2

Synchronous Model

A *functional* component produces outputs when supplied with inputs, and its behavior can be mathematically described using a mapping between input and output values. A *reactive* component, in contrast, maintains an internal state and interacts with other components via inputs and outputs in an ongoing manner. We first focus on a *discrete* and *synchronous* model of reactive computation in which all components execute in a sequence of rounds. In each round, a reactive component reads its inputs; based on its current state and inputs, it computes outputs and updates the internal state.

2.1 Reactive Components

As a first example, consider the `Delay` component shown in figure 2.1. The component has a Boolean input variable *in*, a Boolean output variable *out*, and an internal state modeled by a Boolean variable x. To describe the behavior of the component, we first need to describe the initial values for the state variables. For `Delay`, assume that the initial value of x is 0. In each round of execution, the component sets the output variable *out* to the value of the state variable x at the beginning of the round and then updates the state to the value of the input variable in the current round. Thus, in the first round, the output will be 0, and in each subsequent round, the output will be equal to the input in the previous round.

2.1.1 Variables, Valuations, and Expressions

To explain the various aspects of the definition of a component precisely, we need a bit of mathematical notation concerning variables, expressions over variables, and assignments of values to variables. We use *typed variables* to describe components. The commonly used types are:

- `nat` denoting the set of natural numbers.

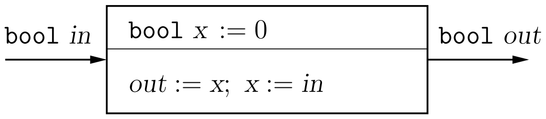


Figure 2.1: Reactive Component `Delay`

- `int` denoting the set of integers.
- `real` denoting the set of real numbers.
- `bool` denoting the set of Boolean values $\{0, 1\}$.
- An enumerated type contains a finite number of symbolic constants; an example of such a type is the set $\{on, off\}$ with two values.

Given a set V of typed variables, a *valuation* over V is a type-consistent assignment to all the variables in V. That is, a valuation over V is a function q with domain V such that for each variable $v \in V$, $q(v)$ is a value belonging to the type of v. We use Q_V to denote the set of all valuations over V. For example, if V contains two variables, the variable x of type `bool` and variable y of type `nat`, then a valuation q assigns a Boolean value to x and a natural number to y, and the set Q_V contains all such possible valuations.

A *typed expression* e over a set V of typed variables is constructed using variables in V, constants, and primitive operations over types corresponding to these variables. Over numerical types, such as `nat`, `int`, and `real`, we will use arithmetic operations such as addition and multiplication and comparison operations such as $=$ and $\leq$. To construct Boolean expressions, we use the following logical operators:

- *Negation* ($\neg$): the expression $\neg\, e$ evaluates to 1 precisely when e evaluates to 0;
- *Conjunction* ($\wedge$): the expression $e_1 \wedge e_2$ evaluates to 1 precisely when both e_1 and e_2 evaluate to 1; and
- *Disjunction* ($\vee$): the expression $e_1 \vee e_2$ evaluates to 1 precisely when at least one of e_1 or e_2 evaluates to 1.
- *Implication* ($\rightarrow$): the expression $e_1 \rightarrow e_2$ evaluates to 1 precisely when either e_1 evaluates to 0 or e_2 evaluates to 1.

2.1.2 Inputs, Outputs, and States

The component `Delay` of figure 2.1 has one input variable, one output variable, and one state variable. In general, a component C has a set I of typed input variables, a set O of typed output variables, and a set S of typed state variables.

All three sets should be *finite*. To avoid conflicts in variable names, these sets should also be *disjoint* from one another.

For the `Delay` component, $I = \{in\}$, $O = \{out\}$, and $S = \{x\}$.

In our illustrations, we draw components as rectangular boxes. For each input variable, there is an incoming arrow incident upon this box, and for each output variable, there is an outgoing arrow. These arrows are labeled with the names and types of the corresponding variables. The state variables are listed inside the component box.

An *input* to a reactive component C is a valuation over the set I of its input variables, and the set of all possible inputs is Q_I. An *output* of a component C is a valuation over the set O of its output variables, and the set of all possible outputs is Q_O. A *state* of a component C is a valuation over the set S of its state variables, and the set of its states is Q_S.

For the `Delay` component, an input is a Boolean value for the variable *in*, an output is a Boolean value for the variable *out*, and a state is a Boolean value for the variable *x*. Thus, each of the sets Q_I, Q_O, and Q_S contains two elements.

2.1.3 Initialization

To describe the dynamics of the component, we must specify the initial states and how the component reacts to a given input in each state. A variety of programming styles are used to describe this, ranging from imperative style (for instance, SystemC and Esterel), declarative equational style (for instance, Lustre), and hierarchical state machines (for instance, Stateflow). To be analyzable by tools, the precise *syntax*—what are the legal code fragments for describing the initialization and update, and the precise *semantics*—what are the corresponding mathematical sets of initial states and reactions, needs to be formalized. This can be challenging for real-world languages and even for "toy" languages, defining semantics formally requires a potentially overwhelming level of mathematical notation. We will use a combination of common imperative constructs and state machines, introducing the features as needed, without rigorously formalizing the mathematical semantics.

The *initialization* of a component, denoted *Init*, specifies the initial values for all the state variables in S. Whenever a state variable is declared, the corresponding initial values are described using an assignment. For example, in the `Delay` component, the state variable x is initialized using the assignment $x := 0$. Sometimes we want to specify multiple possible initial values to allow modeling of situations where initial conditions are only partially known. For this purpose, we will use a new construct called `choose`, which returns an arbitrarily chosen value from its argument set. For the `Delay` component, consider an alternative declaration for the variable x given by

$$\texttt{bool}\ x\ := \texttt{choose}\ \{0, 1\}.$$

In this modified version, `choose` may return either 0 or 1; as a result, the initial value of the variable x may be either 0 or 1. Another example of initialization using the `choose` construct is the declaration

$$\texttt{real}\ x := \texttt{choose}\ \{z \mid 0 \leq z \leq 2\}.$$

This means that the variable x is real-valued, and its initial value can be any real number between 0 and 2.

A state q of the component is called an *initial state* if, for every state variable x, the value $q(x)$ is consistent with the initialization of the variable x. The set of all initial states is denoted $[\![\mathit{Init}]\!]$. Thus, the initialization *Init* is a syntactic description of how the component initializes state variables, and the corresponding set $[\![\mathit{Init}]\!]$ is its mathematical semantics.

For the component `Delay`, the set $[\![\mathit{Init}]\!]$ contains the single initial state that assigns the value 0 to x.

In our illustrations of components, we split the box representing the component by a horizontal line, and the top part lists all the state variables along with their types, followed by their initialization.

2.1.4 Update

The computation of a component in response to an input in each round is given by its *reaction description*, denoted *React*. If the component in state s, when supplied with input i, can produce output o and update its state to t, we write $s \xrightarrow{i/o} t$. Such a response is called a *reaction*.

A natural way to describe the reactions is using code that assigns values to the output variables and updates the values of the state variables. This code can use the values of the input variables and the state variables at the beginning of the round. The set of all possible reactions of the component is the semantics of the reaction description and is denoted $[\![\mathit{React}]\!]$.

