombCopy* **controls** y **awaits** x
  initupdate
    $[\!]\ true \to y' := x'$

copies the value of x into y without delay. In the initial round, the atom waits for x being initialized, and assigns the initial value of x to y. In every update round, the atom waits for x being updated, and assigns the next value of x to y. Consequently, both y and x have the same value at the end of every round. The sequential atom

atom *SeqCopy* **controls** y **reads** x
  update
    $[\!]\ true \to y' := x$

copies the value of x into y with a delay of one round (the initial command is irrelevant for the purposes of this example). In every update round, the atom assigns the current value of x to y. Consequently, the value of y at the end of every update round is the same as the value of x at the beginning of the round. In Rml, combinational atoms can be recognized by the keyword **initupdate**. For example, the atoms $A1$ and $A2$ from Example 1.4 are combinational. ■

Lazy and eager atoms

An atom sleeps in an update round if the values of all controlled variables stay unchanged. An atom that may sleep in every update round is called lazy. The progress of a lazy atom cannot be enforced, because the atom may put off the next modification of the controlled variables for any number of rounds. By contrast, if certain current values of read variables or next values of awaited variables force an immediate change in value for a controlled variable, then the atom is called eager.

> Lazy vs. eager atom
>
> Given a finite set X of typed variables, the sleep assignment for X is the guarded assignment γ from X to X' with the guard $p_\gamma = \mathit{true}$ and the assignment $e_\gamma^{x'} = x$ for each variable $x' \in X'$. An atom U is lazy if the update command update_U contains the sleep assignment for the set $\mathsf{ctr}X_U$ of controlled variables. Otherwise, U is an eager atom.

Example 1.6 [Continuous vs. occasional copying] Both atoms *CombCopy* and *SeqCopy* from Example 1.5 are eager. In the first case, all modifications of y follow immediately, within the same round, the corresponding modifications of x; in the second case, the modifications of y are delayed by exactly one round. By contrast, the lazy atom

atom *LazyCopy* **controls** y **reads** y **awaits** x
  update
    $[\!]\ \mathit{true} \to y' := x'$
    $[\!]\ \mathit{true} \to y' := y$

copies the value of x into y at arbitrary times (the initial command is irrelevant for the purposes of this example). In every update round, either the value of y stays unchanged, or it is updated to the next value of x. Consequently, some values of x may not be copied into y. In Rml, the atom prefix **lazy** can be used instead of the sleep assignment for the update command. For example, the atom *LazyCopy* can alternatively be specified as

lazy atom *LazyCopy* **controls** y **reads** y **awaits** x
  update
    $[\!]\ \mathit{true} \to y' := x'$.

Note that the variable y is read, even though, because the keyword **lazy** is used, y does not literally occur in the update command as an unprimed symbol. ■

Remark 1.2 [Lazy implies sequential and, mostly, nondeterministic] Every lazy atom U reads its controlled variables; that is, $\mathsf{ctr}X_U \subseteq \mathsf{read}X_U$. It follows that all lazy atoms are sequential. Furthermore, with the exception of (trivial) atoms whose update commands contain only the sleep assignment, lazy atoms are nondeterministic. ■

Passive and active atoms

During an update round, an atom can notice changes in the values of awaited variables. If the awaited variable is also read, then the atom can directly compare the current value with the next value. If the awaited variable is not read, then the atom can remember the next value from the previous round, by storing it in a controlled variable, and compare it with the next value from the present round. Therefore, every change in the value of an awaited variable is an observable event. An atom is called passive if it may sleep in every update round in which no observable event occurs; that is, the atom may sleep whenever the values of all awaited variables stay unchanged. By contrast, if the value of a controlled variable is changed in certain update rounds independent of observable events, then the atom is called active. The active atoms are round-driven and the passive atoms are event-driven: while the progress of an active atom can be enforced by the expiration of rounds, the progress of a passive atom can be enforced only by other atoms that modify awaited variables.

> Passive vs. active atom
>
> Given two finite sets X and Y of typed variables, the conditional sleep assignment for X with respect to Y is the guarded assignment γ from $X \cup Y \cup Y'$ to X' with the guard $p_\gamma = (Y' = Y)$ and the assignment $e_\gamma^{x'} = x$ for each variable $x' \in X'$. An atom U is passive if U is either combinational, or lazy, or the update command update_U contains the conditional sleep assignment for the set $\mathsf{ctr}X_U$ of controlled variables with respect to the set $\mathsf{await}X_U$ of awaited variables. Otherwise, U is an active atom.

Remark 1.3 [Passive includes combinational and lazy] By definition, all combinational and lazy atoms are passive. This reflects the fact that conditional sleep assignments are redundant for combinational and lazy atoms: if the conditional sleep assignment is added to the update command of a combinational or lazy atom, then the behavior of the atom remains the same. For a lazy atom, this is trivially so. For a combinational atom, this is because if the values of the awaited variables do not change in an update round, then the atom may compute the same next values for the controlled variables as in the previous round. ■

Example 1.7 [Round vs. event counting] If the behavior of the nonnegative-integer variable n is defined by the active atom

atom *ActiveCount* **controls** n **reads** n
  init
    $[\!]\ \mathit{true} \rightarrow n' := 0$
  update
    $[\!]\ \mathit{true} \rightarrow n' := n + 1$

then the value of n at the end of the i-th update round is i. Thus, active atoms can count the number of rounds that expire. For example, an active atom may count the number of rounds that expire between two consecutive changes in the value of a variable x that is controlled by another atom. By contrast, passive atoms do not have the ability to count rounds; they can count only the number of observable events, such as the number of changes in the value of x. Specifically, if the behavior of n is defined by the passive atom

atom *PassiveCount* **controls** n **reads** n, x **awaits** x
  init
    $[\!]\ true \rightarrow n' := 0$
  update
    $[\!]\ x' \neq x \rightarrow n' := n + 1$
    $[\!]\ x' = x \rightarrow n' := n$

then the value of n at the end of the i-th update round is $j \leq i$, where j is the number of times that the value of x has changed during the first i update rounds. In Rml, the atom prefix **passive** can be used instead of the conditional sleep assignment for the update command. The atom *PassiveCount* is not a good example for illustrating the use of the keyword **passive**, however, because the conditional sleep assignment $x' = x \rightarrow n' := n$ coincides with the default assignment of the update command (if all guards are false, then the controlled variables stay unchanged), and therefore can be omitted with or without prefixing the atom description. A better example for the use of the keyword **passive** will follow in the next section. Now, for the record: the atoms *ActiveCount* (trivially) and *PassiveCount* are both sequential and eager. The combinational and lazy copiers *CombCopy* and *LazyCopy* from Examples 1.5 and 1.6 are passive (trivially), and the sequential, eager copier *SeqCopy* is active. This is because *SeqCopy* needs to be sensitive to the expiration of rounds in order to delay copying by exactly one round. ■

Remark 1.4 [Classification of atoms] The atoms can be partitioned into four pairwise disjoint classes: the combinational atoms, the lazy atoms, the active atoms, and the atoms that are sequential, eager, and passive. ■

1.1.3 Modules

A reactive module is a system, or system component, that interacts with other systems, or other components, which, collectively, make up the environment of the module. The behavior of some variables is controlled by the the module, and the behavior of other variables is controlled by the environment. We refer to the former as the controlled variables of the module, and to the latter as the environment variables. The controlled variables are partitioned into atoms, and so are the environment variables. In the initialization round and in every update round, the module and the environment take turns in the form of subrounds. In each subround of the initialization round, either the module initializes an atom of controlled variables, or the environment initializes an atom of environment variables. In each subround of an update round, either the module updates an atom of controlled variables, or the environment updates an atom of environment variables. Deadlock-freedom requires that, in the initialization round, the module is prepared to initialize its variables for all possible initial values of the environment variables, and in every update round, the module is prepared to update its variables for all possible current and next values of the environment variables.

In addition to being partitioned into atoms, the controlled variables are classified as to whether or not their values can be observed by the environment, and the environment variables are classified as to whether or not their values can be observed by the module. If a controlled variable is visible to the environment, then the updating of the environment variables may depend on the values of the controlled variable. Symmetrically, if an environment variable is visible to the module, then the updating of the controlled variables may depend on the values of the environment variable. Thus, a module description refers to three classes of variables —private, interface, and external. Each private variable can be read and modified by the module, and neither read nor modified by the environment. Each interface variable can be read by both the module and the environment, and modified by the module only. Each external variable can be read by both the module and the environment, and modified by the environment only. In other words: the module controls the private variables and the interface variables; the environment observes the interface variables and the external variables. The fourth class of variables —environment variables that are not visible to the module— is, naturally, not part of the module description.

> Module
>
> A (reactive) module P consists of a variable declaration and a set atoms_P of atoms. The variable declaration of P consists of three pairwise disjoint, finite sets of typed variables —the set $\mathsf{priv}X_P$ of private variables, the set $\mathsf{intf}X_P$ of interface variables, and the set $\mathsf{extl}X_P$ of external variables. We refer to $\mathsf{ctr}X_P = \mathsf{priv}X_P \cup \mathsf{intf}X_P$ as the controlled variables of P, to $\mathsf{obs}X_P = \mathsf{intf}X_P \cup \mathsf{extl}X_P$ as the observable variables, and to $X_P = \mathsf{ctr}X_P \cup \mathsf{obs}X_P$ as the module variables. The set atoms_P is a consistent set of X_P-atoms so that each variable $x \in X_P$ is controlled by some atom in atoms_P iff x is a controlled variable of P.

Terminology. For the controlled variables of a module P, by definition, $\mathsf{ctr}X_P = (\cup\ U \in \mathsf{atoms}_P \mid \mathsf{ctr}X_U)$. Similarly, we refer to $\mathsf{read}X_P = (\cup\ U \in \mathsf{atoms}_P \mid \mathsf{read}X_U)$ as the read variables of the module P, to $\mathsf{await}X_P = (\cup\ U \in \mathsf{atoms}_P \mid \mathsf{await}X_U)$ as the awaited variables of P, and to $\prec_P = \prec_{\mathsf{atoms}_P}$ as the await relation of P. The execution orders for atoms_P are called execution orders of P. ∎

Important special cases of modules are the finite, the closed, and the deterministic modules. A module is finite if all module variables can assume only finitely many values. A module is closed if the behavior of the controlled variables is not influenced by the behavior of any environment variables (although the behavior of some environment variables may be influenced by the behavior of the controlled variables). A module is deterministic if the behavior of the environment variables uniquely determines the behavior of the controlled variables.

> Finite, closed, and deterministic modules
>
> The module P is finite if all module variables in X_P have finite types; otherwise, P is an infinite module. The module P is closed if the set $\mathsf{extl}X_P$ of external variables is empty; otherwise, P is an open module. The module P is deterministic if all atoms in atoms_P are deterministic; otherwise, P is a nondeterministic module.

The execution of modules

A module P is executed by dividing every round into two phases. In the first phase of a round, the external variables of P are initialized or updated nondeterministically: each external variable obtains an arbitrary value of the appropriate type. In the second phase of the round, the controlled variables are initialized or updated by carrying out the initial or update commands of P in some execution order. The first phase ensures that all initial and next values of external variables are available should they be needed in the second phase. The atom consistency of P ensures, by Proposition 1.1, the existence of an execution order for the second phase. The outcome of the execution is an initialized trajectory of P, which gives a sequence of values for each variable in X_P. By Exercise 1.2, the choice of execution order does not influence the outcome of the execution. However, since the external variables are initialized and updated nondeterministically, and since the initial and update commands may be nondeterministic, a module can have many initialized trajectories. Only modules that are both closed and deterministic have, for any given number of rounds, a unique initialized trajectory.

The observable part of an initialized trajectory of the module P, which gives a sequence of values for each variable in $\mathsf{obs}X_P$, is called a trace of P. Thus, every trace of P shows a possible observable behavior of P over time. Since different initialized trajectories (outcomes of executions) may give rise to the same trace (observable behavior), even modules that are both closed and deterministic can have many traces of a given length. Formal definitions of trajectories and traces will be given in Chapters 2 and 5.

Example 1.8 [Scheduler] The scheduler from Example 1.4 can be built from the three modules whose Rml descriptions are given in Figure 1.4 without atom bodies. The modules *Task1* and *Task2* are closed; the module *Scheduler* is open. For illustration, we execute the module *Scheduler* in isolation. There are

module *Task1* **is**
  interface $new_1 : \mathbb{N}$
  atom $A1$ **controls** new_1

module *Task2* **is**
  interface $new_2 : \mathbb{N}$
  atom $A2$ **controls** new_2

module *Scheduler* **is**
  private $prior : \{1, 2\}$
  interface $task_1, task_2 : \mathbb{N};\ proc : \{0, 1, 2\}$
  external $new_1, new_2 : \mathbb{N}$
  atom $A3$ **controls** $task_1$ **reads** $task_1$ **awaits** $new_1, proc$
  atom $A4$ **controls** $task_2$ **reads** $task_2$ **awaits** $new_2, proc$
  atom $A5$ **controls** $proc, prior$ **reads** $task_1, task_2, prior$

Figure 1.4: The three scheduler modules

two execution orders, $A5, A3, A4$ and $A5, A4, A3$. In the first phase of the initial round, the external variables new_1 and new_2 are assigned arbitrary nonnegative integers, and in the second phase, the initial commands of the three atoms are executed in one of the two execution orders. In the first phase of every update round, the external variables new_1 and new_2 are assigned arbitrary new nonnegative integers, and in the second phase, the update commands of the three atoms are executed in some execution order. Since all initial and update commands are deterministic except for the initial value of the variable *prior*, for any two sequences of values for the external variables new_1 and new_2, and any initial value of *prior*, the module *Scheduler* has a unique initialized trajectory. The two trajectories of Figure 1.2 are initialized trajectories of the module *Scheduler*, and a third initialized trajectory is shown in Figure 1.5. In the third trajectory, the values of the external variables new_1 and new_2 are updated arbitrarily, in a manner that is not compliant with the modules *Task1* and *Task2*. If the values of the private variable *prior* are omitted from an initialized trajectory, we obtain a trace of the module *Scheduler*. ■

Block diagrams

We depict the structure of modules graphically using block diagrams. The block diagram for a module consists of delay elements and gates which are connected by wires, and of a module boundary. Each controlled variable is represented by a delay element whose output wire carries, in every update round, the current value of the variable. Each atom is represented by a gate whose

new_1	1	2	0	0	1	0	3	2	0	0	0	0	5	0	0
new_2	0	2	0	0	0	0	5	0	0	0	0	0	5	0	0
$task_1$	1	2	2	1	2	1	4	5	5	4	4	3	8	7	7
$task_2$	0	2	1	1	0	0	4	4	3	3	2	2	6	6	5
proc	0	1	2	1	2	1	2	1	2	1	2	1	2	1	2
prior	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1

new_1	1	2	0	0	1	0	3	2	0	0	0	0	5	0	0
new_2	0	2	0	0	0	0	5	0	0	0	0	0	5	0	0
$task_1$	1	2	2	1	2	1	4	5	5	4	4	3	8	7	7
$task_2$	0	2	1	1	0	0	4	4	3	3	2	2	6	6	5
proc	0	1	2	1	2	1	2	1	2	1	2	1	2	1	2

Figure 1.5: An initialized trajectory of the module *Scheduler* and the corresponding trace

output wires carry the next values of the variables that are controlled by the atom. The output wires of a gate are connected with the corresponding delay elements, where the updated values of the variables are stored for the next round. Thus there are two wires for each variable —one that carries the current value (delay output) and one that carries the next value (delay input) of the variable. The wires from delay elements to gates represent read dependencies between variables, and the wires from gates to gates represent await dependencies. Since the precedence relation on the atoms (gates) is asymmetric (Proposition 1.1), every wire cycle contains at least one delay element.

The delay elements and the gates of a module are circumscribed by a dotted line that denotes the module boundary. Each interface variable x is represented by two output wires, labeled x and x', that penetrate the module boundary from the inside to the outside. In every update round, the unprimed output wire carries the current value of the variable x, and the primed output wire carries the next value of x. Each external variable y is represented by one or two input wires, labeled y and y', that penetrate the module boundary from the outside to the inside. Since the module may use only the current value of y, or only the next value of y, the primed input wire or the unprimed input wire can be absent.

Example 1.9 [Scheduler] The block diagrams for the modules *Task1*, *Task2*, and *Scheduler* from Example 1.8 are shown in Figure 1.6. ■

Remark 1.5 [Block diagrams as types] The block diagram for a module contains the same information as the variable declarations and the atom declarations of

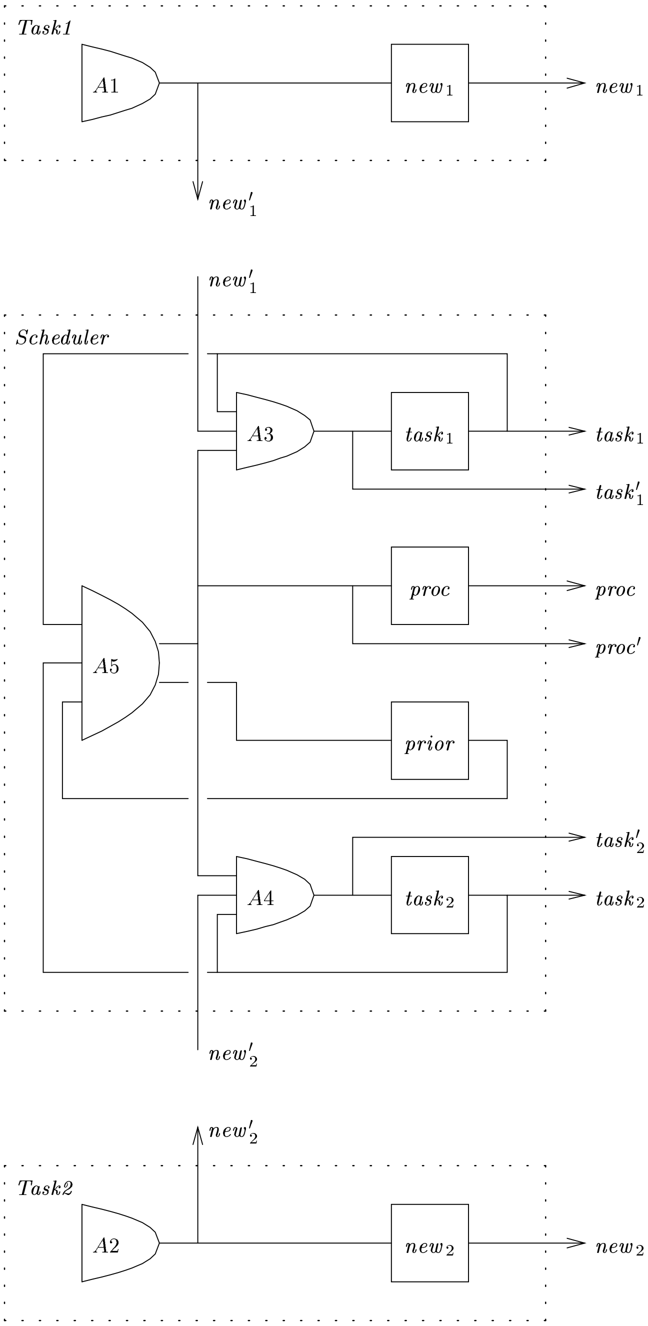

Figure 1.6: Block diagrams for the scheduler modules

the module. We call this information the type of the module. The type of a module does not include the initial and the update commands of the module. In particular, from the type of a module, it can be concluded if the module is finite, or closed, but not if the module is deterministic. ■

Asynchronous and synchronous modules

A module stutters in an update round if the values of all interface variables stay unchanged. A module that may stutter in every update round is called asynchronous. The environment cannot enforce observable progress of an asynchronous module. While an asynchronous module can privately record all updates of external variables, all updates of interface variables proceed at a speed that is independent of the environment speed. By contrast, a module that interacts with the environment synchronously may agree to modify an interface variable dependent on, and within the same round as, the modification of an external variable.

> Asynchronous vs. synchronous module
>
> The module P is asynchronous if all interface variables in $\mathsf{intf}X_P$ are controlled by lazy atoms. Otherwise, P is a synchronous module.

Example 1.10 [Zero-delay vs. unit-delay vs. buffered squaring] The module

module *SyncSquare* **is**
  interface $out\colon \mathbb{N}$
  external $in\colon \mathbb{N}$
  atom controls *out* **awaits** *in*
    initupdate
      $[\!]\ true \rightarrow out' := (in')^2$

waits, in every round, for the next value of the external nonnegative-integer variable *in*, computes the square, and displays the result in the interface variable *out*, all within the same round. This is done by a single, combinational atom. Thus, *SyncSquare* is an operator that transforms an input stream of nonnegative integers into an output stream of corresponding squares. The operator is synchronous, because every output value is produced in the very round in which the corresponding input value arrives. The module

module *DelayedSyncSquare* **is**
  interface $out\colon \mathbb{N}_\perp$
  external $in\colon \mathbb{N}$
  atom controls *out* **awaits** *in*
    init
      $[\!]\ true \rightarrow out' := \perp$
    update
      $[\!]\ true \rightarrow out' := in^2$

requires exactly one round to produce the square of an input value (initially, the output value is the undefined value $\perp$). Since a new output value cannot be delayed arbitrarily, the module *DelayedSyncSquare* is again synchronous.

By contrast, the asynchronous module *AsyncSquare* from Figure 1.7 implements an operator that requires an arbitrary number of rounds (possibly 0) for producing the square of an input value. From one update round to the next, the unprocessed input values are stored in the private queue *buffer*. If consecutive unprocessed input values are equal, only one representative is stored in *buffer*, and the square is computed only once. The queue *buffer* is updated in every round in which a new unprocessed input value arrives, so that no input values are lost. New output values, on the other hand, are produced after arbitrary delays. The module *AsyncSquare* therefore has two atoms. The atom *ComputeOut*, which controls *out*, is lazy: in every update round, it either computes a square and displays the result in the interface variable *out*, or it sleeps. If the queue *buffer* is empty, the square is computed for the external variable *in*; otherwise, the square is computed for the first element of the queue. The atom *StoreIn*, which controls *buffer*, is eager: in every update round, it waits both for the next input value and for the action taken by the atom *ComputeOut*, and reacts as follows. Whenever the input value changes and the queue *buffer* is nonempty —i.e., there are already some unprocessed input values— the new input value is added to the queue. The same happens if the input value changes and the queue is empty, but the output value does not change —i.e., the atom *ComputeOut* has decided to sleep. Whenever the output value changes (*ComputeOut* has decided to compute a square) and *buffer* is nonempty (the square is computed for the first element of the queue), the first element is removed from the queue. Note that, because the keyword **passive** is used, the update command of *StoreIn* contains both the conditional sleep assignment

$$[\!]\ \mathit{in}' = \mathit{in} \wedge \mathit{out}' = \mathit{out} \to \mathit{buffer}' := \mathit{buffer}$$

and the default assignment

$$[\!]\ \mathit{out}' \neq \mathit{out} \wedge \mathit{IsEmpty}(\mathit{buffer}) \to \mathit{buffer}' := \mathit{buffer}.$$

Figure 1.8 shows one, arbitrarily chosen, initialized trajectory for each of the three modules *SyncSquare*, *DelayedSyncSquare*, and *AsyncSquare*. For the modules *SyncSquare* and *DelayedSyncSquare*, which have no private variables, the traces coincide with the initialized trajectories. For the module *AsyncSquare*, we obtain the traces by omitting the values of the private queue *buffer* from the initialized trajectories. Every trace of the synchronous modules *SyncSquare* and *DelayedSyncSquare* is also a trace of the asynchronous module *AsyncSquare*, but the converse is not true. While *SyncSquare* and *DelayedSyncSquare* each have exactly one trace for any given sequence of input values, *AsyncSquare* may have infinitely many. ■

module *AsyncSquare* **is**
  private *buffer* : **queue of** $\mathbb{N}$
  interface *out* : $\mathbb{N}_\perp$
  external *in* : $\mathbb{N}$

  lazy atom *ComputeOut*
    controls *out*
    reads *out*, *buffer*
    awaits *in*
    init
      $[\!]$ $true \to out' := (in')^2$
      $[\!]$ $true \to out' := \perp$
    update
      $[\!]$ $IsEmpty(buffer) \to out' := (in')^2$
      $[\!]$ $\neg IsEmpty(buffer) \to out' := Front(buffer)^2$

  passive atom *StoreIn*
    controls *buffer*
    reads *in*, *out*, *buffer*
    awaits *in*, *out*
    init
      $[\!]$ $out' \neq \perp \to buffer' := EmptyQueue$
      $[\!]$ $out' = \perp \to buffer' := Enqueue(in', EmptyQueue)$
    update

$$[\!]\ \begin{bmatrix} \wedge\ in' \neq in \\ \wedge\ out' \neq out \\ \wedge\ \neg IsEmpty(buffer) \end{bmatrix} \to buffer' := Enqueue(in, Dequeue(buffer))$$

$$[\!]\ \begin{bmatrix} \wedge\ in' = in \\ \wedge\ out' \neq out \\ \wedge\ \neg IsEmpty(buffer) \end{bmatrix} \to buffer' := Dequeue(buffer)$$

$$[\!]\ \begin{bmatrix} \wedge\ in' \neq in \\ \wedge\ out' = out \end{bmatrix} \to buffer' := Enqueue(in, buffer)$$

Figure 1.7: Asynchronous squaring

in	1	2	2	3	3	3	4	4	4	4	5	6	7	8	9
out	1	4	4	9	9	9	16	16	16	16	25	36	49	64	81

in	1	2	2	3	3	3	4	4	4	4	5	6	7	8	9
out	$\perp$	1	4	4	9	9	9	16	16	16	16	25	36	49	64

in	1	2	2	3	3	3	4	4	4	4	5	6	7	8	9
out	$\perp$	$\perp$	1	4	9	9	9	9	9	16	25	25	25	25	36
buffer	1	1, 2	2	3			4	4	4			6	6, 7	6, 7, 8	7, 8, 9

Figure 1.8: Three initialized trajectories of the modules *SyncSquare* (top), *DelayedSyncSquare* (middle), and *AsyncSquare* (bottom)

Exercise 1.3 {P3} [Squaring inputs] Following Example 1.10, you are asked to implement two more operators that transform an input stream of nonnegative integers into an output stream of corresponding squares. (a) Define a module *AsyncSquare2* which, like *AsyncSquare*, requires an arbitrary number (possibly 0) of rounds to produce a square but, unlike *AsyncSquare*, computes the square of each individual input value, even if it is identical to the previous input value. Give an initialized trajectory of *AsyncSquare* such that the corresponding trace is not a trace of *AsyncSquare2*. (b) Define a module *SyncSquare2* which requires at least 2 and at most 5 rounds to produce a square. (c) Draw the block diagrams for the four modules *SyncSquare*, *SyncSquare2*, *AsyncSquare*, and *AsyncSquare2*. ■

Passive and active modules

A module sleeps in an update round if the values of all controlled variables stay unchanged. The environment stutters in an update round if the values of all external variables stay unchanged. A module that may sleep in every update round in which the environment stutters is called passive. The environment can enforce the progress of a passive module only by modifying external variables.

Passive vs. active module

The module P is passive if all atoms in atoms_P are passive. Otherwise, P is an active module.

Remark 1.6 [Progress of passive modules] As long as the environment stutters, a passive module may sleep: for a passive module we obtain an initialized trajectory of any given length by simply repeating initial values for all variables. In particular, a closed, passive module may sleep in every update round. The following exercise presents a generalization of this remark. ■

Exercise 1.4 {T2} [Stutter closure for passive modules] Consider a module P, and for each variable in X_P, consider a sequence of $k \geq 1$ values. Construct sequences of $k+1$ values by repeating the last value of each sequence of length k. (a) Prove that if P is passive and the sequences of length k form an initialized trajectory of P, then also the sequences of length $k+1$ form an initialized trajectory of P. (b) In part (a), can you replace "passive" by "asynchronous" and "initialized trajectory" by "trace"? ■

Exercise 1.5 {P1} [Classification of modules] A module may be asynchronous and passive, synchronous and passive, asynchronous and active, or synchronous and active. Give examples for all four classes of modules. ■

Private determinism

For a deterministic module, the behavior of the controlled variables, both private and interface, is uniquely determined by the behavior of the external variables. Thus, for a nondeterministic module, nondeterminism may manifest itself in the initialization and updating of private variables, or of interface variables, or both. If all nondeterminism is limited to interface variables, we call it private determinism: a module exhibits private determinism if the behavior of the private variables is uniquely determined by the behavior of the observable variables, both interface and external. It follows that for every privately deterministic module, there is a one-to-one correspondence between the initialized trajectories (i.e., the outcomes of executions) and the traces (i.e., the observable behaviors).

> Private determinism
>
> The module P has private determinism if all private variables in $\mathsf{priv}X_P$ are controlled by deterministic atoms. Otherwise, P has private nondeterminism.

Example 1.11 [Determinism vs. private determinism vs. nondeterminism] The modules *SyncSquare* and *DelayedSyncSquare* of Example 1.10 are deterministic. The module *AsyncSquare* is nondeterministic, because it requires an arbitrary number of rounds for processing an input value. However, *AsyncSquare* has private determinism, because the initial value of the private queue *buffer* is uniquely determined by the initial values of the observable variables *in* and *out*, and in every update round, the next value of *buffer* is uniquely determined by the current value of *buffer* and the current and next values of *in* and *out*. In this way, given a trace of *AsyncSquare*, we can construct a unique corresponding initialized trajectory. By contrast, the module *LossyAsyncSquare* of Figure 1.9, which implements asynchronous squaring using a lossy queue, does not have private determinism. The module *LossyAsyncSquare* shares the atom *ComputeOut* with the module *AsyncSquare*, but differs in the atom that controls the private

module *LossyAsyncSquare* **is**
  private *buffer*: **queue of** $\mathbb{N}$
  interface *out*: $\mathbb{N}_\perp$
  external *in*: $\mathbb{N}$

  lazy atom *ComputeOut*
    controls *out*
    reads *out*, *buffer*
    awaits *in*

  passive atom *LossyStoreIn*
    controls *buffer*
    reads *in*, *out*, *buffer*
    awaits *in*, *out*
    init
$$\begin{array}{ll} [\!] \; true & \rightarrow buffer' := EmptyQueue \\ [\!] \; out' = \perp & \rightarrow buffer' := Enqueue(in', EmptyQueue) \end{array}$$
    update
$$\begin{array}{ll} [\!] \begin{bmatrix} \wedge\ in' \neq in \\ \wedge\ out' \neq out \\ \wedge\ \neg IsEmpty(buffer) \end{bmatrix} & \rightarrow buffer' := Enqueue(in, Dequeue(buffer)) \\ [\!] \begin{bmatrix} \wedge\ out' \neq out \\ \wedge\ \neg IsEmpty(buffer) \end{bmatrix} & \rightarrow buffer' := Dequeue(buffer) \\ [\!] \begin{bmatrix} \wedge\ in' \neq in \\ \wedge\ out' = out \end{bmatrix} & \rightarrow buffer' := Enqueue(in, buffer) \end{array}$$

Figure 1.9: Lossy asynchronous squaring

in	1	2	2	3	3	3	4	5	6	7
out	$\perp$	1	1	1	1	1	1	9	9	9
buffer	1			3	3	3	3, 4	4, 5	4, 5	4, 5

in	1	2	2	3	3	3	4	5	6	7
out	$\perp$	1	1	1	1	1	1	9	9	9
buffer	1			3	3	3	3		6	6, 7

Figure 1.10: Two observably equivalent initialized trajectories of the module *LossyAsyncSquare*

queue *buffer*: whenever the input value changes, it may or may not be added to *buffer*, and thus, some input values can be lost. Furthermore, in any given update round, whether the next input value is lost or not is independent of the current value of *buffer* and independent of the current and next values of *in* and *out*. Figure 1.10 shows two distinct initialized trajectories of the module *LossyAsyncSquare* which give rise to the same trace. ■

1.2 Operations on Reactive Modules

We build complex modules from simple modules using three operations — parallel composition, variable renaming, and variable hiding.

1.2.1 Parallel Composition

The composition operation combines two modules into a single module whose behavior captures the interaction between the two component modules. Two modules can be composed only if their variable declarations are mutually consistent, and if the combined await dependencies of the two modules are not circular.

Module compatibility

The two modules P and Q are compatible if (1a) $\mathsf{priv}X_P$ and X_Q are disjoint, (1b) X_P and $\mathsf{priv}X_Q$ are disjoint, (1c) $\mathsf{intf}X_P$ and $\mathsf{intf}X_Q$ are disjoint, and (2) the transitive closure $(\prec_P \cup \prec_Q)^+$ is asymmetric.