For the `Delay` component, the reaction description *React* is a sequence of two assignment statements:

$$\mathit{out} := x;\ \ x := \mathit{in}.$$

That is, in a state s, given an input i, the component copies the state to the output and updates the state to the current input. In this case, the component has four possible reactions:

$$0 \xrightarrow{0/0} 0;\quad 0 \xrightarrow{1/0} 1;\quad 1 \xrightarrow{0/1} 0;\quad 1 \xrightarrow{1/1} 1.$$

In our illustrations, the reaction description is given in the lower half of the component box.

The reaction description typically is a sequence of statements, where each statement is either an assignment statement or a conditional statement. An assignment statement is of the form $x := e$, where e is an expression of the same type as the type of the variable x. We allow the right-hand side of an assignment to use the `choose` construct to specify a set of possible values, thereby permitting multiple responses to the same input in a given state. A conditional statement is of the form

$$\texttt{if } b \texttt{ then } \mathit{stmt}_1 \texttt{ else } \mathit{stmt}_2$$

where b is a Boolean expression, and stmt_1 and stmt_2 are code fragments given as sequences of assignment and conditional statements. To execute the conditional statement, the expression b is evaluated first. If it evaluates to 1, the code for stmt_1 is executed; otherwise the code for stmt_2 is executed. We will use curly braces { and } to group statements together when needed. It is possible that the `else`-branch of a conditional statement is missing.

Given a state s and an input i, to find the possible reactions of the component, we execute the reaction description code. If we can execute the code without errors, and if the values assigned to all the output variables give the output valuation o, while the updated values of all the state variables give the state valuation t, then this contributes the reaction $s \xrightarrow{i/o} t$ to the set $[\![\mathit{React}]\!]$. Such a successful execution may not be possible for different reasons. Two common cases are discussed below.

First, when the code tries to execute an assignment statement $x := e$, we expect x to be either an output variable or a state variable; an attempt to assign a value to an input or an undeclared variable is an error. To be able to evaluate the expression e, it should refer only to state variables, input variables, and those output variables assigned values by previously executed assignment statements. In the description of the `Delay` component, if we replace the reaction description by

$$x := \mathit{out};\ \ \mathit{out} := \mathit{in}$$

then the first statement cannot be executed as no value of the output variable out is known, and for this description, the corresponding set of reactions is the empty set.

Second, if the code does not assign values to all the output variables, then this execution cannot define a valid reaction. For example, in the `Delay` component, if we replace the reaction description by the conditional assignment

$$\texttt{if } (x \neq \mathit{in}) \texttt{ then } \mathit{out} := x$$

then the statement updates the output only when the input differs from the current state. The modified component then has only two reactions: $0 \xrightarrow{1/0} 1$ and $1 \xrightarrow{0/1} 0$. This can be interpreted as a specification of a component that rejects the input 0 in state 0 and rejects the input 1 in state 1.

The reaction description can also use *local* variables, that is, auxiliary variables used to store results of intermediate computation. Suppose given integer values of input variable in_1 and in_2, we want to compute the output variable *out* with value equal to the difference of the squares of the two input values. The code below achieves this using only one multiplication:

```
local int $x$, $y$;
   $x := in_1 + in_2$;
   $y := in_1 - in_2$;
   $out := x * y$.
```

In this description, the values of the variables x and y are not available to the other components and are not stored across rounds.

We summarize the discussion so far by presenting a formal definition of a reactive component below:

Synchronous Reactive Component

A *synchronous reactive component* C is described by

- a finite set I of typed input variables defining the set Q_I of inputs,
- a finite set O of typed output variables defining the set Q_O of outputs,
- a finite set S of typed state variables defining the set Q_S of states,
- an initialization *Init* defining the set $[\![Init]\!] \subseteq Q_S$ of initial states, and
- a reaction description *React* defining the set $[\![React]\!]$ of reactions of the form $s \xrightarrow{i/o} t$, where s, t are states, i is an input, and o is an output.

2.1.5 Executions

The operational semantics of a component can be captured by defining its executions. To execute a component, we first initialize all the variables to obtain an initial state. The execution then proceeds for a finite number of rounds. In each round, values for input variables are chosen. Then the code in the reaction description of the component is executed to determine its output and updated state.

Formally, an *execution* of a synchronous reactive component C of length k, where $k \geq 0$, consists of a finite sequence of the form

$$s_0 \xrightarrow{i_1/o_1} s_1 \xrightarrow{i_2/o_2} s_2 \xrightarrow{i_3/o_3} s_3 \quad \cdots\cdots \quad s_{k-1} \xrightarrow{i_k/o_k} s_k,$$

where

1. for $0 \leq j \leq k$, each s_j is a state of C, and for $1 \leq j \leq k$, each i_j is an input of C, and each o_j is an output of C;

2. s_0 is an initial state of C; and

3. for $1 \leq j \leq k$, $s_{j-1} \xrightarrow{i_j/o_j} s_j$ is a reaction C.

For instance, one possible execution of the `Delay` component of length 6 is:

$$0 \xrightarrow{1/0} 1 \xrightarrow{1/1} 1 \xrightarrow{0/1} 0 \xrightarrow{1/0} 1 \xrightarrow{1/1} 1 \xrightarrow{1/1} 1.$$

Exercise 2.1: Consider a modified version of the `Delay` component, called `OddDelay`, that has a Boolean input variable *in*, a Boolean output variable *out*, and two Boolean state variables *x* and *y*. Both the state variables are initialized to 0, and the reaction description is given by:

```
if $y$ then $out := x$ else $out := 0$;
$x := in$;
$y := \neg y$.
```

Describe in words the behavior of the component `OddDelay`. List a possible execution of the component if it is supplied with the sequence of inputs $0, 1, 1, 0, 1, 1$ for the first six rounds. ■

2.1.6 Extended-State Machines

State machines are commonly used for describing the behavior in model-based design. Figure 2.2 describes the component `Switch` that models a light switch. The component has a single Boolean input variable *press*. In every round, the value 1 for the input variable indicates that the switch is pressed. Initially, the light is off, and when the switch is pressed, it is turned on. The light gets switched off either when the switch is pressed again or if 10 rounds elapse without the switch being pressed.

In the state-machine notation, there is an implicit state variable, called the *mode* of the state machine, ranging over an enumerated type. For the component `Switch`, the mode ranges over the enumerated type $\{\texttt{off}, \texttt{on}\}$. The different modes of the machine are drawn as circles. This visual representation highlights different modes of operation for the component. The sourceless incoming arrow for the mode `off` indicates that the initial value of the variable *mode* is `off`.

In *extended* state machines, the description using the modes of the machine is augmented with additional state variables. In our example, the component `Switch` uses an additional state variable *x* of type `int`. The initialization arrow into the initial mode is labeled with the declaration of these additional state variables along with their initial values. In our example, there is a single additional state variable *x* of type `int`, and it is initialized to the value 0.

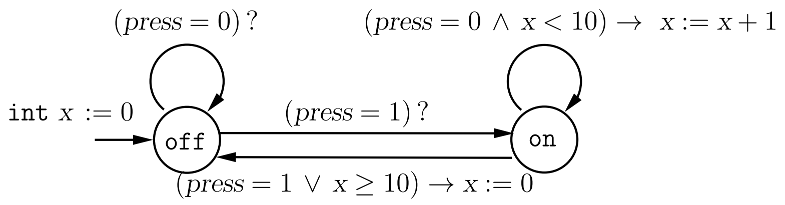

$(press = 0)\,?$
$(press = 0 \wedge x < 10) \rightarrow x := x + 1$
int $x := 0$
off
$(press = 1)\,?$
on
$(press = 1 \vee x \geq 10) \rightarrow x := 0$

Figure 2.2: Description of `Switch` as an Extended-State Machine

In extended-state machines, reactions are specified using mode-switches. A mode-switch is depicted as an edge between two modes and has an associated guard-condition and a code fragment to update variables. If the guard-condition is the expression *Guard* and the code to update variables is *Update*, then the edge is annotated with *Guard* $\rightarrow$ *Update*. If the guard-condition always holds (that is, it equals the constant 1), then it is omitted, and the edge is annotated only with the code *Update*; and if the update code does not modify any variables, then it is omitted, and the edge is annotated only with the guard-condition *Guard*?

In our illustrative example, we have four mode-switches. The mode-switch from `off` to `on` is guarded with the condition $press = 1$ and does not change the value of x; the mode-switch from `off` to `off` is guarded with the condition $press = 0$ and does not change the value of x; the mode-switch from `on` to `on` is guarded with the conjunctive condition $(press = 0 \wedge x < 10)$, and it increments the value of x; and the mode-switch from `on` to `off` is guarded with the disjunctive condition $(press = 1 \vee x \geq 10)$, and it resets the value of x to 0.

When the mode is `off`, if the input is 0, then the guard-condition for the mode-switch from `off` to `off` is satisfied, and this mode-switch is executed. The new mode is the same as the old mode `off`, and since there is no explicit update code, the value of the state variable x stays unchanged. If the input is 1, then the guard-condition for the mode-switch from `off` to `on` is satisfied, and this mode-switch is executed. As a result, the updated value of the mode is `on`, with the value of x unchanged.

When the mode is `on`, if the input is 0 and the value of x is still below the timeout-threshold 10, the mode-switch from `on` to `on` is executed, leaving the mode unchanged while incrementing x. Thus, the component can stay in the mode `on` for at most 10 consecutive rounds. When either the input *press* has the value 1 or the value of x reaches 10, the guard-condition for the mode-switch from `on` to `off` is satisfied. Executing this mode-switch updates the mode to `off` and x to 0.

In this example, each state can be represented as a pair, where the first component belongs to the enumerated type $\{\texttt{off}, \texttt{on}\}$ and the second component is an integer. The set $[\![Init]\!]$ of initial states contains the sole state $(\texttt{off}, 0)$.

We can associate a set $[\![React]\!]$ of reactions to formally capture the meaning of the mode-switches. Given a state that assigns values to the mode and x and an input value for *press*, we can obtain a reaction by executing the update code of a mode-switch out of the current mode, provided the corresponding guard-condition is satisfied. For every integer n, we have reactions

$$(\texttt{off}, n) \xrightarrow{1/} (\texttt{on}, n);\ (\texttt{off}, n) \xrightarrow{0/} (\texttt{off}, n);\ (\texttt{on}, n) \xrightarrow{1/} (\texttt{off}, 0);$$

for every integer $n < 10$, we have the reaction $(\texttt{on}, n) \xrightarrow{0/} (\texttt{on}, n+1)$; and for every integer $n \geq 10$, we have the reaction $(\texttt{on}, n) \xrightarrow{0/} (\texttt{off}, 0)$.

In this example, the component `Switch` has no output variables. In the presence of output variables, the update code associated with each mode-switch assigns values to all the output variables. Also, in our example, the guard-conditions of two mode-switches out of the same mode are disjoint, and thus there is no choice in terms of which mode-switch should be executed. In general, this assumption need not hold. In fact, the state-machine notation is a convenient way of specifying multiple choices as later examples will illustrate.

Exercise 2.2: Describe the component `OddDelay` from Exercise 2.1 as an extended-state machine with two modes. The mode of the state machine should capture the value of the state variable *y*, while the state variable *x* should be updated using assignments in the mode-switches. ■

Exercise 2.3: We want to design a reactive component with three Boolean input variables *x*, *y*, and *reset* and a Boolean output variable *z*. The desired behavior is the following. The component waits until it has encountered a round in which the input variable *x* is high and a round in which the input variable *y* is high, and as soon as both of these have been encountered, it sets the output *z* to high. It repeats the behavior when, in a subsequent round, the input variable *reset* is high. By default the output *z* is low. For instance, if *x* is high in rounds 2,3,7,12, *y* is high in rounds 5,6,10, and *reset* is high in round 9, then *z* should be high in rounds 5 and 12. Design a synchronous reactive component that captures this behavior. You may want to use the extended-state machine notation. ■

2.2 Properties of Components

2.2.1 Finite-State Components

In many embedded applications, it suffices to consider types with only finitely many values. The type `bool` and enumerated types are finitely valued, whereas the numerical types such as `nat`, `int`, and `real` are not finite. When all the variables of a component have finite types, the set Q_I of inputs, the set Q_O of outputs, and the set Q_S of states are all finite. This is the case for the

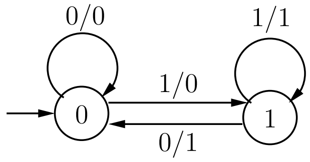

$$\begin{tikzcd} \arrow[r] & 0 \arrow[loop above, "0/0"] \arrow[r, shift left, "1/0"] & 1 \arrow[loop above, "1/1"] \arrow[l, shift left, "0/1"] \end{tikzcd}$$

Figure 2.3: Mealy Machine Corresponding to the Component `Delay`

component `Delay`. Such components are called finite-state components and are amenable to powerful automated analysis.

Finite-state Component

A synchronous reactive component C is said to be *finite-state* if the type of each of its input, output, and state variables is finite.

Note that the component `Switch` of figure 2.2 is not finite-state according to the definition due to the integer-valued state variable `x`. A close examination of this component reveals that, along every possible execution of the component, the value of x never exceeds 10. Thus, the only relevant range of values for x is from 0 to 10, and we can modify the description of `Switch` by changing the type of x to the range-type `int`$[0, 10]$. The resulting component is a finite-state component. In general, we allow restricting the numerical types `int`, `nat`, and `real` to the corresponding range-types `int`$[low, high]$, `nat`$[low, high]$, and `real`$[low, high]$, where *low* and *high* are numerical constants of the corresponding types. The resulting restricted versions of `int` and `nat` are finite types.

For a finite-state component, its behavior can be illustrated by a labeled finite graph. The nodes of the graph are states of the component. If s is an initial state of the component, then there is a sourceless edge incident on s. If $s \xrightarrow{i/o} t$ is a reaction of the component, then there is an edge from node s to node t labeled with input i and output o. Such graphs are called *Mealy machines*. Executions of the component are simply paths through this graph starting at an initial state.

The Mealy machine representation of the `Delay` component is shown in figure 2.3.

Exercise 2.4: Consider the component `OddDelay` from exercise 2.1. Is the component finite-state? Draw the corresponding Mealy machine. ■

2.2.2 Combinational Components

Consider the `Comparator` component shown in figure 2.4. The component has two input variables, in_1 and in_2, both of type `nat`. It has a Boolean output variable *out*. In each round, the component reads the inputs in_1 and in_2 and

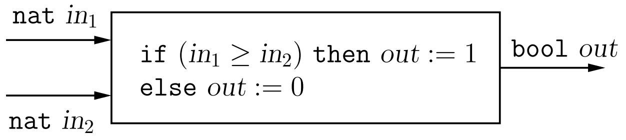


Figure 2.4: Combinational Component `Comparator`

sets the output variable *out* to 1 if the value of in_1 is greater than or equal to the value of in_2, and to 0 otherwise.