Remark 1.7 [Independent modules] If the module variables of two modules are disjoint, then the two modules are compatible. ■

The composition operation is defined for compatible modules.

Module composition

If P and Q are two compatible modules, then the (parallel) composition $P\|Q$ is the module such that

- each private variable of a component module is a private variable of the compound module: $\mathsf{priv}X_{P\|Q} = \mathsf{priv}X_P \cup \mathsf{priv}X_Q$;
- each interface variable of a component module is an interface variable of the compound module: $\mathsf{intf}X_{P\|Q} = \mathsf{intf}X_P \cup \mathsf{intf}X_Q$;
- each external variable of a component module is an external variable of the compound module, provided it is not an interface variable of the other component: $\mathsf{extl}X_{P\|Q} = (\mathsf{extl}X_P \cup \mathsf{extl}X_Q)\backslash\mathsf{intf}X_{P\|Q}$;
- each atom of a component module is an atom of the compound module: $\mathsf{atoms}_{P\|Q} = \mathsf{atoms}_P \cup \mathsf{atoms}_Q$.

Remark 1.8 [Composing several modules] The composition operation on modules is commutative and associative. In Rml, we therefore omit parentheses when writing module expressions such as $P\|Q\|R$. ■

Example 1.12 [Scheduler] By composing the three modules from Example 1.8 we obtain the module

module *SchedulerSystem* **is** *Task1* $\|$ *Task2* $\|$ *Scheduler*.

The type of the compound module is

module *SchedulerSystem* **is**
  private $prior\colon \{1,2\}$
  interface $new_1, new_2\colon \mathbb{N};\ \ task_1, task_2\colon \mathbb{N};\ \ proc\colon \{0,1,2\}$
  atom $A1$ **controls** new_1
  atom $A2$ **controls** new_2
  atom $A3$ **controls** $task_1$ **reads** $task_1$ **awaits** $new_1, proc$
  atom $A4$ **controls** $task_2$ **reads** $task_2$ **awaits** $new_2, proc$
  atom $A5$ **controls** $proc, prior$ **reads** $task_1, task_2, prior$

and the corresponding block diagram is shown in Figure 1.11. In the pictorial representation of parallel composition, each primed output wire of one component module is connected with all primed input wires of other component modules that represent the same variable, and (not occurring in the scheduler example) each unprimed output wire is connected with the corresponding unprimed input wires. The module *SchedulerSystem* is infinite, closed, nondeterministic, synchronous (all atoms are eager), and active. While every initialized trajectory of the compound module *SchedulerSystem* is also an initialized trajectory of the component module *Scheduler*, the converse is not true. This is

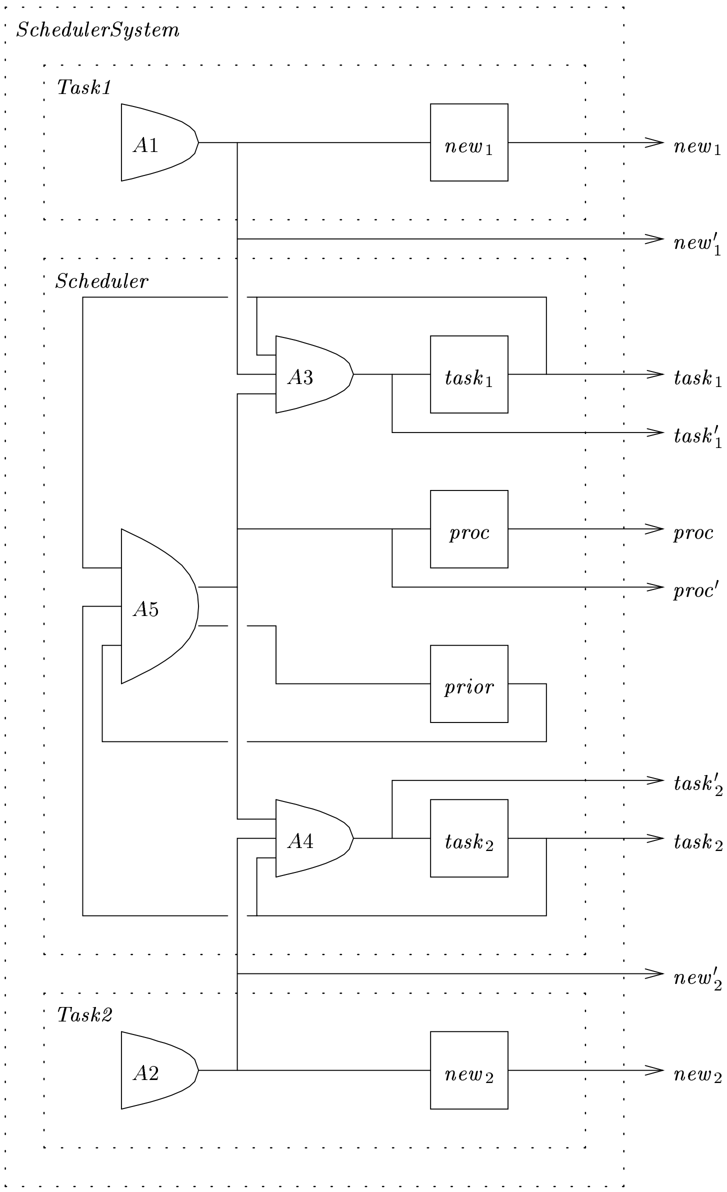

Figure 1.11: Block diagram for the scheduler system

because the component modules *Task1* and *Task2* constrain the behavior of the variables new_1 and new_2, which are external to *Scheduler*. For example, the two trajectories of Figure 1.2 are initialized trajectories of *SchedulerSystem*; the trajectory of Figure 1.5 is not. In Chapter 2 we will formally construct the trajectories of a compound module from the trajectories of the component modules. ■

Remark 1.9 [Module properties under parallel composition] The composition of two modules is finite iff both component modules are finite. The composition of two open modules may be closed. The composition of two modules is deterministic (or has private determinism) iff both component modules are deterministic (or have private determinism). The composition of two modules is asynchronous (or passive) iff both component modules are asynchronous (or passive). ■

Abstract block diagrams

In block diagrams, we may choose to hide the internal structure of a module and view it as a black box with input and output wires. If the atom structure of a module is suppressed, we draw the module boundary as a solid line instead of a dotted line. In order to compose such abstract block diagrams, every module needs to be annotated with information about the await dependencies between variables. The amount of compatibility information that is both necessary and sufficient is captured by the following definition. Given a module P, a derived await dependency $x \prec_P^d y$ of P consists of an external variable x and an interface variable y such that $x \prec_P^+ y$. The derived await dependency $x \prec_P^d y$ indicates that the initial value of the interface variable y may depend on the initial value of the external variable x, and in every update round, the next value of y may depend on the next value of x. Therefore, P cannot be composed with a module Q with external variable y, interface variable x, and $y \prec_Q^d x$.

Exercise 1.6 {T2} [Derived await dependencies] Consider two modules P and Q whose variables satisfy conditions (1a)–(1c) for module compatibility. (a) Show that the derived await dependencies of the two modules contain exactly the information that is necessary and sufficient for determining compatibility; that is, prove that P and Q are compatible iff the transitive closure $(\prec_P^d \cup \prec_Q^d)^+$ is asymmetric. (b) Assuming P and Q are compatible, construct the derived await dependencies of the compound module $P\|Q$ from the derived await dependencies of the component modules. ■

A block diagram for a module P must show either its internal structure or its derived await dependencies. Since it is immaterial if a derived await dependency is an actual await dependency (contained in $\prec_P$) or only "derived" from other await dependencies (contained in $\prec_P^+$), we often omit the superscript d from the symbol $\prec^d$.

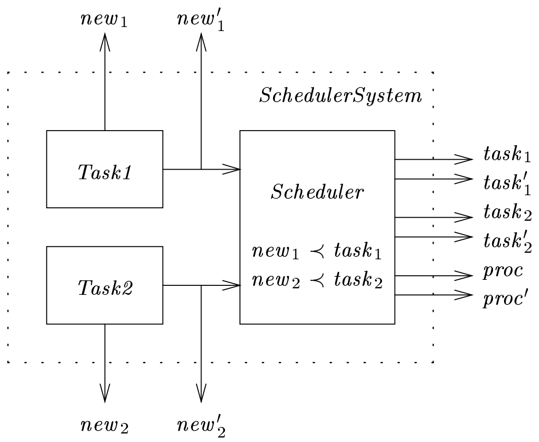

Figure 1.12: Abstract block diagram for the scheduler system

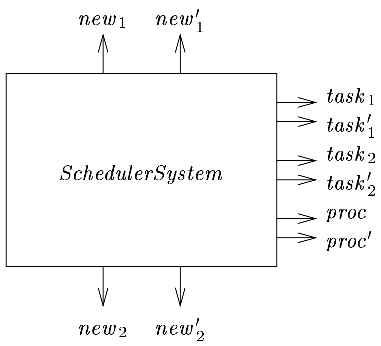

Figure 1.13: Very abstract block diagram for the scheduler system

Example 1.13 [Scheduler] The block diagram of Figure 1.12 does not show the atom structure of the component modules for the scheduler system from Example 1.12. The module *Scheduler* has two derived await dependencies, $new_1 \prec task_1$ and $new_2 \prec task_2$; for example, it cannot be composed with a module that awaits $task_1$ and controls new_1. The block diagram of Figure 1.13 further abstracts the internal structure of the compound module and views the entire scheduler system as a black box. The module *SchedulerSystem* is closed, and therefore has no derived await dependencies. ■

1.2.2 Variable Renaming

Before composing two modules, it may be necessary to rename private variables in order to make the two modules compatible. Variable renaming is also useful for identifying an interface variable of one module with an external variable of another module, and for creating multiple copies of a module.

> Variable renaming
>
> Let X be a finite set of typed variables, and let ρ be a renaming for X. By ρ' we denote the renaming for $X \cup X'$ such that for all variables $x \in X$, $x[\rho'] = x[\rho]$ and $x'[\rho'] = x[\rho]'$. Given an X-atom U, the renamed atom $U[\rho]$ is the $X[\rho]$-atom with the set $\mathsf{ctr}X_U[\rho]$ of controlled variables, the set $\mathsf{read}X_U[\rho]$ of read variables, the set $\mathsf{await}X_U[\rho]$ of awaited variables, the initial command $\mathsf{init}_U[\rho]$, and the update command $\mathsf{update}_U[\rho]$. Given a module P, and a renaming ρ for the set X_P of module variables, the renamed module $P[\rho]$ is the module with the set $\mathsf{priv}X_P[\rho]$ of private variables, the set $\mathsf{intf}X_P[\rho]$ of interface variables, the set $\mathsf{extl}X_P[\rho]$ of external variables, and the set $\{U[\rho] \mid U \in \mathsf{atoms}_P\}$ of atoms.

Example 1.14 [Scheduler] From the generic task module

module *Task* **is**
  interface *new* : $\mathbb{N}$
  atom controls *new*
    init update
      $[\!]$ *true* $\rightarrow$ *new*$' := 0$
      $[\!]$ *true* $\rightarrow$ *new*$' := 5$

we can construct the two task modules of the scheduler system from Example 1.12 by renaming. In Rml, we write

module *Task1* **is** *Task*[*new* := new_1]
module *Task2* **is** *Task*[*new* := new_2].

The interface variable *new* is renamed to create two distinct copies of the module *Task*. ■

Remark 1.10 [Module properties under variable renaming] Variable renaming preserves the cardinality (finite vs. infinite), closure (closed vs. open), determinism (deterministic vs. privately deterministic vs. nondeterministic), synchronicity (asynchronous vs. synchronous), and round-sensitivity (passive vs. active) properties of modules. ■

Implicit renaming of private variables

Two modules P and Q can be composed only if (1a) the private variables of P are disjoint from the module variables of Q, and (1b) the private variables of Q are disjoint from the module variables of P. If, however, we are not interested in the internal structures of P and Q, then we may not know the names of their private variables. To allow the parallel composition of modules in such circumstances, we treat the private variables of a module as "dummies," like the bound variables of a quantified formula (these variables can be renamed freely, and must be renamed suitably for safe substitution, before a free variable is replaced by an expression). In particular, in Rml we do not distinguish between modules that differ only in the names of private variables. Whenever we write $P \| Q$, we do not insist on conditions (1a) and (1b) for module compatibility, but rather assume that, implicitly, the private variables of P and Q are renamed suitably before the two modules are composed. (We still do insist, of course, on condition (1c) that the interface variables of P and Q are disjoint, and on condition (2) that the derived await dependencies of P and Q can be combined without introducing dependency cycles.)

1.2.3 Variable Hiding

The hiding of interface variables allows us to construct module abstractions of varying degrees of detail. For instance, after composing two modules, it may be appropriate to convert some interface variables to private variables, so that they are used only for the interaction of the component modules, and are no longer visible to the environment of the compound module.

> Variable hiding
>
> Given a module P, and an interface variable x in $\mathsf{intf}X_P$, by hiding x in P we obtain the module **hide** x **in** P with the set $\mathsf{priv}X_P \cup \{x\}$ of private variables, the set $\mathsf{intf}X_P \backslash \{x\}$ of interface variables, the set $\mathsf{extl}X_P$ of external variables, and the set atoms_P of atoms.

Remark 1.11 [Hiding several variables] In Rml, we write **hide** x_1, x_2 **in** P as an abbreviation for the module **hide** x_1 **in** (**hide** x_2 **in** P), which is identical to the module **hide** x_2 **in** (**hide** x_1 **in** P). ■

Example 1.15 [Scheduler] If we hide the interface variables $task_1$ and $task_2$ in the scheduler system from Example 1.14, we obtain the module

module *SchedulerSystem2* **is hide** $task_1, task_2$ **in** *SchedulerSystem*

whose block diagram is shown in Figure 1.14. Our conventions for the pictorial representation of variable renaming and variable hiding are evident from the figure. Hiding preserves the initialized trajectories of a module, but not the traces. A trace for the module *SchedulerSystem2* gives values only to the observable (interface) variables new_1, new_2, and *proc*. ■

Remark 1.12 [Module properties under variable hiding] Variable hiding preserves the cardinality, closure, and round-sensitivity properties of modules. Hiding preserves determinism, but may not preserve private determinism. By hiding a variable in a synchronous module we may obtain an asynchronous module. ■

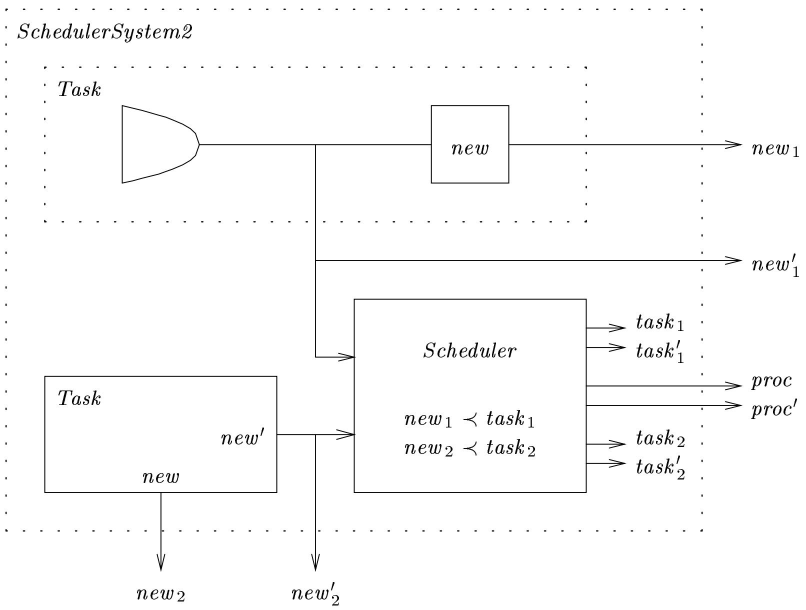

Figure 1.14: Block diagram for the scheduler system with renaming and hiding

Remark 1.13 [Abstract block diagrams as abstract types] Every block diagram for a module P, no matter how abstract, contains four pieces of information: the read external variables, the awaited external variables, the interface variables, and the derived await dependencies of P. We call this information the abstract type of the module. Since the names of private variables are immaterial, the abstract types of two modules suffice for determining if the two modules are compatible. Given a complex module that is built from simple modules using the three operations of composing, renaming, and hiding, and given the (abstract) types of all simple modules, we can infer the (abstract) type of the complex module. It is for this reason that the operations of composing, renaming, and hiding can be performed also on block diagrams. ■

1.3 Examples of Reactive Modules

We draw on examples from several application domains —synchronous and asynchronous hardware, concurrent programs with read-shared variables, and distributed programs with synchronous and asynchronous message passing.