The component does not need to maintain any internal state and, hence, has no state variables. When there are no state variables, there is no initialization, and the reaction description assigns values to the output variables in terms of values of the input variables.

When the set S of state variables is empty, formally there is a unique valuation for S, and let us denote this unique state by $s_\emptyset$. This will also be the initial state. For every pair of natural numbers m and n, if $m \geq n$, `Comparator` has a reaction $s_\emptyset \xrightarrow{(m,n)/1} s_\emptyset$, and if $m < n$, then it has a reaction $s_\emptyset \xrightarrow{(m,n)/0} s_\emptyset$. A possible execution of the component is

$$s_\emptyset \xrightarrow{(2,3)/0} s_\emptyset \xrightarrow{(5,1)/1} s_\emptyset \xrightarrow{(40,40)/1} s_\emptyset.$$

Components such as `Comparator` without any state variables are called combinational.

Combinational Component

A synchronous reactive component C is said to be *combinational* if the set of its state variables is empty.

Note that the component `Comparator` corresponds to the Boolean-valued expression $in_1 \geq in_2$. A component C that has the variable *out* as one of its input variables can instead take both in_1 and in_2 as input variables, and we can substitute every occurrence of *out* in the reaction description of C by the expression $in_1 \geq in_2$ without changing its behavior. Conversely, note that the values of the expressions used in the reaction description of a component can be explicitly modeled as combinational components. For example, to take logical conjunction of two Boolean variables x and y, we can simply use the expression $x \wedge y$ or construct a combinational component that takes two input variables x and y and produces an output that has value 1 precisely when both input variables are 1. Whether a desired expression is modeled as a combinational component is a design choice, which is influenced by the primitives supported by the language used to describe the reactions.

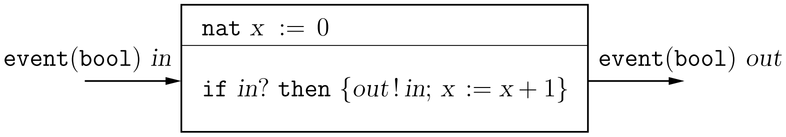


Figure 2.5: Event-Triggered Component `TriggeredCopy`

2.2.3 Event-Triggered Components*

In the synchronous execution model of reactive components, the notion of a *round* is global, and each component participates in every round. This may not be realistic in some scenarios, and we want to allow the possibility of a component to specify its own notion of a round. For example, a system may consist of multiple hardware components, each operating at a different clock frequency, and in this case, each component will participate in only those rounds in which its own clock signal is high. To model this behavior, we use input variables of type `event`. The basic type `event` is the enumerated type $\{\top, \bot\}$, where $\top$ denotes that the event is present and $\bot$ denotes that the event is absent. More generally, we allow the event type to be parameterized by another type: the event can be absent or can be present with a value belonging to the parameter type. For example, the type `event(bool)` has three values 0, 1, and $\bot$. For an event variable x, the Boolean expression $x?$, meaning x is present, stands for the expression $x \neq \bot$. Note that while input, output, and local variables can be of type `event`, a state variable cannot be of this type, as it is not meaningful to consider a state to be absent.

As an example, consider the component `TriggeredCopy`, shown in figure 2.5, that copies its input to output. The type of input variable *in* is `event(bool)`: in each round, the input can be absent and, if present, takes a Boolean value. Whenever the input is present, denoted by *in*?, it is copied to the output; when input is absent, so is the output. The type of the output variable *out*, thus, is also `event(bool)`. The assignment *out* := *in* is written as *out*! *in* to highlight that the event *out* is issued.

By default, an output event variable is absent, that is, if the event output variable *out* is not explicitly assigned a value during a round, then its value is assumed to be $\bot$. In the reaction description of the component `TriggeredCopy`, if the input event is absent, then the code does not assign any value to the output variable, and thus the component responds to an absent input with an absent output.

The component `TriggeredCopy` does maintain a state variable *x*: initially *x* is 0, whenever the input is present, *x* is incremented, and whenever the input is absent, *x* stays unchanged. Thus, the value of the state *x* shows the number of past rounds in which the input has been present. For every natural number n,

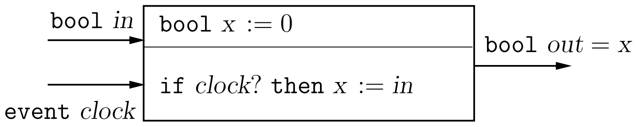


Figure 2.6: Event-Triggered Component `ClockedCopy`

the component has three reactions:

$$n \xrightarrow{\perp/\perp} n; \quad n \xrightarrow{0/0} n+1; \quad n \xrightarrow{1/1} n+1.$$

A sample execution of `TriggeredCopy` is

$$0 \xrightarrow{\perp/\perp} 0 \xrightarrow{0/0} 1 \xrightarrow{1/1} 2 \xrightarrow{\perp/\perp} 2 \xrightarrow{\perp/\perp} 2 \xrightarrow{1/1} 3.$$

We say that the input variable *in* is a *trigger* for the component `TriggeredCopy`, and the component is *event-triggered*. If the input is absent in a round, then the component is passive: the output is absent, and the state stays unchanged. Such a reaction is called a *stuttering* reaction. In the implementation, the component does not have to be "executed" to produce such a reaction.

As another example, consider the event-triggered component `ClockedCopy` shown in figure 2.6. It has a Boolean input variable *in* and an input event variable *clock* that acts as a trigger. Every time the clock event is present, the component updates its state variable x to the current value of the input variable *in*. Any changes to the input *in* during rounds in which the event *clock* is absent are ignored. The output *out* is a Boolean variable, and its value equals the updated state. Such an output variable is called a *latched* output. In this case, the component does not need to explicitly compute the value of *out*, and this is indicated by associating *out* with the state variable x in the declaration of the output.

The output variable *out* always has a value, even during rounds in which the trigger *clock* is absent, and equals the value of the input variable *in* from the most recent round in which the event *clock* was present. One possible execution of the component is shown below, where input is listed as a pair with the value of *in* followed by the value of *clock*:

$$0 \xrightarrow{(1,\perp)/0} 0 \xrightarrow{(1,\top)/1} 1 \xrightarrow{(0,\perp)/1} 1 \xrightarrow{(0,\perp)/1} 1 \xrightarrow{(0,\top)/0} 0.$$

Formally, for a synchronous reactive component C, an output variable y is said to be *latched* if there exists a state variable x such that in every reaction $s \xrightarrow{i/o} t$ of the component, the value of the output variable y is the updated value of the state variable x: $o(y) = t(x)$. In the implementation of a component, a latched output does not need to be explicitly stored or computed; the corresponding state variable needs to be made accessible to other components.

Now we can define the notion of an event-triggered component in its generality. Each of its output variables should be either latched or an event. When the triggering input events are absent, the state should stay unchanged (and, as a result, the latched outputs also stay unchanged), and event outputs should be absent.

EVENT-TRIGGERED COMPONENT

For a synchronous reactive component $C = (I, O, S, \mathit{Init}, \mathit{React})$, a set $J \subseteq I$ of input variables is said to be a *trigger* if:

1. every input variable in J is of type `event`;
2. every output variable either is latched or is of type `event`; and
3. if i is an input with all events in J absent (that is, for all input variables $x \in J$, $i(x) = \perp$), then for all states s, if $s \xrightarrow{i/o} t$ is a reaction, then $s = t$ and $o(y) = \perp$ for every output variable y of event type.