1.3.1 Synchronous Circuits

Synchronous circuits are built from logic gates and memory cells that are driven by a sequence of clock ticks. Each logic gate computes a boolean value once per clock cycle, and each memory cell stores a boolean value from one clock cycle to the next. We model each logic gate and each memory cell as a reactive module so that every update round represents a clock cycle. The wires that connect the logic gates and the memory cells are modeled as boolean variables. As is customary in circuit design, we denote the values of wires by 0 and 1 instead of *false* and *true*, respectively. We construct synchronous circuits from three basic building blocks: as basic logic gates we use the Not gate and the And gate, and as basic memory cell we use the latch (set-reset flip-flop). These building blocks are then combined to circuits by applying the three module operations of parallel composition, variable renaming, and variable hiding.

Combinational circuits

Figure 1.15 defines three deterministic, synchronous, passive modules for modeling Not, And, and Or gates. The module *SyncNot* models a zero-delay Not gate, which takes a boolean input and produces a boolean output. The input is modeled as an external variable, *in*, because it is modified by the environment and visible to the gate. The output is modeled as an interface variable, *out*, because it is modified by the gate and visible to the environment. In the initial round, the Not gate waits for the input value to be initialized before computing the initial output value, by negating the initial input value. In every update round, the Not gate waits for the input value to be updated before computing the next output value, by negating the updated input value. The module *SyncNot* is passive, because the output changes only if the input changes, and synchronous, because the output changes in the very round (clock cycle) in which the input changes (zero delay). The module *SyncAnd* models a zero-delay And gate in similar fashion. The And gate takes two boolean inputs, represented by the external variables in_1 and in_2, and produces a boolean output, represented by the interface variable *out*. In the initial round, both input values must be initialized before the gate issues the initial output value. In every update round, both input values must be updated before the gate issues the next output value with zero delay.

From Not and And gates we can build all combinational circuits. For example, by de Morgan's law, a zero-delay Or gate can be defined by composing a zero-delay And gate with three zero-delay Not gates that negate both inputs and the output of the And gate. The resulting module *SyncOr* has the same abstract type as the module *SyncAnd* —two awaited boolean inputs represented by the external variables in_1 and in_2, and a boolean output represented by the interface variable *out* which depends on both inputs (the long dashes in Figure 1.15 indicate Rml commentary). The private variables z_1, z_2, and z_3 of *SyncOr*

module *SyncNot* **is**
  interface $out : \mathbb{B}$
  external $in : \mathbb{B}$
  atom controls out **awaits** in
    initupdate
      $[\!] \; in' = 0 \rightarrow out' := 1$
      $[\!] \; in' = 1 \rightarrow out' := 0$

module *SyncAnd* **is**
  interface $out : \mathbb{B}$
  external $in_1, in_2 : \mathbb{B}$
  atom controls out **awaits** in_1, in_2
    initupdate
      $[\!] \; in'_1 = 0 \rightarrow out' := 0$
      $[\!] \; in'_2 = 0 \rightarrow out' := 0$
      $[\!] \; in'_1 = 1 \wedge in'_2 = 1 \rightarrow out' := 1$

module *SyncOr* **is**
  —**interface** out
  —**external** in_1, in_2
  hide z_1, z_2, z_3 **in**
    $\| \; SyncAnd[in_1, in_2, out := z_1, z_2, z_3]$
    $\| \; SyncNot[in, out := in_1, z_1]$
    $\| \; SyncNot[in, out := in_2, z_2]$
    $\| \; SyncNot[in, out := z_3, out]$

Figure 1.15: Synchronous Not, And, and Or gates

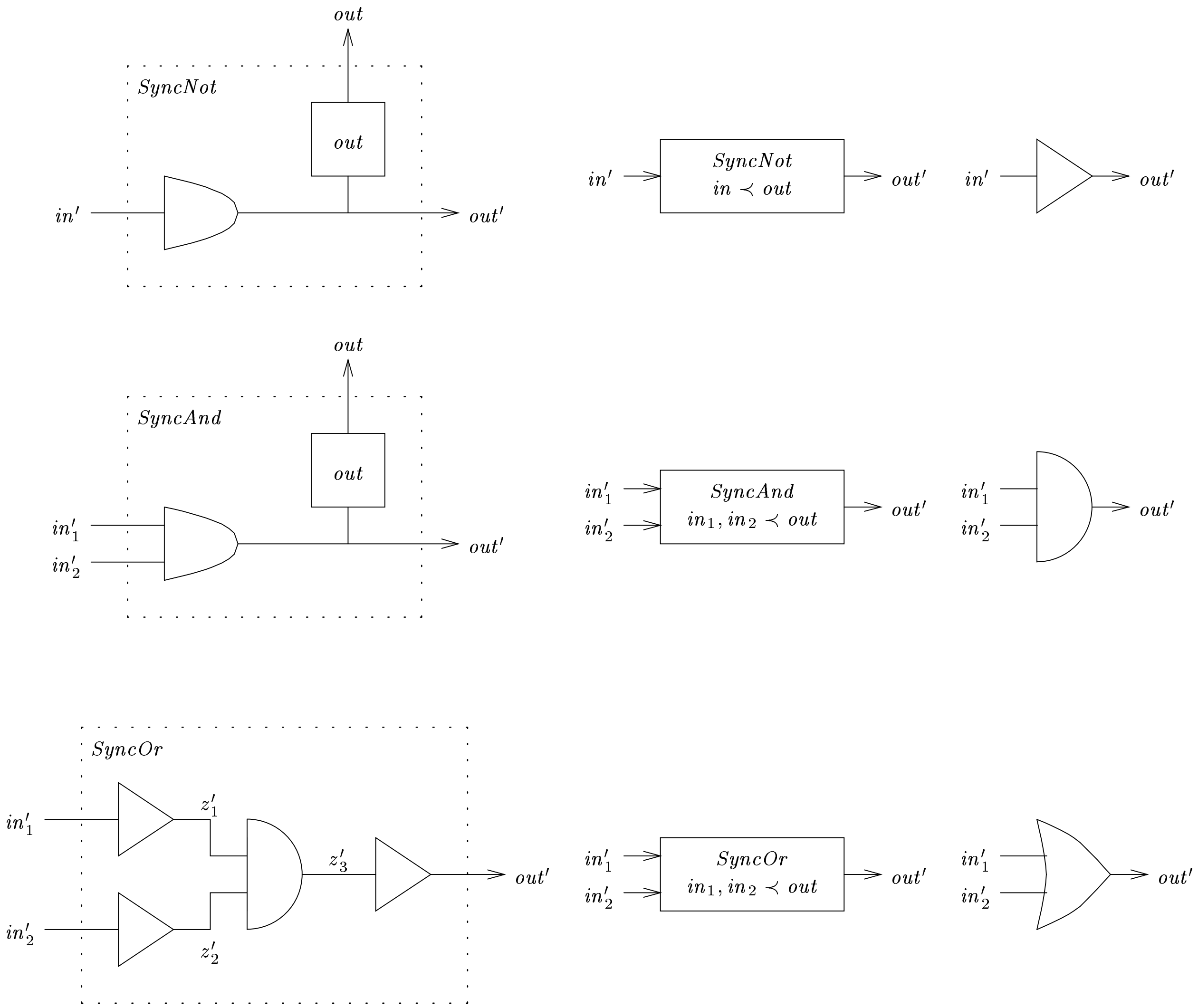

Figure 1.16: Block diagrams for the synchronous Not, And, and Or gates

represent internal wires that connect the four component gates. The values of these internal wires can be neither read nor modified by the environment. Since *out* waits for z_3, which waits for both z_1 and z_2, which wait for in_1 and in_2, respectively, we obtain the two derived await dependencies $in_1 \prec out$ and $in_2 \prec out$; that is, every update of the output must be preceded by updates of both inputs. The module *SyncOr* is again passive (all atoms are combinational) and synchronous (an eager atom controls the output).

Figure 1.16 shows, in the left column, detailed block diagrams for the zero-delay Not, And, and Or gates, and in the center column, corresponding abstract block diagrams. We omit the unprimed output wires from the abstract block diagrams of logic gates, because they are not used for building circuits (to save the value of a wire from one clock cycle of a synchronous circuit to the subsequent clock cycle, the value must be latched). We abbreviate the abstract block diagrams of the logic gates using module boundaries of different shapes which resemble the standard gate symbols. This allows us to suppress the derived await dependencies. The abbreviations are shown in the right column of Figure 1.16.

Sequential circuits

Figure 1.17 defines a nondeterministic, synchronous, active module for modeling a unit-delay latch. The latch takes two boolean inputs, represented by the external variables *set* and *reset*, and produces a boolean output, represented by the interface variable *out*. Unlike the logic gates, the latch has a boolean state, which is represented by the private variable *state*. The latch behaves like a Moore machine. In every update round, the latch first issues its state as output and then waits for the updated input values to compute its next state. If the updated value of *set* is 1 and the updated value of *reset* is 0, then the latch changes its state to 1. If the updated value of *set* is 0 and the updated value of *reset* is 1, then the latch changes its state to 0. If both updated input values are 0, then the state of the latch (which is equal to the already updated output out') does not change. If both updated input values are 1, then the next state of the latch is arbitrary —it may be either 0 or 1 (in this case, two of the guards apply, and the value of *state* is updated nondeterministically). What remains to be specified are the initial values of *out* and *state*. The initial output of the latch is arbitrary (this is a second source of nondeterminism). The initial state of the latch is computed combinationally from the initial values of *out*, *set*, and *reset* as during update rounds (in particular, if both *set* and *reset* are initially 0, then the initial value of *state* is determined by the initial value of *out*).

The resulting module *SyncLatch* is active, because in every round a new output is issued independently of any input change, and synchronous, because after an input change the output changes in the very next round (unit delay). Figure 1.18 shows the detailed block diagram for the latch and, below, an abstract

$$
\begin{array}{l}
\textbf{module } \mathit{SyncLatch} \textbf{ is} \\
\quad \textbf{private } \mathit{state} : \mathbb{B} \\
\quad \textbf{interface } \mathit{out} : \mathbb{B} \\
\quad \textbf{external } \mathit{set}, \mathit{reset} : \mathbb{B} \\
\quad \textbf{atom } \mathit{ComputeOutput} \textbf{ controls } \mathit{out} \textbf{ reads } \mathit{state} \\
\qquad \textbf{init} \\
\qquad\quad [\!] \ \mathit{true} \to \mathit{out}' := \mathbb{B} \\
\qquad \textbf{update} \\
\qquad\quad [\!] \ \mathit{true} \to \mathit{out}' := \mathit{state} \\
\quad \textbf{atom } \mathit{ComputeNextState} \textbf{ controls } \mathit{state} \textbf{ awaits } \mathit{out}, \mathit{set}, \mathit{reset} \\
\qquad \textbf{initupdate} \\
\qquad\quad [\!] \ \mathit{set}' = 1 \qquad\qquad\qquad\quad \to \mathit{state}' := 1 \\
\qquad\quad [\!] \ \mathit{reset}' = 1 \qquad\qquad\qquad \to \mathit{state}' := 0 \\
\qquad\quad [\!] \ \mathit{set}' = 0 \wedge \mathit{reset}' = 0 \to \mathit{state}' := \mathit{out}'
\end{array}
$$

Figure 1.17: Synchronous latch

block diagram. As with logic gates, we omit the unprimed output wire from the abstract block diagram of the latch, because it is not used for building circuits. Note that while zero-delay logic gates have (derived) await dependencies between inputs and outputs, the unit-delay latch does not. For this it was necessary to model the latch with two atoms, each controlling one variable, rather than with a single atom controlling both variables: in the module *SyncLatch*, the state variable *state* waits for the input variables *set* and *reset*; the output variable *out* does not. This decoupling of the output computation, which requires no inputs, from the next-state computation, which requires both inputs, into separate subrounds is essential for composing latches with logic gates which, in every round (clock cycle), provide the latch inputs dependent on the latch outputs.

Example 1.16 [Binary counter] As an example of a sequential circuit, we design a three-bit binary counter. The counter takes two boolean inputs, represented by the external variables *start* and *inc*, for starting and incrementing the counter. The counter value ranges from 0 to 7, and is represented by three bits. We do not make any assumption about the initial counter value. A start command resets the counter value to 0 and overrides any increment command that is issued in the same round. An increment command increases the counter value by 1. If the counter value is 7, the increment command changes the counter value to 0. In every round, the counter issues its value as output —the low bit on the interface variable out_0, the middle bit on the interface variable out_1, and the high bit on the interface variable out_2. (While combinational circuits are

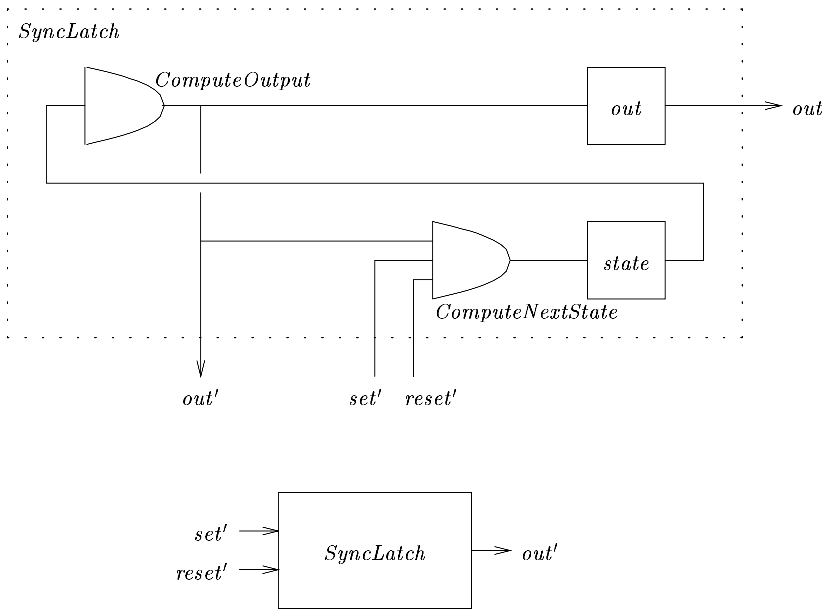

Figure 1.18: Block diagrams for the synchronous latch

passive, sequential circuits are active.)