A component C is said to be *event-triggered* if there exists a subset $J \subseteq I$ of its input variables such that J is a trigger for C.

Exercise 2.5: Design an event-triggered combinational component `ClockedMax` with two input variables *x* and *y* of type `nat` and an input event variable *clock*. The output variable *z* of the component should be of type `event(nat)` such that the value of *z* should be the maximum of the inputs *x* and *y* during rounds in which *clock* is present. ■

Exercise 2.6: Design an event-triggered component `SecondToMinute` with the input event variable *second* and the output event variable *minute* such that *minute* is present every 60^{th} time the event *second* is present. ■

Exercise 2.7: Design a component `ClockedDelay` with a Boolean input variable *x*, an input event variable *clock*, and an output variable *y* of type `event(bool)` with the following behavior: if *clock* is present during rounds, say, $n_1 < n_2 < n_3 < \cdots$ then in round n_1, the output should be some default value, say 0; in round n_{j+1}, for each j, the output should equal the value of *x* in round n_j; and in the remaining rounds (that is, rounds during which the input event *clock* is absent), output should be absent. ■

2.2.4 Nondeterministic Components

In our examples so far, the components are *deterministic*: for a given sequence of inputs, the component has a unique execution producing a unique sequence of outputs. Such deterministic behavior is ensured if the component has a single initial state, and in every state, for a given input, there is exactly one possible reaction.

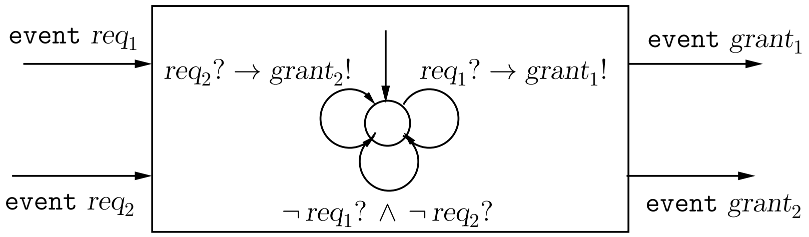


Figure 2.7: Nondeterministic Component `Arbiter`

Deterministic Component

A synchronous reactive component C is said to be *deterministic* if:

1. C has a single initial state, and
2. for every state s and every input i, there is precisely one output o and one state t such that $s \xrightarrow{i/o} t$ is a reaction of C.

The components `Delay`, `Switch`, `Comparator`, `TriggeredCopy`, and `ClockedCopy` are all deterministic. Determinism is a desirable property for components that are meant to be implemented.

Nondeterministic components, in contrast, can respond with different output sequences for the same input sequence. Such components are useful for modeling parts of the system that are not yet fully designed and for capturing constraints on the environment. As an example, consider the component `Arbiter` shown in figure 2.7. It has two input variables, req_1 and req_2, and two output variables $grant_1$ and $grant_2$, all of which are events. This component is designed to resolve contention among incoming requests. The dynamics of the component are described using the extended-state machine notation. The machine has only one mode, and thus no state needs to be maintained explicitly to record this mode. In each round, a mode-switch whose guard-condition is satisfied is chosen, and the corresponding update code is executed.

When only the request req_1 is present, the guard-condition of the mode-switch "$req_1? \to grant_1!$" is satisfied, and the event $grant_1$ is issued. Note that the event $grant_2$ is absent by default in this case. The case when only the request req_2 is present is symmetric. If both requests are absent, then the guard-condition of the mode-switch labeled "$\neg\, req_1? \wedge \neg\, req_2?$" is satisfied. For this mode-switch, there is no explicit update code, and thus both the output events are absent by default. However, if both input requests are present, then the guard-conditions $req_1?$ and $req_2?$ of two mode-switches are satisfied. In such a case, one of them gets executed. There are two possible reactions of the component: either $grant_1$ is present and $grant_2$ is absent or $grant_1$ is absent and $grant_2$ is present. Such a nondeterministic behavior captures what an arbiter should do, namely,

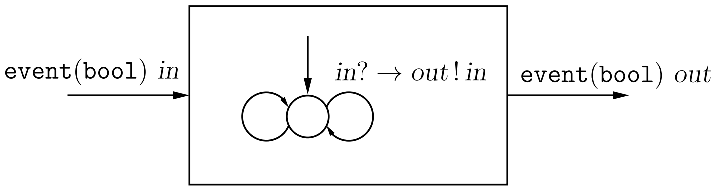


Figure 2.8: Nondeterministic Component `LossyCopy`

a grant output should be issued only when requested, and at most one grant output should be issued in any round, without constraining how the contention is resolved, leaving open the possibility of different implementations. Note that the component `Arbiter` is combinational and event-triggered.

As another example of a nondeterministic component, consider the combinational and event-triggered component `LossyCopy` shown in figure 2.8. It has an input event *in* and an output event *out*. The desired behavior of the component is that in each round, either the input is copied to the output or the output is absent. This is again described by a single-mode extended-state machine with two mode-switches: one with guard-condition *in*? and update code *out*!*in* and one with default guard-condition that always holds and default update code that does not assign any explicit value to *out*. When the input event is present, the guard-conditions of both the mode-switches are satisfied. In one case, the value of the input is issued on the output event; in the other case, no action is taken, and thus the output event is absent. When the input event is absent, only the mode-switch with default guard-condition is enabled, and the output event is absent. Such a component can be used to model potential loss of messages along a network link. One possible execution of the component is

$$s_\emptyset \xrightarrow{0/0} s_\emptyset \xrightarrow{1/\perp} s_\emptyset \xrightarrow{\perp/\perp} s_\emptyset \xrightarrow{1/1} s_\emptyset \xrightarrow{\perp/\perp} s_\emptyset \xrightarrow{0/\perp} s_\emptyset.$$

Exercise 2.8: Consider the `Delay` component of figure 2.1, and suppose we replace the reaction description by

$out := x;$
$x := \texttt{choose}(in, x)$

Describe in words the behavior of the modified component. Draw the Mealy machine corresponding to the component. ■

Exercise 2.9: For the nondeterministic component `Arbiter` of figure 2.7, the reactions are expressed using the extended-state machine notation. Write an equivalent description using straight-line update code. You can use a local variable whose value is assigned nondeterministically using the `choose` construct. ■

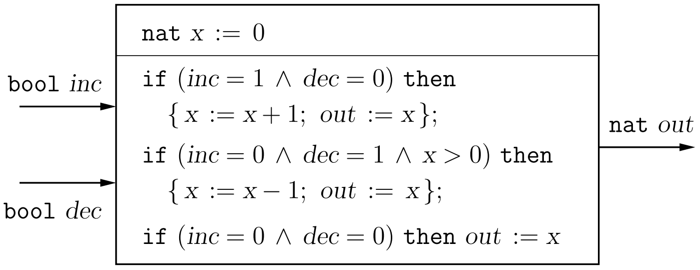

nat $x := 0$
bool inc
bool dec
nat out
if $(inc = 1 \wedge dec = 0)$ then
$\{x := x + 1;\ out := x\}$;
if $(inc = 0 \wedge dec = 1 \wedge x > 0)$ then
$\{x := x - 1;\ out := x\}$;
if $(inc = 0 \wedge dec = 0)$ then $out := x$

Figure 2.9: Component `Counter` with Input Assumptions

2.2.5 Input-Enabled Components

All the components we have seen so far have the following property: in every state and for every input, the component has *at least* one reaction. This property makes the components *input-enabled*. For a given sequence of inputs, an input-enabled component has at least one corresponding execution.