Figure 1.19 shows a possible design of the three-bit counter from three one-bit counters (for clarity, in Rml we can annotate the component modules of a compound module with the names of the observable variables even if the variables are not renamed). Note that $carry_0$ waits for both *start* and *inc*, then $carry_1$ waits for $carry_0$, and $carry_2$ waits for $carry_1$. It follows that all three bits of the counter are updated in a single round (clock cycle). Figure 1.20 shows block diagrams for the one-bit counter *Sync1BitCounter* and the three-bit counter *Sync3BitCounter*. The module *Sync3BitCounter* has no derived await dependencies; it is finite, open, nondeterministic (because the initial counter value is arbitrary), privately deterministic, synchronous, and active. Figure 1.21 shows an initial trajectory of *Sync3BitCounter* and, for some of the variables, the corresponding timing diagram. (The private variables of the three one-bit counters have been renamed implicitly; for instance, z has been renamed to z_0, z_1, and z_2.) ■

Exercise 1.7 {P3} [Synchronous circuits] (a) Define a passive module *SyncNor* that models a zero-delay Nor gate. Use the variable names in_1 and in_2 for

module *Sync1BitCounter* **is**
—**interface** $out, carry$
—**external** $start, inc$
hide $set, reset, z$ **in**
$\|$ $SyncLatch[set, reset, out]$
$\|$ $SyncAnd[in_1, in_2, out := out, inc, carry]$
$\|$ $SyncOr[in_1, in_2, out := carry, start, reset]$
$\|$ $SyncNot[in, out := reset, z]$
$\|$ $SyncAnd[in_1, in_2, out := inc, z, set]$

module *Sync3BitCounter* **is**
—**interface** out_0, out_1, out_2
—**external** $start, inc$
hide $carry_0, carry_1, carry_2$ **in**
$\|$ $Sync1BitCounter[start, inc, out, carry := start, inc, out_0, carry_0]$
$\|$ $Sync1BitCounter[start, inc, out, carry := start, carry_0, out_1, carry_1]$
$\|$ $Sync1BitCounter[start, inc, out, carry := start, carry_1, out_2, carry_2]$

Figure 1.19: One-bit and three-bit binary counters

input, and use *out* for output. (b) Why is

hide z **in**
$\|$ $SyncNor[in_1, in_2, out := set, z, out]$
$\|$ $SyncNor[in_1, in_2, out := reset, out, z]$

not a legal definition of a module? (c) Consider the module

module *SyncDelay* **is**
private $state : \mathbb{B}$
interface $out : \mathbb{B}$
external $in : \mathbb{B}$
atom *ComputeOutput* **controls** *out* **reads** *state*
atom *ComputeNextState* **controls** *state* **awaits** *in*
initupdate
$[\!]$ $true \rightarrow state' := in'$

which shares the atom *ComputeOutput* with the module *SyncLatch* from Figure 1.17. Give a few initialized trajectories of the module *SyncDelay*. Then characterize, in precise words, the set of all initialized trajectories of *SyncDelay*. Is the module *SyncDelay* finite? Closed? Deterministic? Privately determinis-

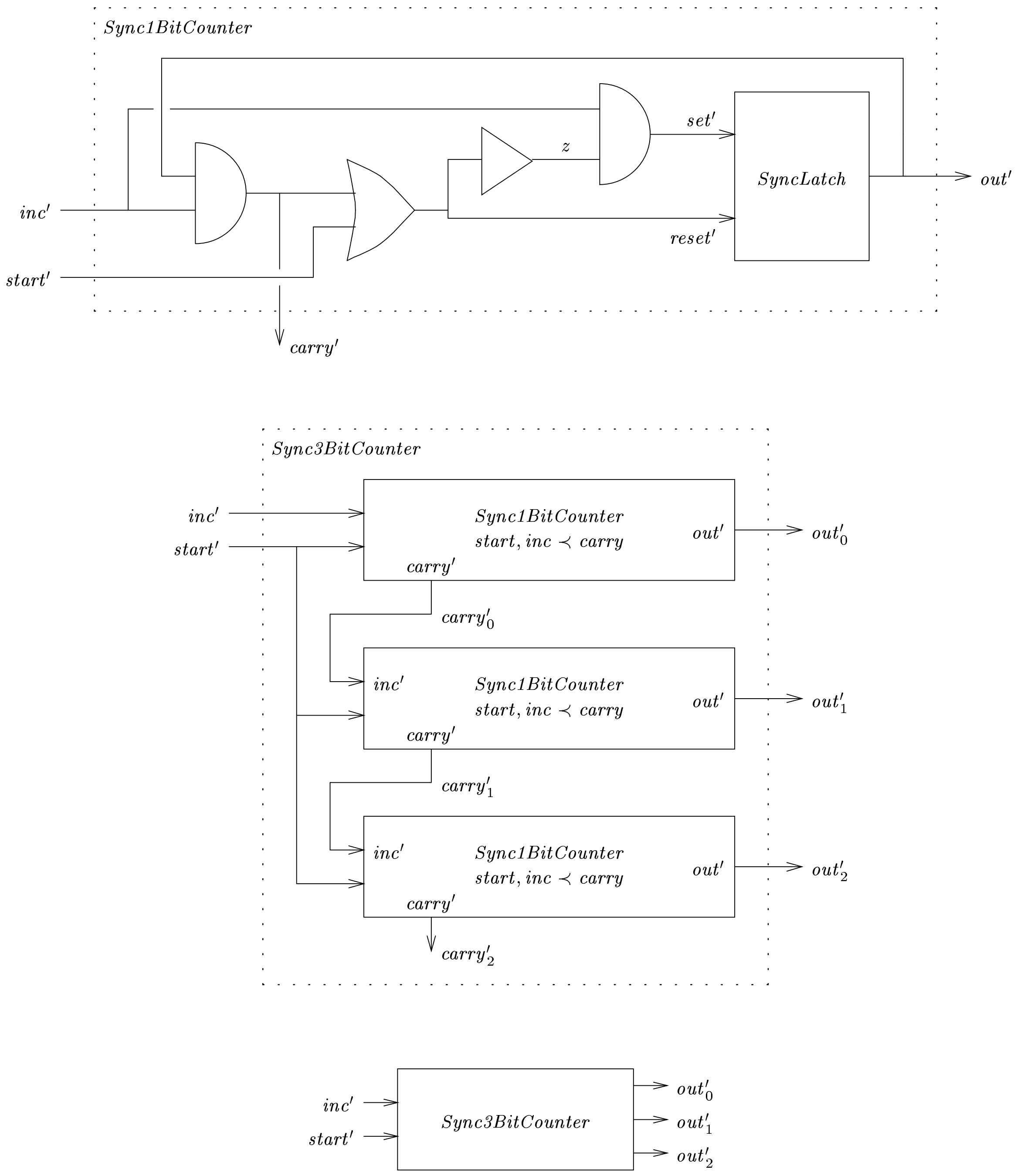

Figure 1.20: Block diagrams for the one-bit and three-bit binary counters

inc	1	0	1	1	1	1	1	1	0	0	0	1	1	0	1
start	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
z_0	0	1	1	0	0	1	0	1	1	1	1	0	1	0	1
set_0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	1
$reset_0$	1	0	0	1	1	0	1	0	0	0	0	1	0	1	0
$carry_0$	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0
z_1	0	1	1	1	0	1	1	1	1	1	1	0	1	0	1
set_1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
$reset_1$	1	0	0	0	1	0	0	0	0	0	0	1	0	1	0
$carry_1$	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
z_2	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1
set_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$reset_2$	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
$carry_2$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
out_0	1	0	0	1	0	0	1	0	1	1	1	1	0	1	0
out_1	1	0	0	0	1	0	0	1	1	1	1	1	0	0	0
out_2	0	1	1	1	1	0	0	0	0	0	0	0	1	1	0

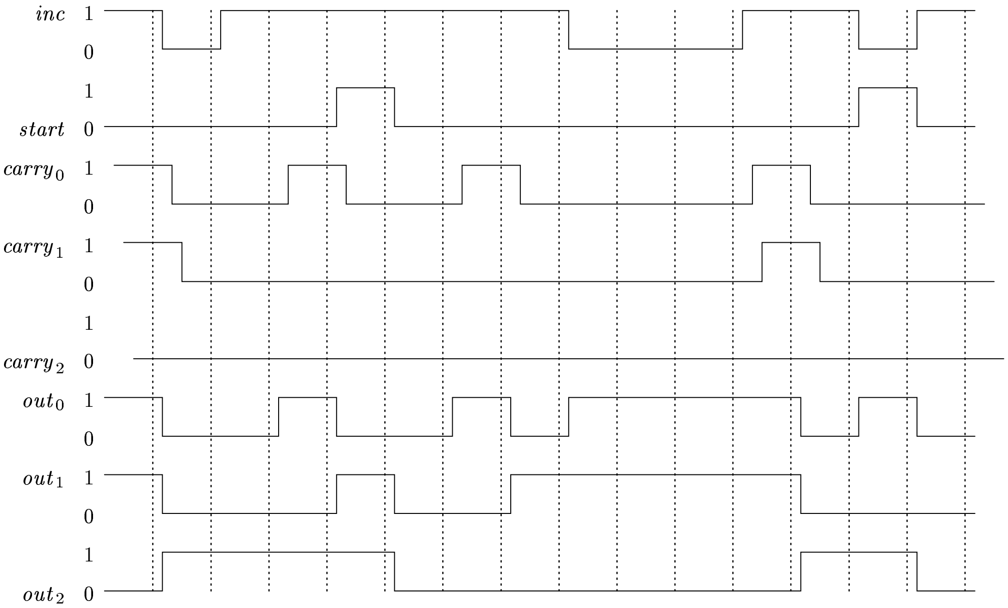

Figure 1.21: An initialized trajectory of the module *Sync3BitCounter*

tic? Asynchronous? Passive? (d) Draw block diagrams for the module

> **module** *SyncLatch2* **is**
> —**interface** out
> —**external** $set, reset$
> **hide** z_1, z_2, z_3 **in**
> $\|$ $SyncNor[in_1, in_2, out := set, z_3, z_1]$
> $\|$ $SyncNor[in_1, in_2, out := reset, out, z_2]$
> $\|$ $SyncDelay[in, out := z_1, out]$
> $\|$ $SyncDelay[in, out := z_2, z_3]$

at three different levels of abstraction. Compare the abstract type of the module *SyncLatch2* with the abstract type of the module *SyncLatch*. Give an initialized trajectory of *SyncLatch2* and draw its timing diagram. How do the traces of *SyncLatch2* differ from the traces of *SyncLatch*? Can the traces of *SyncLatch* be matched by removing one of the component modules from the compound module *SyncLatch2*? ■

1.3.2 Shared-variables Protocols

We refer to concurrent programs that communicate through read-shared variables as processes. We model each process as a reactive module with a single atom. The sequential control of a process is often encoded by a controlled variable called pc, which stands for "program counter." The interface variables of a process can be read by other processes. All inter-process communication occurs in this way, by processes reading (rather than awaiting) external variables, which are controlled by other processes. Hence, there are no awaited variables: in every update round, each process determines the next values of the controlled variables based solely on the current values of variables. Processes are combined by applying the three module operations of parallel composition, variable renaming, and variable hiding. In the synchronous case, all processes proceed in lock-step —one step per update round. In the asynchronous case, each process may or may not proceed in any given update round.

The mutual-exclusion problem

A paradigmatic problem in concurrent programming is the mutual-exclusion problem, which asks for a programming solution to ensure that no two processes simultaneously access a common resource, such as an I/O device or a write-shared variable. We illustrate the use of reactive modules for modeling concurrent programs by modeling two protocols —one synchronous, the other asynchronous— which solve the mutual-exclusion problem. We restrict our attention to the two-process case. Without loss of generality, we assume that each process has a so-called "critical section," which contains all accesses to the common resource. The interface variable pc_1 of the first process indicates if the process control is outside the critical section ($pc_1 = outC$), requesting to enter the critical section ($pc_1 = reqC$), or inside the critical section ($pc_1 = inC$). The interface variable pc_2 of the second process indicates the status of the second process in the same manner. Each process starts outside its critical section: the initial command for pc_1 is

init
$[\!]\ true \to pc_1' := outC$

and similarly for pc_2. Each process may remain outside its critical section for an arbitrary number of rounds, and it may remain inside the critical section for an arbitrary number of rounds. In other words, each process may request to enter the critical section at any time, and it may leave the critical section at any time. We model these assumptions using nondeterminism: the update command for pc_1 contains the guarded assignments

update
$[\!]\ pc_1 = outC \to$
$[\!]\ pc_1 = outC \to pc_1' := reqC$
$[\!]\ pc_1 = inC \quad \to$
$[\!]\ pc_1 = inC \quad \to pc_1' := outC$

and similarly for pc_2. Since the program counters pc_1 and pc_2 are interface variables, in every update round, each process "knows" about the current status of the other process. Our task is to add guarded assignments that permit each process to enter its critical section, by updating pc_1 or pc_2 from $reqC$ to inC, in a controlled fashion. We say that the i-th process has the opportunity to enter the critical section if the guard is true for some guarded command that sets pc_i to inC. (If a process has the opportunity to enter the critical section, the process does not necessarily need to enter, because it may have additional nondeterministic choices.)

In a correct solution to the mutual-exclusion problem, the parallel composition of both processes has to meet several requirements. First and foremost is the requirement of mutual exclusion: it must not happen, ever, that both processes are inside their critical sections simultaneously. The mutual-exclusion requirement can be enforced easily, say, by never permitting the second process to enter the critical section. This, however, is not a satisfactory solution and is ruled out by the following, second requirement, which is called accessibility if either of the processes requests to enter the critical section, then in the current or some future round, the process will have the opportunity to enter; furthermore, this opportunity will present itself no matter how the other process behaves —i.e., how it resolves its nondeterministic choices— as long as the other process does not stay inside the critical section forever.

Synchronous mutual exclusion

The protocol *SyncMutex* of Figure 1.22 provides a synchronous solution to the mutual-exclusion problem. The first process, Q_1, proceeds into its critical section when the second process is not in its critical section (but may be requesting to enter), and the second process, Q_2, proceeds into its critical section when the first process is outside its critical section (and not requesting to enter). In particular, if both processes are trying to enter their critical sections in the same round, only the first process will succeed. In that case, the second process will enter its critical section as soon as the first process leaves its critical section. This guarantees accessibility. Both processes proceed synchronously, because if a process tries to enter its critical section, then it will proceed into the critical section in the first round in which it is permitted to do so.

Exercise 1.8 {P2} [Synchronous mutual exclusion] The module *SyncMutex* is active, because if a process is permitted to enter its critical section in the first round in which it tries to enter, then the process proceeds into the critical section in the same round, without waiting for a change in the value of the external variable. Modify the protocol *SyncMutex* to obtain a synchronous, passive solution to the mutual-exclusion problem. (Do not use additional variables.) ■

Asynchronous mutual exclusion

In the asynchronous model of concurrent programming, all processes proceed at independent, and possibly varying, speeds. The assumption of speed independence abstracts details about the execution of a concurrent program: it captures parallel implementations on multiple processors of unknown speeds, as well as time-sharing implementations on a single processor with an unknown scheduling policy. In reactive modules, each speed-independent process is specified by a lazy atom. Then, in every update round, each speed-independent process may either proceed (i.e., the values of some controlled variables change) or sleep (i.e., the values of all controlled variables stay unchanged). For concurrent programs with read-shared variables, the assumption of speed independence is captured formally by the following definition.

> Speed-independent process set
>
> A speed-independent process is a lazy atom without awaited variables. A speed-independent process set is a module all of whose atoms are speed-independent processes.

Remark 1.14 [Properties of speed-independent process sets] Every speed-independent process set is both asynchronous and passive. The speed-independent process sets are closed under parallel composition, variable renaming, and variable hiding; that is, if these operations are applied to speed-independent process sets, then the results are again speed-independent process sets. ■

$$
\begin{array}{l}
\textbf{module } Q_1 \textbf{ is} \\
\quad \textbf{interface } pc_1 : \{outC, reqC, inC\} \\
\quad \textbf{external } pc_2 : \{outC, reqC, inC\} \\
\quad \textbf{atom controls } pc_1 \textbf{ reads } pc_1, pc_2 \\
\qquad \textbf{init} \\
\qquad\quad [\!]\ true \to pc_1' := outC \\
\qquad \textbf{update} \\
\qquad\quad \begin{array}{lll}
[\!]\ pc_1 = outC & \to & \\
[\!]\ pc_1 = outC & \to & pc_1' := reqC \\
[\!]\ pc_1 = reqC \wedge pc_2 \neq inC & \to & pc_1' := inC \\
[\!]\ pc_1 = inC & \to & \\
[\!]\ pc_1 = inC & \to & pc_1' := outC
\end{array} \\
\\
\textbf{module } Q_2 \textbf{ is} \\
\quad \textbf{interface } pc_2 : \{outC, reqC, inC\} \\
\quad \textbf{external } pc_1 : \{outC, reqC, inC\} \\
\quad \textbf{atom controls } pc_2 \textbf{ reads } pc_1, pc_2 \\
\qquad \textbf{init} \\
\qquad\quad [\!]\ true \to pc_2' := outC \\
\qquad \textbf{update} \\
\qquad\quad \begin{array}{lll}
[\!]\ pc_2 = outC & \to & \\
[\!]\ pc_2 = outC & \to & pc_2' := reqC \\
[\!]\ pc_2 = reqC \wedge pc_1 = outC & \to & pc_2' := inC \\
[\!]\ pc_2 = inC & \to & \\
[\!]\ pc_2 = inC & \to & pc_2' := outC
\end{array} \\
\\
\textbf{module } SyncMutex \textbf{ is } Q_1 \,\|\, Q_2
\end{array}
$$

Figure 1.22: Synchronous mutual exclusion

module P_1 **is**
  interface $pc_1 : \{outC, reqC, inC\};\ x_1 : \mathbb{B}$
  external $pc_2 : \{outC, reqC, inC\};\ x_2 : \mathbb{B}$
  lazy atom controls pc_1, x_1 **reads** pc_1, pc_2, x_1, x_2
    init
      $[\!]\ true \to pc_1' := outC;\ x_1' := \mathbb{B}$
    update
      $[\!]\ pc_1 = outC \quad \to pc_1' := reqC;\ x_1' := x_2$
      $[\!]\ pc_1 = reqC \wedge (pc_2 = outC \vee x_1 \neq x_2) \to pc_1' := inC$
      $[\!]\ pc_1 = inC \quad \to pc_1' := outC$

module P_2 **is**
  interface $pc_2 : \{outC, reqC, inC\};\ x_2 : \mathbb{B}$
  external $pc_1 : \{outC, reqC, inC\};\ x_1 : \mathbb{B}$
  lazy atom controls pc_2, x_2 **reads** pc_1, pc_2, x_1, x_2
    init
      $[\!]\ true \to pc_2' := outC;\ x_2' := \mathbb{B}$
    update
      $[\!]\ pc_2 = outC \quad \to pc_2' := reqC;\ x_2' := \neg x_1$
      $[\!]\ pc_2 = reqC \wedge (pc_1 = outC \vee x_1 = x_2) \to pc_2' := inC$
      $[\!]\ pc_2 = inC \quad \to pc_2' := outC$

module $Pete$ **is hide** x_1, x_2 **in** $P_1 \,\|\, P_2$

Figure 1.23: Asynchronous mutual exclusion

	initial	P_2	P_1, P_2			P_2	P_2	P_1, P_2	P_1	P_2
pc_1	*outC*	*outC*	*reqC*	*reqC*	*reqC*	*reqC*	*reqC*	*inC*	*outC*	*outC*
pc_2	*outC*	*reqC*	*inC*	*inC*	*inC*	*outC*	*reqC*	*reqC*	*reqC*	*inC*
x_1	*true*	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*
x_2	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*

Figure 1.24: An initialized trajectory of the module *Pete*

The previous, synchronous solution to the mutual-exclusion problem violates speed independence. An asynchronous solution, which must take the form of a speed-independent process set, is more difficult to devise and understand. The protocol *Pete* of Figure 1.23 provides the asynchronous solution due to Peterson, in which each process employs an additional boolean interface variable (x_1 and x_2, respectively). If both processes are trying to enter their critical sections in the same round, then the first process can succeed if $x_1 \neq x_2$, and the second process can succeed if $x_1 = x_2$. This guarantees the mutual-exclusion requirement. However, in contrast to the simple protocol *SyncMutex*, it is not obvious that *Pete* meets the accessibility requirement; indeed, much of this book will be devoted towards developing algorithms for checking if a finite module like *Pete* meets a requirement like accessibility. Figure 1.24 shows a sample initialized trajectory of *Pete*. Since the two atoms of *Pete* are speed-independent processes, in any given update round, either none, one, the other, or both atoms may sleep. The first line of Figure 1.24 indicates for every update round which processes proceed.

Exercise 1.9 {`T3`} [Accessibility for Peterson's protocol] Prove that the module *Pete* meets the accessibilty requirement. (As with the proofs required by other exercises, your aim need not be a derivation in some formal calculus, but an argument that is sufficiently rigorous and detailed as to convince the reader and, more importantly, yourself.) ■

Exercise 1.10 {`P3`} [Three-process mutual exclusion] You are asked to generalize Peterson's protocol to the case of three processes: first specify the three-process mutual-exclusion problem; then present your solution in the form of a finite module which is a speed-independent three-process set. Give an initialized trajectory of your protocol along which each process enters the critical section at least once (annotate the trajectory, as in Figure 1.24, with the processes that proceed during the update rounds). ■

Exercise 1.11 {`P3`} [Interleaving model] Peterson's protocol was originally designed under the interleaving assumption that in every update round at least one of the two processes sleeps. The interleaving assumption is stronger (more restrictive) than the assumption of speed independence, which permits update rounds in which both processes proceed. (a) Implement Peterson's original protocol using three synchronous, passive modules, two of which represent the two processes. The third module represents a scheduler which, in every update round, nondeterministically determines which of the two processes sleeps. The decision of the scheduler is communicated to the processes through an auxiliary variable that is controlled by the scheduler and awaited by the processes. After parallel composition, hide the auxiliary variable to obtain an asynchronous protocol which has the same abstract type as *Pete* and strictly fewer traces. Give a trace of *Pete* which is not a trace of your new protocol. (b) Every asynchronous protocol that meets the mutual-exclusion requirement under the assumption of speed independence also meets the mutual-exclusion requirement under the interleaving assumption. Can you find an asynchronous protocol, in the form of a speed-independent two-process set, which violates the mutual-exclusion requirement, but does so only along initialized trajectories that contain at least one update round in which both processes proceed? ■

1.3.3 Message-passing Protocols

We refer to distributed programs that communicate through messages as agents. The transmission of messages is governed by message-passing protocols, which ensure that all messages that are sent are also received. We illustrate the use of reactive modules for modeling distributed programs by modeling several protocols for passing messages between agents.

Event variables

We refer to every change in the value of a variable x as an x event. Thus, in every update round, an x event either happens or does not happen; the x events may happen as rarely as never, or as often as once per update round. In this spirit, in reactive modules we model the happening of a pure event — a happening without value, such as an individual clock tick or the fact that a message is being transmitted from a sender to a receiver— by toggling a boolean variable. Suppose, for example, that the boolean variable *tick* is used for modeling clock ticks. Then, the pure event "clock tick" happens whenever the value of *tick* changes either from *true* to *false*, or from *false* to *true*. In those update rounds in which the next value of *tick* is equal to the current value, no clock tick happens. In other words, the clock ticks are represented by the *tick* events.

If a boolean variable x is used to model pure events, then we are interested in all changes to the value of x, but the actual value of x at the beginning or end of any given round is irrelevant. Hence, Rml provides a special type, denoted $\mathbb{E}$, for the modeling of pure events. The variables of type $\mathbb{E}$ are called event variables. Each event variable ranges over the set $\mathbb{B}$ of boolean values, but compared to boolean variables, the initialization and updating of event variables is strongly restricted. The initialization of event variables is implicit: each event variable is initialized nondeterministically to either *true* or *false*. In update commands, an event variable x can occur only in the following two ways. First, the Rml expression $x!$ stands for the assignment $x' := \neg x$, which issues an x event. (If $x!$ is absent from a guarded assignment, then $x' := x$ by default, and no x event is issued.) It follows that the atom that controls x must read x. Second, the Rml expression $x?$ stands for the boolean expression $x' \neq x$, which checks if an x event is happening. It follows that an atom that does not control x, reads x if and only if it awaits x. Given a module P, we write $\mathsf{event}X_P$ for the set of event variables of P.

Synchronous communication

We are given two agents —a sender and a receiver. The sender produces a message, then sends the message to the receiver and produces another message, etc. The receiver, concurrently, waits to receive a message, then consumes the message and waits to receive another message, etc. We model each agent as a module that cannot observe the control variables of the other agent. The private variable *pc* of the sender indicates if the agent is producing a message ($pc = produce$) or attempting to send a message ($pc = send$). The sender starts by producing a message:

$$\begin{array}{l} \textbf{init} \\ \quad [\!]\ true \rightarrow pc' := produce \end{array}$$

Messages are produced by the lazy atom *Producer*, which requires an unknown number of rounds to produce a message. Once a message is produced, the producer issues a $done_P$ event (which is private to the sender) and the produced message is shown as msg_P (which initially is undefined). We assume that messages have the finite type $\mathbb{M}$, and that any stream of messages from the finite set $\mathbb{M}$ may be produced. We model these assumptions using nondeterminism:

$$\begin{array}{l} \textbf{lazy atom } Producer \textbf{ controls } done_P, msg_P \textbf{ reads } pc, done_P \\ \quad \textbf{init} \\ \quad\quad [\!]\ true \rightarrow msg'_P := \bot \\ \quad \textbf{update} \\ \quad\quad [\!]\ pc = produce \rightarrow done_P!;\ msg'_P := \mathbb{M} \end{array}$$

Once a message has been produced, the sender is ready to send the message:

$$\begin{array}{l} \textbf{update} \\ \quad [\!]\ pc = produce \wedge done_P? \rightarrow pc' := send \end{array}$$

The private variable *pc* of the receiver indicates if the agent is waiting to receive a message ($pc = receive$) or consuming a message ($pc = consume$). The receiver starts by waiting to receive a message:

$$\begin{array}{l}\textbf{init}\\ \quad [\!] \ \mathit{true} \to pc' := \mathit{receive}\end{array}$$

The received message is stored in the private variable msg_R. Messages are consumed by the lazy atom *Consumer*, which requires an unknown number of rounds to consume a message. Once a message is consumed, the consumer issues a $done_C$ event (which is private to the receiver) and the consumed message is shown as msg_C (which initially is undefined):

$$\begin{array}{l}\textbf{lazy atom } \mathit{Consumer} \textbf{ controls } done_C, msg_C \textbf{ reads } pc, done_C, msg_R\\ \quad \textbf{init}\\ \quad\quad [\!] \ \mathit{true} \to msg'_C := \bot\\ \quad \textbf{update}\\ \quad\quad [\!] \ pc = \mathit{consume} \to done_C!; \ msg'_C := msg_R\end{array}$$

Once a message has been consumed, the receiver waits to receive another message:

$$\begin{array}{l}\textbf{update}\\ \quad [\!] \ pc = \mathit{consume} \wedge done_C? \to pc' := \mathit{receive}\end{array}$$

Our task is to add guarded assignments that permit the sender to send a message, by updating *pc* from *send* to *produce*, and guarded assignments that permit the receiver to receive a message, by updating *pc* from *receive* to *consume*, in a controlled fashion. Roughly speaking, when composing both agents, the stream of consumed messages msg_C should contain the same message values, in the same order, as the stream of produced messages msg_P. Formal requirements for message-passing protocols will be stated in Chapter ??.

The protocol *SyncMsg* of Figure 1.25 has the sender and the receiver synchronize to transmit a message; that is, when ready to send a message, the sender is blocked until the receiver becomes ready to receive, and when ready to receive a message, the receiver is blocked until the sender transmits a message. The synchronization of both agents is achieved by two-way handshaking in three subrounds (or "stages") within a single update round. The first subround belongs to the atom *Stage1* of the receiver. If the receiver is ready to receive a message, it asynchronously issues an interface *ready* event to signal its readiness to the sender. The second subround belongs to the atom *Stage2* of the sender. If the sender sees an external *ready* event and is ready to send a message, it synchronously issues an interface *transmit* event to signal a transmission, and it offers the message which is to be transmitted in the interface variable msg_S. The third subround belongs to the atom *Stage3* of the receiver. If the receiver sees an external *transmit* event, it copies the message from the external variable msg_S to the private variable msg_R. The three-stage, two-way handshaking structure

module *SyncSender* **is**
  private $pc : \{produce, send\};\ \ done_P : \mathbb{E}$
  interface $transmit : \mathbb{E};\ \ msg_S : \mathbb{M};\ \ msg_P : \mathbb{M}_\perp$
  external $ready : \mathbb{E}$

  passive atom *Stage2*
    controls $pc, transmit, msg_S$
    reads $pc, done_P, ready, transmit, msg_P$
    awaits $done_P, ready$
    init
      $[\!]\ true \to pc' := produce;\ \ msg'_S := \mathbb{M}$
    update
      $[\!]\ pc = produce \wedge done_P? \to pc' := send$
      $[\!]\ pc = send \wedge ready? \to transmit!;\ \ msg'_S := msg_P;\ \ pc' := produce$

  lazy atom *Producer* **controls** $done_P, msg_P$ **reads** $pc, done_P$

module *Receiver* **is**
  private $pc : \{receive, consume\};\ \ msg_R : \mathbb{M};\ \ done_C : \mathbb{E}$
  interface $ready : \mathbb{E};\ \ msg_C : \mathbb{M}_\perp$
  external $transmit : \mathbb{E};\ \ msg_S : \mathbb{M}$

  passive atom *Stage3*
    controls pc, msg_R
    reads $pc, transmit, done_C$
    awaits $transmit, msg_S, done_C$
    init
      $[\!]\ true \to pc' := receive;\ \ msg'_R := \mathbb{M}$
    update
      $[\!]\ pc = receive \wedge transmit? \to msg'_R := msg'_S;\ \ pc' := consume$
      $[\!]\ pc = consume \wedge done_C? \to pc' := receive$

  lazy atom *Stage1* **controls** $ready$ **reads** $pc, ready$
    update
      $[\!]\ pc = receive \to ready!$

  lazy atom *Consumer* **controls** $done_C, msg_C$ **reads** $pc, done_C, msg_R$

module *SyncMsg* **is**
  —**interface** msg_P, msg_C
  hide $ready, transmit, msg_S$ **in**
    $\|$ *SyncSender*
    $\|$ *Receiver*

Figure 1.25: Synchronous message passing

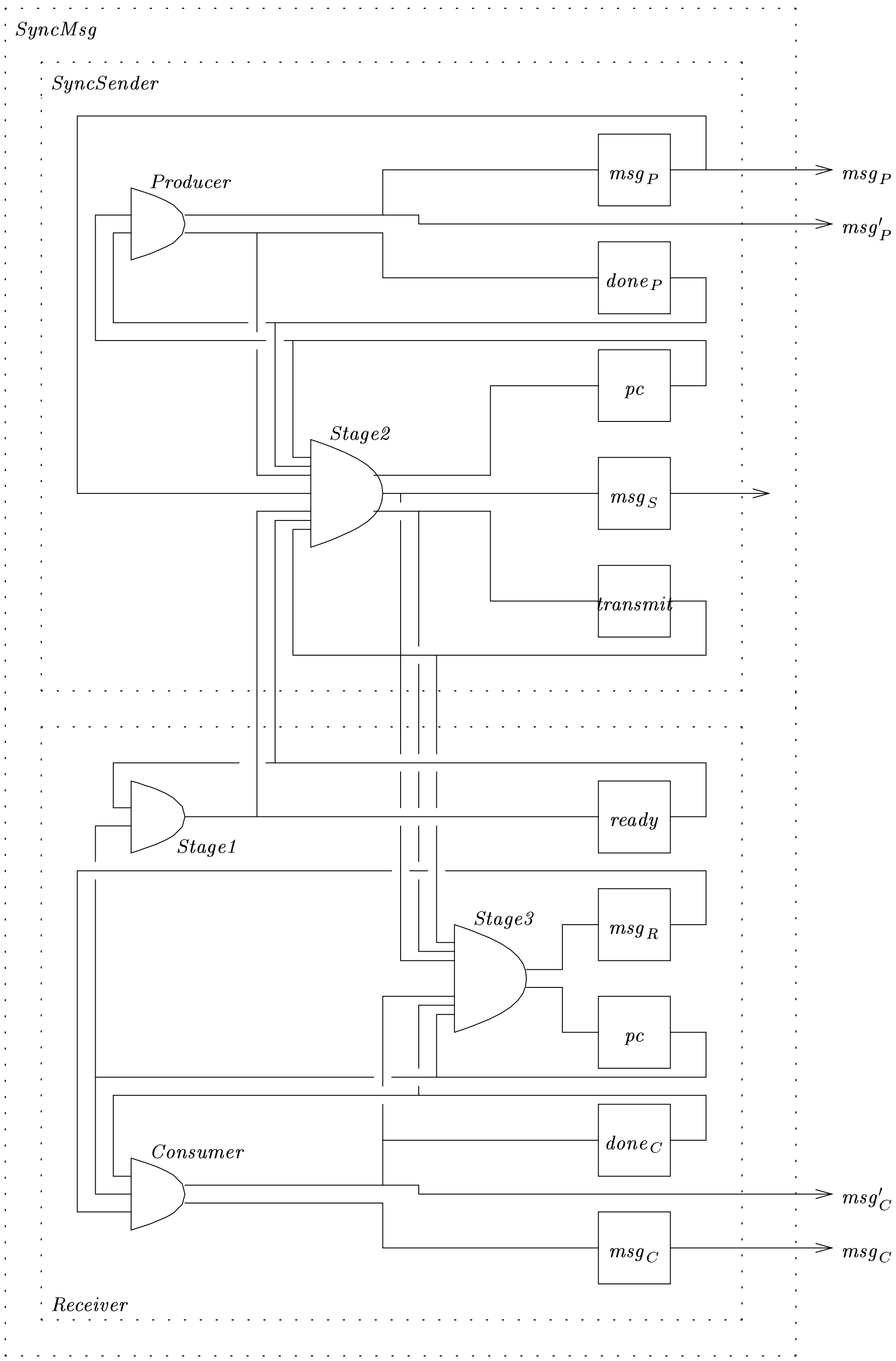

Figure 1.26: Block diagram for synchronous message passing

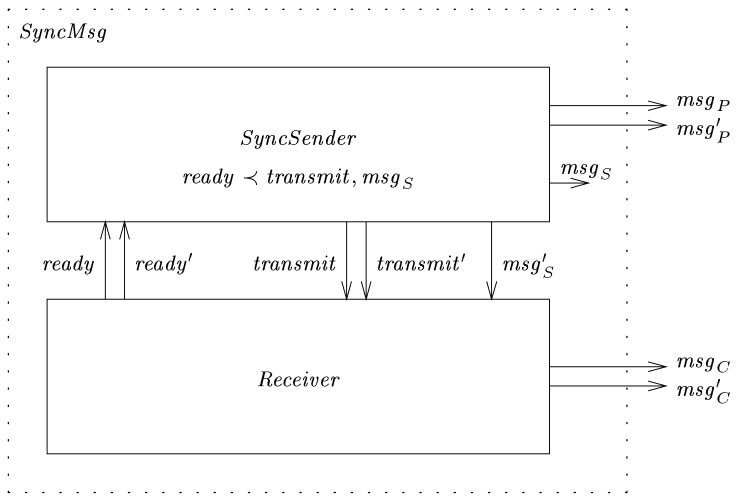

Figure 1.27: Abstract block diagram for synchronous message passing

of the protocol *SyncMsg* can be seen in the block diagram of Figure 1.26 and in the abstract block diagram of Figure 1.27.