INPUT-ENABLED COMPONENT

For a synchronous reactive component C, an input i is said to be *enabled* in a state s if there exists an output o and a state t such that $s \xrightarrow{i/o} t$ is a reaction of C. The component C is said to be *input-enabled* if every input is enabled in every state.

There are design problems where it is useful to make assumptions about the inputs that the environment may supply. As an example, consider the component `Counter` of figure 2.9, which maintains a non-negative counter using a state variable x. The initial value of x is 0. The component has two Boolean input variables inc and dec: when the input inc is 1, the counter state is incremented by 1, and when the input dec is 1, the counter state is decremented by 1. The counter does not expect both input variables inc and dec to be 1 simultaneously, nor does it expect the counter to be decremented when the counter value is 0. We describe the reactions of `Counter` only for inputs satisfying this assumption as shown in figure 2.9. The output of the component is the updated value of x. When both input variables inc and dec are 1, and when dec is 1 with the state x equal to 0, the reaction description does not assign any value to the output, and thus there is no corresponding reaction.

In general, the input assumption can be a constraint on the sequence of inputs supplied to the component. When a component C with input assumptions is used as part of a larger system, we need to check that its input assumptions are indeed satisfied by the components that supply its inputs.

Note that, by definition, every deterministic component has exactly one reaction corresponding to a given state and input and, thus, is input-enabled.

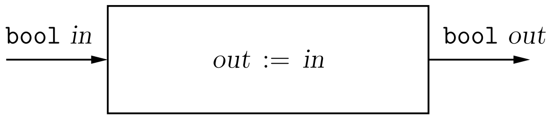


Figure 2.10: The Combinational Component `Relay`

Exercise 2.10: Design a nondeterministic component `CounterEnv` that supplies inputs to the counter of figure 2.9. The component `CounterEnv` has no inputs, and its outputs are the Boolean variables *inc* and *dec*. It should produce all possible combinations of outputs as long as the component `Counter` is willing to accept these as inputs: it should never set both *inc* and *dec* to 1 simultaneously, and it should ensure that the number of rounds with *dec* set to 1 never exceeds the number of rounds with *inc* set to 1. ■

2.2.6 Task Graphs and Await Dependencies

Let us consider a component `Relay`, shown in figure 2.10, that has a Boolean input variable *in* and a Boolean output variable *out*. The component `Relay` is a combinational component without any state variables and, in each round, simply copies the input to the output.

Let us compare the components `Relay` and `Delay`. Observe that they have identical input/output variables. In a given round, the output of the component `Delay` does not depend on its input in that round, whereas the component `Relay` can produce its output only after reading the input for the current round. Intuitively, the output of `Relay` must *await* its input, whereas this is not necessary for `Delay`. This crucial *intra-round dependency* of output variables on input variables can impact how components can be composed.

In the current reaction description of the component `Delay`, given the input and the current state, the update code consisting of two assignments computes the output and updated state. This monolithic description hides the intra-round independence of the output on the input. To avoid this problem, we allow the reaction description to be split into multiple tasks. This is illustrated by revising the description of the component `Delay` to obtain the component `SplitDelay` shown in figure 2.11.

The reaction description for the component `SplitDelay` is split into execution of two tasks A_1 and A_2. The task A_1 computes the value of the variable *out* using the value of the state variable x, whereas the task A_2 updates the value of the state variable x using the value of the input variable *in*. In general, each task has an associated *read-set* R and *write-set* W of variables, and it assigns values to variables in the write-set given the values of variables in the read-set. Note that the write-set of a task should not include any input variables of the component. The read-/write-sets can also include local variables used in

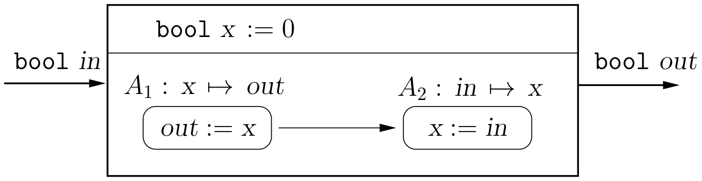


Figure 2.11: Component `SplitDelay` with Split Reaction

the reaction description. Since an output variable is used for communication with other components, it should be written by exactly one task. With this restriction, once the task A of a component C responsible for writing an output variable y executes, the value of y will no longer change within the current round and can be used by other components, even if some other tasks within the component C have not yet executed.

In our illustrations, we depict tasks as rectangular boxes with rounded corners. The declaration

$$A \, : \, x_1, x_2, \ldots x_m \; \mapsto \; y_1, y_2, \ldots y_n$$

indicates that the task A has the read-set $R = \{x_1, x_2, \ldots x_m\}$ and the write-set $W = \{y_1, y_2, \ldots y_n\}$. The *update description* of a task describes its computation as a sequence of assignment and conditional statements or, alternatively, using the extended-state machine notation. The update description can be nondeterministic: the assignments may use the `choose` construct, and in the extended-state machine, guards of multiple mode-switches out of the same mode may be simultaneously satisfied. Thus, the mathematical semantics of the update description *Update* of a task is a relation $[\![\mathit{Update}]\!]$ between the values of variables in the read-set and the values of variables in the write-set; that is, $[\![\mathit{Update}]\!]$ contains pairs of the form (s, t) with $s \in Q_R$ and $t \in Q_W$.

In example of figure 2.11, for the task A_1, the read-set is $\{x\}$, the write-set is $\{out\}$, and the update is described by the assignment $out := x$; and for the task A_2, the read-set is $\{in\}$, the write-set is $\{x\}$, and the update is described by the assignment $x := in$.

When the reaction description is split into multiple tasks, we need to specify constraints on the order in which the tasks should be executed. In our example of `SplitDelay`, the task A_1 must be executed before the task A_2; executing the assignment statements corresponding to these two tasks in this order is necessary for the desired behavior. We write $A_1 \prec A_2$ to express the precedence constraint that the task A_1 should be executed before the task A_2. In our illustrations, the precedence constraint $A_1 \prec A_2$ is captured by an arrow from the task A_1 to the task A_2. Thus, in the *task graph* description of the component, nodes correspond to tasks (with associated read-sets, write-sets, and update descriptions), and edges correspond to precedence constraints on the order of execution of the tasks within a round.

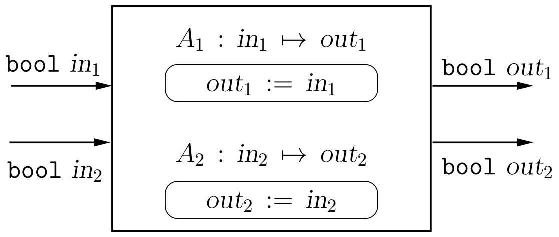


Figure 2.12: Component `ParallelRelay`: out_1 awaits in_1 and out_2 awaits in_2

Given the precedence relation $\prec$ over tasks, the relation $\prec^+$ denotes the *transitive closure* of the relation $\prec$: for two tasks A and A', $A \prec^+ A'$ holds if there is a path from A to A' in the task graph. In other words, if there is a chain of precedence constraints $A_1 \prec A_2 \prec \cdots \prec A_n$, for $n > 1$, then $A_1 \prec^+ A_n$. If $A \prec^+ A'$ holds, then the task A must execute before the task A'. Thus, the relation $\prec^+$ captures all the constraints on the execution order implied by the precedence constraints. We require that the precedence relation $\prec$ is *acyclic*: there is no task A such that $A \prec^+ A$, that is, the task graph does not contain any cycles. In particular, if $A_1 \prec A_2$ is a precedence constraint, then we cannot have the constraint $A_2 \prec A_1$.