Both agents *SyncSender* and *Receiver* are passive, because the sender may sleep in any given update round except when the receiver signals *ready*, and the receiver may sleep in any given update round except when the sender signals *transmit*. The sender is synchronous, because it signals *transmit* in the very round in which the receiver signals *ready*; the receiver is asynchronous. The entire protocol *SyncMsg*, after hiding the variables *transmit* and msg_S, is both asynchronous and passive. Figure 1.28 shows a sample initialized trajectory of the module *SyncMsg*, assuming that the possible values of messages are $\mathbb{M} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$. (The two program counters have been renamed implicitly, and their values are abbreviated. Instead of giving the values of event variables, the figure indicates when the corresponding events happen.) The corresponding trace of *SyncMsg* consists of a stream of produced messages msg_P and a stream of consumed messages msg_C.

Exercise 1.12 {P2} [Synchronous message passing] (a) Give a few additional initialized trajectories of the module *SyncMsg* and the corresponding traces. Then characterize, in precise words, the set of all traces of *SyncMsg*. (b) Define a module that has the same abstract type and the same traces as the module *SyncMsg*, but as few private variables as possible. ■

Exercise 1.13 {P3} [Dining philosophers] Suppose that there are two rooms and n philosophers. In one room, the philosophers think; in the other room, they eat

pc_S	p	p	s	s	p	s	s	s	p	p	p	s	s	p	s
$transmit$					◇				◇					◇	
msg_S	c	c	c	c	a	a	a	a	b	b	b	b	b	c	c
$done_P$			◇			◇						◇			◇
msg_P	$\perp$	$\perp$	a	a	a	b	b	b	b	b	b	c	c	c	b
pc_R	r	r	r	r	c	c	c	r	c	r	r	r	r	c	r
$ready$		◇			◇				◇		◇	◇		◇	
msg_R	b	b	b	b	a	a	a	a	b	b	b	b	b	c	c
$done_C$								◇		◇					◇
msg_C	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	a	a	b	b	b	b	b	c

msg_P	$\perp$	$\perp$	a	a	a	b	b	b	b	b	b	c	c	c	b
msg_C	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	a	a	b	b	b	b	b	c

Figure 1.28: An initialized trajectory of the module *SyncMsg* and the corresponding trace

while seated at a round table. Every philosopher owns one of the n chairs at the table. There is one chopstick between each of the n plates, and every philosopher uses both the left and the right chopstick for eating (it follows that at most $\lfloor n/2 \rfloor$ philosophers can be eating at the same time). Every philosopher begins by thinking and, when hungry, enters the dining room. There, the philosopher sits down at the table at the designated chair, picks up the chopstick to the left (or waits until it becomes available), and then the chopstick to the right (or waits). Once in control of both chopsticks, the philosopher eats, then releases both chopsticks, leaves the dining room, thinks, and returns when hungry again.

The passive module *Stick* of Figure 1.29 implements a chopstick. The private variable *pc* indicates if the chopstick is available ($pc = free$), picked up by the philosopher to the left ($pc = left$), or picked up by the philosopher to the right ($pc = right$). An external req_L event indicates that the philosopher to the left requests the chopstick, an interface $grant_L$ event indicates that the philosopher to the left picks up the chopstick, and an external $release_L$ event indicates that the philosopher to the left releases the chopstick. The event variables req_R, $grant_R$, and $release_R$ refer to the philosopher to the right. (a) Define a passive module *Phil* which implements a philosopher and, using multiple, renamed copies of *Phil* and *Stick*, define a compound module *Dine4* which implements the dining-philosophers scenario for $n = 4$. Illustrate the communication structure of the module *Dine4* by drawing block diagrams for *Phil*, *Stick*, and *Dine4* at suitable levels of abstraction. (b) Give an initialized trajectory of your module *Dine4* which ends up in a situation where all 4 philosophers sit at the table, have picked up one chopstick, and wait for the other chopstick to become avail-

$$
\begin{array}{l}
\textbf{module } \mathit{Stick} \textbf{ is} \\
\quad \textbf{private } pc\colon \{\mathit{free}, \mathit{left}, \mathit{right}\} \\
\quad \textbf{interface } \mathit{grant}_L, \mathit{grant}_R \colon \mathbb{E} \\
\quad \textbf{external } \mathit{req}_L, \mathit{release}_L, \mathit{req}_R, \mathit{release}_R \colon \mathbb{E} \\
\quad \textbf{passive atom} \\
\qquad \textbf{controls } pc, \mathit{grant}_L, \mathit{grant}_R \\
\qquad \textbf{reads } pc, \mathit{req}_L, \mathit{grant}_L, \mathit{release}_L, \mathit{req}_R, \mathit{grant}_R, \mathit{release}_R \\
\qquad \textbf{awaits } \mathit{req}_L, \mathit{release}_L, \mathit{req}_R, \mathit{release}_R \\
\qquad \textbf{init} \\
\qquad\quad [\!]\ \mathit{true} \to pc' := \mathit{free} \\
\qquad \textbf{update} \\
\qquad\quad [\!]\ pc = \mathit{free} \wedge \mathit{req}_L? \to \mathit{grant}_L!;\ pc' := \mathit{left} \\
\qquad\quad [\!]\ pc = \mathit{free} \wedge \mathit{req}_R? \to \mathit{grant}_R!;\ pc' := \mathit{right} \\
\qquad\quad [\!]\ pc = \mathit{left} \wedge \mathit{release}_L? \to pc' := \mathit{free} \\
\qquad\quad [\!]\ pc = \mathit{right} \wedge \mathit{release}_R? \to pc' := \mathit{free}
\end{array}
$$

Figure 1.29: A chopstick for the dining philosophers

able. There are several ways to prevent this deadlock situation. (b1) Have each philosopher pick up both chopsticks simultaneously (or wait until both chopsticks become available). (b2) Add to the entrance of the dining room a guard that admits at most $n-1=3$ philosophers into the dining room at any given time. Define a passive module *Guard* and draw the abstract block diagram for the dining-philosophers scenario with a guard. (Hide all communication between the philosophers and the guard so that the resulting module has the same abstract type as the module *Dine4*.) ■

Exercise 1.14 {P3} [Write-shared variables] Consider a concurrent program with two processes, R_1 and R_2, both of which have read and write access to a boolean variable x. When R_1 wishes to read the value of x, it issues an interface read_1 event and expects, depending on the current value of x, either an external *return_true* event or an external *return_false* event within the same round. Similarly, when R_2 wishes to read x, it issues an interface read_2 event and expects an external *return_true* or *return_false* event within the same round. When R_1 wishes to assign the value i to x, for $i \in \{\mathit{true}, \mathit{false}\}$, it issues an interface write_1_i event. Similarly, when R_2 wishes to assign the value i to x, it issues an interface write_2_i event. If both R_1 and R_2 issue conflicting write requests, then x is updated nondeterministically to one of the written values. (a) Define a passive module R_x for modeling the shared variable x. Your module should have the private boolean variable x, the two interface event variables *return_true* and *return_false*, and the external event variables read_1, read_2, $\mathit{write}_1\_\mathit{true}$, $\mathit{write}_1\_\mathit{false}$, $\mathit{write}_2\_\mathit{true}$, and $\mathit{write}_2\_\mathit{false}$. (b) Give an alternative implementation *Pete2* of Peterson's mutual-exclusion protocol (Figure 1.23) which uses instead of the two read-shared boolean variables x_1 and x_2 the single write-shared boolean variable x. Define two modules R_1 and R_2 for modeling the two processes of the protocol, encoding $x_1 = x_2$ by $x = \mathit{true}$ and $x_1 \neq x_2$ by $x = \mathit{false}$. The resulting module

module *Pete2* **is hide** ... **in** $R_1 \,\|\, R_2 \,\|\, R_x$

should have the same abstract type and the same traces as the module *Pete*. (c) How would you change the definition of R_x in part (a) if x is a nonnegative-integer variable rather than a boolean variable? ■

Asynchronous communication

While the synchronous message-passing protocol *SyncMsg* of Figure 1.26 performs a two-way handshake within a single round, the asynchronous protocol *AsyncMsg* of Figure 1.30 uses many rounds for a single handshake. The two protocols have identical receiver agents. Every send-receive cycle of *AsyncMsg* consists of four phases —a message production phase, an agent coordination phase, a message transmission phase, and a message consumption phase— each consisting of any number of update rounds. During the message production phase, the sender ($pc = \mathit{produce}$) takes an unknown number of rounds to produce a message. During the agent coordination phase, the sender ($pc = \mathit{wait}$) waits for an external *ready* event, which signals the readiness of the receiver to receive a message. The receiver ($pc = \mathit{receive}$) takes an unknown number of rounds to issue an interface *ready* event. During the message transmission phase, $pc = \mathit{send}$ for the sender and $pc = \mathit{receive}$ for the receiver. The sender takes an unknown number of rounds to transmit the message, asynchronously issuing an interface *transmit* event and simultaneously offering the message in the interface variable msg_S. The receiver, ready to receive, sees the external *transmit* event and copies the message from the external variable msg_S to the private variable msg_R. During the message consumption phase, the receiver ($pc = \mathit{consume}$) takes an unknown number of rounds to consume the message. The message consumption phase overlaps with the ensuing message production phase, which initiates a new send-receive cycle.

Exercise 1.15 {P2} [Asynchronous message passing] (a) Draw block diagrams for the module *AsyncMsg* at several levels of abstraction. (b) Give a few initialized trajectories of the module *AsyncMsg* and the corresponding traces. Then characterize, in precise words, the set of all traces of *AsyncMsg*. (c) How do the traces of the module *AsyncMsg* differ from the traces of the module *SyncMsg* from Exercise 1.12? ■

Exercise 1.16 {P4} [Faulty communication] Suppose that the delivery of messages may be delayed in a communication medium, and that messages may be

module *AsyncSender* **is**
  private $pc\colon \{produce, send, wait\}$; $done_P\colon \mathbb{E}$
  interface $transmit\colon \mathbb{E}$; $msg_S\colon \mathbb{M}$; $msg_P\colon \mathbb{M}_\perp$
  external $ready\colon \mathbb{E}$

  passive atom *ProgramCounter*
    controls pc
    reads $pc, done_P, ready, transmit$
    awaits $done_P, ready, transmit$
    init
      $[\!]\ true \to pc' := produce$
    update
      $[\!]\ pc = produce \wedge done_P? \to pc' := wait$
      $[\!]\ pc = wait \wedge ready? \to pc' := send$
      $[\!]\ pc = send \wedge transmit? \to pc' := produce$

  lazy atom *Transmitter*
    controls $transmit, msg_S$
    reads $pc, transmit, msg_P$
    init
      $[\!]\ true \to msg'_S := \mathbb{M}$
    update
      $[\!]\ pc = send \to transmit!;\ msg'_S := msg_P$

  lazy atom *Producer* **controls** $done_P, msg_P$ **reads** $pc, done_P$

module *AsyncMsg* **is**
  —**interface** msg_P, msg_C
  hide $ready, transmit, msg_S$ **in**
    $\|$ *AsyncSender*
    $\|$ *Receiver*

Figure 1.30: Asynchronous message passing

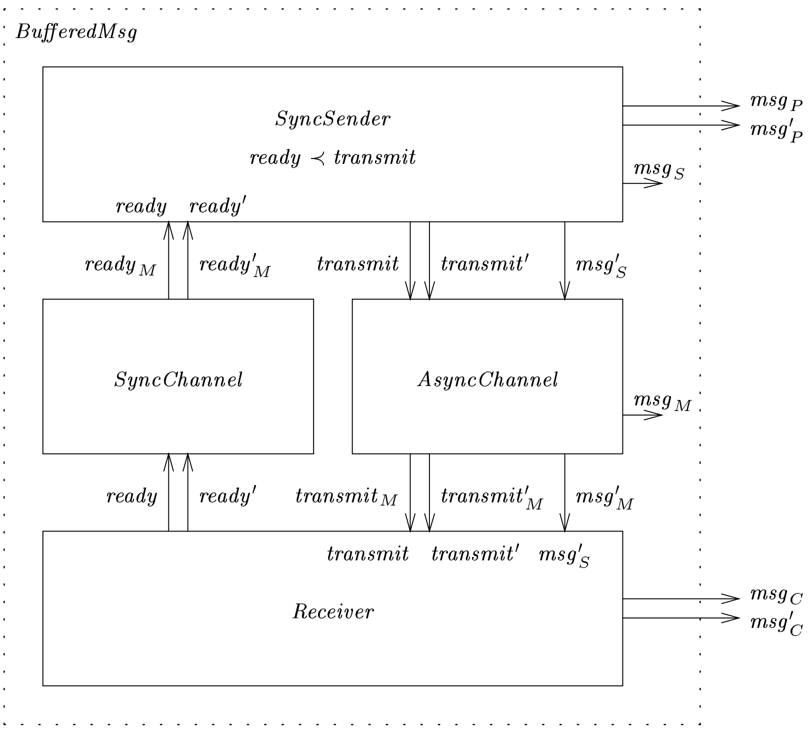

Figure 1.31: Message passing through channels

reordered, lost, and corrupted in the medium. We model the communication medium by two agents that are interjected between the sender and the receiver of the synchronous message-passing protocol *SyncMsg*. The signal channel *SyncChannel*, with the external variable *ready* and the interface variable $ready_M$, delivers signals from the receiver to the sender. The message channel *AsyncChannel*, with the external variables *transmit* and msg_S and the interface variables $transmit_M$ and msg_M, delivers messages from the sender to the receiver. The new message-passing protocol is implemented by the module

module *BufferedMsg* **is**
  —**interface** msg_P, msg_C
  hide $ready, ready_M, transmit, transmit_M, msg_S, msg_M$ **in**
    $\|$ $SyncSender[ready := ready_M]$
    $\|$ *SyncChannel*
    $\|$ *AsyncChannel*
    $\|$ $Receiver[transmit, msg_S := transmit_M, msg_M]$

whose abstract block diagram is shown in Figure 1.31. We wish to model both reliable and unreliable communication media with various latencies and capacities. For each of the following parts of the exercise, give at least one initialized trajectory of the module *BufferedMsg*. (a) Define a synchronous, passive module *SyncChannel* which delivers signals without delay (every signal is delivered in the round in which it is is sent). Then define an asynchronous, passive module *AsyncChannel* which delays messages arbitrarily (by an unknown number of rounds), and delivers the messages in the order in which they are sent. (b) Modify the module *AsyncChannel* so that the messages are not necessarily delivered in the order in which they are sent. (b1) Assume that a message can be overtaken by at most 3 newer messages. (b2) Assume that a message can be overtaken by an arbitrary number of newer messages. (c) Modify the module *AsyncChannel* so that messages may be lost. (c1) Assume that any message may be lost. (c2) Assume that the message channel has a capacity of 5 messages; that is, in any given round, the channel can store at most 5 undelivered messages. If 5 messages are stored and a new message is received from the sender, then the new message is lost. (d) Modify the module *AsyncChannel* so that messages may be reordered, lost, and corrupted (a message is corrupted if its contents is changed arbitrarily). ■

Timed communication

Instead of waiting for the receiver to be ready to receive a message, the sender may choose to retransmit a message repeatedly until the receiver acknowledges the receipt of the message. The decision to retransmit can be based upon timing: a message is retransmitted if no acknowledgment is obtained within a certain amount of time. For measuring time, we let protocols refer to an external digital clock. We model the clock as an asynchronous, passive module:

> **module** *AsyncClock* **is**
>   **interface** $tick : \mathbb{E}$
>   **lazy atom controls** *tick* **reads** *tick*
>   **update**
> $[\!]\ true \rightarrow tick!$

The clock module *AsyncClock* issues *tick* events at undetermined times. The *tick* events represent clock ticks and can be observed by other modules.