The precedence constraints captured by the relation $\prec$ lead to *await* dependencies between output and input variables. Output variables written by a task must await the input variables this task reads, and if $A_1 \prec^+ A_2$, then the output variables written by A_2 must await the input variables read by A_1. We write $y \succ x$ if the output variable y awaits the input variable x according to the precedence constraints $\prec$ over the tasks.

In our example component `SplitDelay`, by the above definitions, the output variable *out* does not await the input variable *in*. By default, when the reaction description of a component is not explicitly split into tasks, it can be viewed as a single task with its read-set containing all the input and state variables, its write-set containing all the output and state variables, and the update description same as the reaction description. In such a case, each output variable awaits each input variable. Then for both `Relay` and `Delay` components, the output variable *out* awaits the input variable *in*.

An output variable may await only some of the input variables, and different output variables may await different input variables. This can be captured if we allow the precedence relation $\prec$ to express ordering constraints among tasks only partially. To illustrate this point, consider the combinational component `ParallelRelay` (see figure 2.12) with two input variables, in_1 and in_2, and two output variables, out_1 and out_2. In each round, the component copies the input variable in_1 to output variable out_1 and copies the input variable in_2 to output variable out_2. We wish to express that out_1 awaits in_1 but not in_2, and out_2

awaits in_2 but not in_1. This is achieved by splitting the reaction description into two tasks, A_1 and A_2. The task A_1 reads in_1 and writes out_1, whereas the task A_2 reads in_2 and writes out_2. There is no precedence constraint between the two tasks, and thus the task graph has no edges. This means that these two tasks are independent and can be executed in any order (and even concurrently if implemented on parallel hardware).

A *task schedule* is a linear ordering of all the tasks that is consistent with the precedence relation. For `SplitDelay`, we have only one task schedule A_1, A_2, whereas for `ParallelRelay`, we have two possible task schedules, A_1, A_2 and A_2, A_1. In general, for a task graph with k tasks, a schedule is an ordering $A_1, A_2, \ldots A_k$ of all the tasks, such that if there is a precedence constraint from the task A to task A', then A must appear before A' in the schedule.

Thus, the ordering constraints expressed by $\prec$ can allow multiple schedules. We can allow any binary relation $\prec$ over tasks as the precedence relation as long as it obeys the following rules. As already discussed, the precedence relation should be acyclic, and this ensures that there is at least one possible way of ordering all the tasks to get a schedule. Second, if a task A_2 is reading an output or a local variable y, then the precedence constraints should enforce that the value of y is already computed when A_2 executes in every possible schedule. This is ensured if there exists a task A_1 that writes y, such that $A_1 \prec^+ A_2$ holds. Third, we want to ensure that two tasks that are independent according to the precedence constraints can be executed in either order without affecting the result of the execution. For example, in the component `SplitDelay`, the task A_1 reads x, and the task A_2 writes x. If these two tasks were unordered (that is, if the precedence edge $A_1 \prec A_2$ was missing), then it should be considered a syntactic error as the result depends on which of the two tasks executes first. In general, two tasks have a *write-conflict* if there is a variable that belongs to the write-set of one of the tasks and also belongs to either the read-set or the write-set of the other task. The tasks A_1 and A_2 have a write-conflict in `SplitDelay` but not in `ParallelRelay`. When two tasks have a write-conflict, the order in which they execute matters, and hence the precedence relation should not leave their ordering unconstrained.

As an illustrative example of the general specification of the tasks, consider the task graph shown in figure 2.13 for a component with state variables x_1 and x_2, input variables in_1 and in_2, and output variables out_1, out_2, and out_3. The reaction description also uses a local variable y. Each output variable is written by exactly one task: out_1 by task A_3, out_2 by task A_2, and out_3 by task A_4. The precedence relation given by $A_1 \prec A_3$, $A_1 \prec A_4$, and $A_2 \prec A_4$ is acyclic. The task A_4 reads the output out_2, and this is legal since it has a precedence-edge from the task A_2 that writes out_2. Similarly, the local variable y is guaranteed to be written by the task A_1 before it is used by A_4. The task A_2 is not ordered with respect to the tasks A_1 and A_3 according to the precedence constraints, and this means that the task A_2 should have no write-conflicts with both A_1 and A_3. This is indeed the case. Finally, verify that for each of the state variables x_1

and x_2, the task that writes to the variable is ordered with respect to the tasks that read this variable. For instance, the task A_2 will read the "old" value of x_2, whereas the task A_3 will read the value of x_1 updated by A_1 and rewrite it. The await dependencies implied by the precedence constraints are: the output variable out_1 awaits the input in_1, the output variable out_2 does not await any inputs, and the output variable out_3 awaits both the input variables in_1 and in_2. Thus, $out_1 \succ in_1$, $out_3 \succ in_1$, and $out_3 \succ in_2$.

The definition and rules for splitting the update into multiple tasks, and the await dependencies they induce among output and input variables, are summarized below.

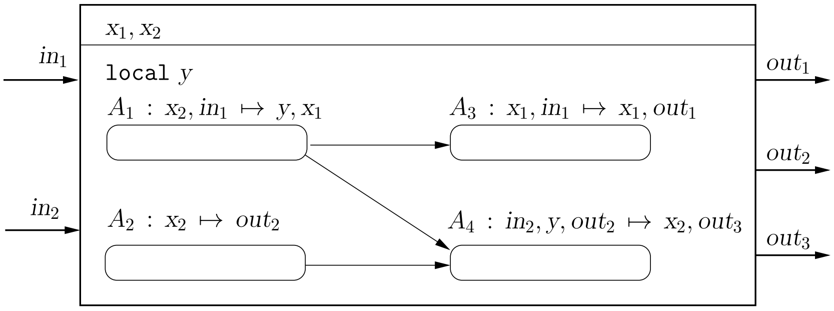
Task Graphs and Await Dependencies

For a synchronous reactive component C with input variables I, output variables O, and state variables S, a task-graph description of the reactions using a set L of local variables consists of a set of tasks and a binary precedence relation $\prec$ over these tasks. Each task A has a read-set $R \subseteq I \cup S \cup O \cup L$, a write-set $W \subseteq O \cup S \cup L$, and an update description *Update* with $[\![Update]\!] \subseteq Q_R \times Q_W$ such that:

1. The precedence relation $\prec$ is acyclic.
2. Each output variable belongs to the write-set of exactly one task.
3. If an output or a local variable y belongs to the read-set of a task A, then there exists a task A' such that y is in the write-set of A' and $A' \prec^+ A$.
4. If a state or a local variable x belongs to the write-set of a task A and also to either the read-set or write-set of a different task A', then either $A \prec^+ A'$ or $A' \prec^+ A$.

For an output variable y and an input variable x, y *awaits* x (also written $y \succ x$), precisely when for the unique task A, such that y belongs to the write-set of A, either x belongs to the read-set of A, or there exists a task A' such that $A' \prec^+ A$ and x belongs to the read-set of A'.