Consider the receiver module *TimedReceiver* of Figure 1.32. Instead of signaling when it is ready to receive a message, the module *TimedReceiver* confirms the receipt of a message by issuing an interface *ack* event for acknowledgment. If for the duration of 4 clock ticks after receiving a message, the message has not been acknowledged, then a time-out occurs. If a time-out occurs, then an acknowledgment is issued right away, in the round that immediately follows the 4th clock tick after message reception. Consequently, the receipt of every message is confirmed within at most 4 time units, as measured by the clock module

module *TimedReceiver* **is**
  private $pc\colon \{receive, confirm, consume\}$; $msg_R\colon \mathbb{M}$;
    $done_C, timeout\colon \mathbb{E}$; $timer\colon [0..3]$
  interface $ack\colon \mathbb{E}$; $msg_C\colon \mathbb{M}_\perp$
  external $transmit, tick\colon \mathbb{E}$; $msg_S\colon \mathbb{M}$

  passive atom *ReceiverProgramCounter*
    controls pc, msg_R, ack
    reads $pc, transmit, timeout, ack, done_C$
    awaits $transmit, msg_S, timeout, done_C$
    init
      $[\!] \; true \to pc' := receive;\; msg'_R := \mathbb{M}$
    update
      $[\!] \; pc = receive \wedge transmit? \quad \to msg'_R := msg'_S;\; pc' := confirm$
      $[\!] \; pc = confirm \wedge \neg timeout? \to$
      $[\!] \; pc = confirm \quad \to ack!;\; pc' := consume$
      $[\!] \; pc = consume \wedge done_C? \quad \to pc' := receive$

  passive atom *ReceiverTimer*
    controls $timer, timeout$
    reads $pc, transmit, tick, timer, timeout$
    awaits $transmit, tick$
    init
      $[\!] \; true \to timer := [0..3]$
    update
      $[\!] \; pc = receive \wedge transmit? \quad \to timer' := 0$
      $[\!] \; pc = confirm \wedge tick? \wedge timer < 3 \to timer' := timer + 1$
      $[\!] \; pc = confirm \wedge tick? \wedge timer = 3 \to timeout!$

  lazy atom *Consumer* **controls** $done_C, msg_C$ **reads** $pc, done_C, msg_R$

Figure 1.32: Timed message passing: receiver

module *TimedSender* **is**
  private $pc\colon \{produce, send, wait\};\ \ done_P, timeout\colon \mathbb{E};\ \ timer\colon [0..4]$
  interface $transmit\colon \mathbb{E};\ \ msg_S\colon \mathbb{M};\ \ msg_P\colon \mathbb{M}_\perp$
  external $ack, tick\colon \mathbb{E}$

  passive atom *SenderProgramCounter*
    controls pc
    reads $pc, done_P, transmit, ack, timeout$
    awaits $done_P, transmit, ack, timeout$
    init
      $[\!] \ true \to pc' := produce$
    update
      $[\!] \ pc = produce \wedge done_P? \to pc' := send$
      $[\!] \ pc = send \wedge transmit? \to pc' := wait$
      $[\!] \ pc = wait \wedge ack? \to pc' := produce$
      $[\!] \ pc = wait \wedge timeout? \to pc' := send$

  passive atom *SenderTimer*
    controls $timer, timeout$
    reads $pc, transmit, tick, timer, timeout$
    awaits $transmit, tick$
    init
      $[\!] \ true \to timer := [0..4]$
    update
      $[\!] \ pc = send \wedge transmit? \to timer' := 0$
      $[\!] \ pc = wait \wedge tick? \wedge timer < 4 \to timer' := timer + 1$
      $[\!] \ pc = wait \wedge tick? \wedge timer = 4 \to timeout!$

  lazy atom *Transmitter*
    controls $transmit, msg_S$
    reads $pc, transmit, msg_P$

  lazy atom *Producer* **controls** $done_P, msg_P$ **reads** $pc, done_P$

Figure 1.33: Timed message passing: sender

AsyncClock. The corresponding sender does not know when the receiver is ready to receive a message. If a message is transmitted when the receiver is not ready to receive it, then the message is lost. Therefore the sender must retransmit every message that is not acknowledged. The sender module *TimedSender* of Figure 1.33 uses the atom *Transmitter* from Figure 1.30 for transmitting a message, and then waits for an acknowledgment for exactly 5 clock ticks. The waiting period of 5 clock ticks suffices, because the receiver acknowledges the receipt of every message within at most 4 clock ticks. If the sender does not obtain an acknowledgment for the duration of 5 clock ticks, then it decides to retransmit the message. The resulting timed message-passing protocol

module *TimedMsg* **is**
  —**interface** msg_P, msg_C
  hide $transmit, msg_S, ack, tick$ **in**
    ‖ *TimedSender*
    ‖ *TimedReceiver*
    ‖ *AsyncClock*

has the same abstract type as the protocols *SyncMsg* and *AsyncMsg*.

Exercise 1.17 {P2} [Timed message passing] (a) Draw block diagrams for the module *TimedMsg* at several levels of abstraction. (b) Give a few initialized trajectories of the module *TimedMsg* and the corresponding traces. Then characterize, in precise words, the set of all traces of *TimedMsg*. (c) How do the traces of the module *TimedMsg* differ from the traces of the module *AsyncMsg* from Exercise 1.15? ■

Exercise 1.18 {P2} [More timed message passing] If the worst-case durations of message transmission and message consumption are known to the sender, then there is no need for the receiver to signal its readiness to receive a message, nor to acknowledge the receipt of a message. Design a timed message-passing protocol that consists of a sender, a message channel, a receiver, and the clock module AsyncClock, and reflects the following three timing assumptions: (1) the production of a message requires at least 3 clock ticks, and after transmitting a message, the sender waits for 4 clock ticks before producing another message; (2) the channel takes at least 2 and at most 5 clock ticks to deliver a message; (3) every message is consumed within a single clock tick. Since $5 + 1 < 3 + 4$, the receiver can be ready to receive every message that is transmitted by the sender. The resulting message-passing protocol should have the same abstract type and the same traces as the module *AsyncMsg*. ■

1.3.4 Asynchronous Circuits*

In an asynchronous circuit, unlike 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 may be delayed. We model each asynchronous logic gate by a nondeterministic passive module for which a change in the values of its inputs causes a corresponding change in the value of the output after an arbitrary number of update rounds. An asynchronous logic gate is stable when its output is the desired function of the inputs, and unstable otherwise. For example, an asynchronous And gate is stable when its output is the conjunction of both inputs. The condition

$$\text{And}(in_1, in_2, out) \;=\; (out = in_1 \cdot in_2)$$

is called the stability condition of an And gate with inputs in_1 and in_2 and output out. The output of an asynchronous gate can change only if the gate is unstable; when this happens the gate becomes stable. The gate takes an unknown number of rounds to become stable. If the gate is stable, and any of the inputs change in a way that violates the stability condition, then the gate turns unstable. If the gate is unstable, and any of the inputs change without rendering the stability condition true, the gate remains unstable. If, however, any of the inputs of an unstable gate change in a way that renders the stability condition true, a hazard is encountered, and the gate fails. If a gate has failed, its output may change arbitrarily. These modeling assumptions for an asynchronous And gate are specified by the asynchronous, passive module *AsyncAnd* of Figure 1.34. The private variable *pc* indicates the status of the gate (*stable*, *unstable*, or *hazard*) at the end of each round. The interface and external variables of *AsyncAnd* are identical to the interface and external variables of the synchronous module *SyncAnd* from Figure 1.15. However, unlike *SyncAnd*, the asynchronous module *AsyncAnd* has no (derived) await dependencies.

Exercise 1.19 {P3} [Asynchronous circuits] (a) Define an asynchronous, passive module *AsyncNot* which specifies an asynchronous Not gate (use the variable name *in* for input, and use *out* for output). Give a few initialized trajectories of the module *AsyncNot*. Then characterize, in precise words, the set of all traces of *AsyncNot*. (b) Give a few initialized trajectories of the module

module *AsyncNor* **is**
  hide z_1, z_2 **in**
    $\|$ $AsyncAnd[in_1, in_2, out := z_1, z_2, out]$
    $\|$ $AsyncNot[in, out := in_1, z_1]$
    $\|$ $AsyncNot[in, out := in_2, z_2]$

and characterize its traces. Given our modeling assumptions, is the module *AsyncNor* a correct implementation of an asynchronous Nor gate? (How do the traces of *AsyncNor* compare with the traces of the module that results from replacing each And condition in the module *AsyncAnd* by a Nor condition?) (c) An asynchronous latch has the two external variables *set* and *reset* and the interface variable *out*. The state of the asynchronous latch is stable when $set = 1$ implies that the state is 1, and $reset = 1$ implies that the state is 0. The

$$
\begin{array}{l}
\textbf{module } \mathit{AsyncAnd} \textbf{ is} \\
\quad \textbf{private } \mathit{pc} \colon \{\mathit{stable}, \mathit{unstable}, \mathit{hazard}\} \\
\quad \textbf{interface } \mathit{out} \colon \mathbb{B} \\
\quad \textbf{external } \mathit{in}_1, \mathit{in}_2 \colon \mathbb{B} \\
\quad \textbf{lazy atom } \mathit{Output} \textbf{ controls } \mathit{out} \textbf{ reads } \mathit{pc}, \mathit{out} \\
\quad\quad \textbf{init} \\
\quad\quad\quad [\!]\ \mathit{true} \to \mathit{out}' := \mathbb{B} \\
\quad\quad \textbf{update} \\
\quad\quad\quad [\!]\ \mathit{pc} = \mathit{unstable} \to \mathit{out}' := \neg \mathit{out} \\
\quad\quad\quad [\!]\ \mathit{pc} = \mathit{hazard} \quad \to \mathit{out}' := \neg \mathit{out} \\
\quad \textbf{passive atom } \mathit{Status} \textbf{ controls } \mathit{pc} \textbf{ reads } \mathit{pc}, \mathit{out} \textbf{ awaits } \mathit{in}_1, \mathit{in}_2, \mathit{out} \\
\quad\quad \textbf{init} \\
\quad\quad\quad [\!]\ \mathrm{And}(\mathit{in}_1', \mathit{in}_2', \mathit{out}') \quad \to \mathit{pc}' := \mathit{stable} \\
\quad\quad\quad [\!]\ \neg\mathrm{And}(\mathit{in}_1', \mathit{in}_2', \mathit{out}') \to \mathit{pc}' := \mathit{unstable} \\
\quad\quad \textbf{update} \\
\quad\quad\quad [\!]\ \mathit{pc} = \mathit{stable} \wedge \neg\mathrm{And}(\mathit{in}_1', \mathit{in}_2', \mathit{out}') \qquad\qquad\qquad\quad \to \mathit{pc}' := \mathit{unstable} \\
\quad\quad\quad [\!]\ \mathit{pc} = \mathit{unstable} \wedge \mathrm{And}(\mathit{in}_1', \mathit{in}_2', \mathit{out}') \wedge \mathit{out}' \neq \mathit{out} \to \mathit{pc}' := \mathit{stable} \\
\quad\quad\quad [\!]\ \mathit{pc} = \mathit{unstable} \wedge \mathrm{And}(\mathit{in}_1', \mathit{in}_2', \mathit{out}') \wedge \mathit{out}' = \mathit{out} \to \mathit{pc}' := \mathit{hazard}
\end{array}
$$

Figure 1.34: Asynchronous And gate

output of the asynchronous latch is stable when *out* is equal to the state of the latch. The state and the output stabilize independently, each taking an unknown number of rounds to switch from unstable to stable. A hazard is encountered if either the state is unstable and a change of the inputs renders it stable, or the output is unstable and a change of the state renders it stable. Define an asynchronous, passive module *AsyncLatch* which specifies an asynchronous latch under these modeling assumptions. Give a few initialized trajectories of the module *AsyncLatch* and characterize its traces. (d) Give a few initialized trajectories of the module

$$
\begin{array}{l}
\textbf{module } \mathit{AsyncLatch2} \textbf{ is} \\
\quad \textbf{hide } z \textbf{ in} \\
\qquad \| \ \mathit{AsyncNor}[in_1, in_2, out := set, z, out] \\
\qquad \| \ \mathit{AsyncNor}[in_1, in_2, out := reset, out, z]
\end{array}
$$

and characterize its traces. Given our modeling assumptions, is the module *AsyncLatch2* a correct implementation of an asynchronous latch? (How do the traces of *AsyncLatch2* compare with the traces of *AsyncLatch*?) ■

Exercise 1.20 {P3} [Explicitly clocked circuits] (a) Modify the modules *AsyncAnd* and *AsyncNot* (from Exercise 1.19) so that each gate, when unstable, stabilizes within at most 3 rounds, provided no hazard is encountered in the meantime. Are the resulting modules *ClockedAnd* and *ClockedNot* synchronous or asynchronous? Active or passive? (b) Modify the module *AsyncLatch* (from Exercise 1.19) so that the state of the latch, when unstable, stabilizes within at most 3 rounds, provided no hazard is encountered in the meantime. Furthermore, the output of the latch, when unstable, stabilizes whenever an external *tick* event occurs. The resulting module *ClockedLatch* should be synchronous and passive. Unlike the synchronous, active latch *SyncLatch* of Figure 1.17, which is implicitly clocked (every update round corresponds to a clock cycle), the synchronous, passive latch *ClockedLatch* is explicitly clocked (every external *tick* event corresponds to a clock cycle). (c) Let *Clocked3BitCounter* be the module that results from replacing every component of the module *Sync3BitCounter* from Example 1.20 as follows: replace each occurrence of *SyncAnd* by *ClockedAnd*, each occurrence of *SyncNot* by *ClockedNot*, and each occurrence of *SyncLatch* by *ClockedLatch*. Define a module *Clock*, which issues interface *tick* events, so that the compound module

$$\textbf{hide } \mathit{tick} \textbf{ in } \mathit{Clocked3BitCounter} \,\|\, \mathit{Clock}$$

implements an asynchronous three-bit counter whose only hazards can be caused by primary inputs (*start* and *inc*) changing too frequently. (You need to determine the minimal frequency of clock ticks which cannot cause hazards.) ■

1.4 Bibliographic Remarks

Reactive modules were introduced by [AlurHenzinger99]. Rml is at its core a synchronous modeling language based on read-shared variables, and thus is closely related to synchronous programming languages such as Esterel by [BerryGonthier88]. In Rml, asynchrony is modeled by nondeterministic progress, and communication events are modeled by changes in the values of variables. Paradigmatic modeling languages that are based on these alternative primitives include the asynchronous shared-variables language Unity by [ChandyMisra88], the asynchronous event-communication language I/O Automata by [Lynch96], and the synchronous event-communication languages Csp and Ccs by [Hoare85, Milner89].

Bibliography

[AlurHenzinger99] R. Alur and T.A. Henzinger, Reactive modules, Formal Methods in System Design 15, 1999, pp. 7–48.

[BerryGonthier88] G. Berry and G. Gonthier, The Synchronous Programming Language Esterel: Design, Semantics, Implementation, Technical Report 842, INRIA, 1988.

[ChandyMisra88] K.M. Chandy and J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, 1988.

[Hoare85] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[Lynch96] N.A. Lynch, Distributed Computing, Morgan-Kaufmann, 1996.

[Milner89] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.

Index