To execute a component with a task-graph description of reactions, we choose a schedule that orders all the tasks in a manner consistent with the precedence relation. The tasks are then executed one by one in this order. Executing a task A means assigning values to the variables in its write-set based on the values of the variables in its read-set. Note that our consistency requirements make sure that an output variable gets assigned a value exactly once, and if it is read by a task A, then the output value has already been assigned when A is executed. Also, if a task A reads a state variable x, then either A reads the value of x at the beginning of the round (this happens if there is no task with x in its write-set that precedes A), or there is a unique task A' with x in its write-set such that A always reads the value written by A' (irrespective of the schedule



x_1, x_2
in_1
in_2
local y
$A_1 : x_2, in_1 \mapsto y, x_1$
$A_2 : x_2 \mapsto out_2$
$A_3 : x_1, in_1 \mapsto x_1, out_1$
$A_4 : in_2, y, out_2 \mapsto x_2, out_3$
out_1
out_2
out_3

Figure 2.13: Illustrative Task Graph

chosen) since the precedence relation totally orders A with respect to all the tasks that write to x.

In our example of figure 2.13, there are five possible schedules:

$$A_1, A_2, A_3, A_4; \;\; A_1, A_2, A_4, A_3; \;\; A_1, A_3, A_2, A_4; \;\; A_2, A_1, A_3, A_4; \;\; A_2, A_1, A_4, A_3.$$

The set of possible reactions of the component will depend on the update descriptions of the four tasks but is independent of the schedule.

Properties such as determinism and input-enabledness can be naturally defined for tasks so that they imply the corresponding properties of the component:

- **Deterministic Tasks:** A task A with read-set R, write-set W, and update description *Update* is said to be *deterministic* if for every valuation s over R, there exists a unique valuation t over W such that $(s, t) \in [\![Update]\!]$. Thus, given values for the read variables, a deterministic task assigns unique values to the variables it writes. If a component has a single initial state and all the tasks in the task-graph description of reactions are deterministic, then the component must be deterministic. This is because the requirements on what constitutes a legal precedence relation ensure that the schedule does not affect the result of executing tasks.

- **Input-Enabled Tasks:** An update task A with read-set R, write-set W, and update description *Update* is said to be *input-enabled* if for every valuation s over R, there exists at least one valuation t over W such that $(s, t) \in [\![Update]\!]$. Thus, given values for the read variables, an input-enabled task produces at least one result. Now consider a component with a task-graph description of its reactions so that all the tasks are input-enabled. Given a state and an input, we can execute all the update tasks in an order consistent with the precedence constraints. Since each task is input-enabled, there is a way to progress at every step, and thus the component can produce at least one reaction. Thus, such a component is input-enabled.

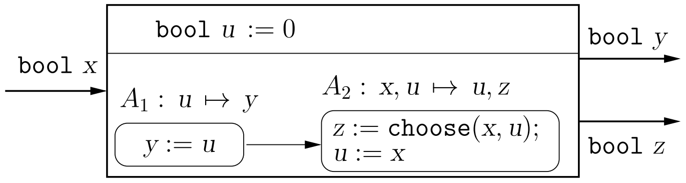


Figure 2.14: Component Specified Using Task Graph

Exercise 2.11: Consider the component from exercise 2.3. Split the reaction description into two tasks so that the output z awaits the inputs x and y but not the input *reset*. ■

Exercise 2.12: Consider a synchronous reactive component C with an input variable x and output variables y and z. The component has two tasks, A_1 and A_2, such that the output y belongs to the write-set of the task A_1, and the output z belongs to the write-set of the task A_2. If we know that the output y awaits the input x, but the output z does not await x, then what can we conclude regarding the precedence constraints between the tasks A_1 and A_2? ■

Exercise 2.13: Consider the synchronous reactive component shown in figure 2.14. List all the possible reactions of the component. Does the output y await x? Does the output z await x? ■

Exercise 2.14: Design a synchronous reactive component `ComputeAverage` with an integer input variable x, an input event variable *clock*, and a real-valued output variable y with the following behavior: in the first round, the output y is 0; in a subsequent round i, let $j < i$ be the most recent round before round i in which the input event *clock* is present (if *clock* is absent in all rounds before i, then let $j = 0$), the output should be the *average* of input values for x in rounds j, $j+1$, upto $i-1$. The following is a sample behavior of the desired component:

Clock	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$
x	5	2	-3	1	6	5	-2	11
y	0	5	3.5	-3	-1	1.33	2.25	-2

The component should be designed so that the output y does not await any of the input variables. ■

2.3 Composing Components

2.3.1 Block Diagrams

Suppose we want to design a reactive component with a Boolean input variable *in* and a Boolean output variable *out*, such that in the first two rounds the

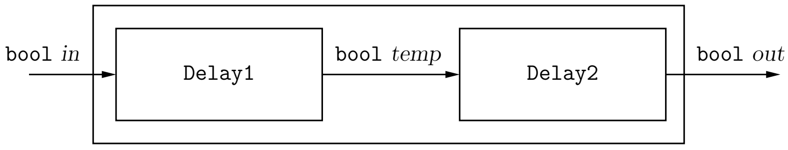


Figure 2.15: Block Diagram for `DoubleDelay` from Two `Delay` Components

output is 0 and in every subsequent round n, the output equals the input in round $n-2$. Instead of designing this component from scratch, we would like to reuse the component `Delay`. Composing two `Delay` components in series gives the desired component. The resulting design of the component `DoubleDelay` is shown in figure 2.15. The design of the component should be obvious from the block diagram, and given the intuitive appeal of such diagrammatic descriptions, almost all tools for high-level embedded systems design support such diagrams. A careful examination of the block diagram reveals that there are three operations on components in such a diagram:

- **Instantiation:** The components `Delay1` and `Delay2` are both instances of the component `Delay`. Such instances are obtained by renaming the input/output variables. For example, the component `Delay1` is exactly like the component `Delay` except its output variable is called *temp* instead of *out*.

- **Parallel Composition:** The two components `Delay1` and `Delay2` run in parallel. The block diagram shows that the output of the component `Delay1` is the same as the input of the component `Delay2`, and this achieves communication between the two components. The communication is synchronous. In each round, the component `Delay1` reads its input *in*, produces output *temp*, and updates its internal state to record the current value of *in*. In the same round, the component `Delay2` reads its input *temp* — as supplied by the component `Delay1`, produces its output *out*, and updates its internal state to record the current value of *temp*.

- **Output Hiding:** For the component `DoubleDelay`, the relevant output variable is *out*, and the variable *temp* is only an auxiliary variable that is used in implementing `DoubleDelay`. The block diagram shows that the variable *temp* is local and not exported to the outside world.

The component `DoubleDelay` is textually defined as

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

We proceed to discuss the three operations in the above expression, namely, parallel composition $\parallel$, renaming $\mapsto$, and hiding $\setminus$, in more details.

2.3.2 Input/Output Variable Renaming

Before composing and connecting components, we may need to rename variables so that there are no name conflicts among state variables of different components, and common names for input/output variables indicate desired input/output connections. It is common practice to assume that the renaming of state variables is *implicit* and performed mechanically without burdening the designer. For instance, in figure 2.15, we can assume that the state variable of the component `Delay1` is called x_1 instead of x, and the state variable of the component `Delay2` is called x_2. The renaming of input/output variables needs to be defined *explicitly* since it establishes the intended communication pattern.

Let $C = (I, O, S, \mathit{Init}, \mathit{React})$ be a synchronous reactive component, x be an input or an output variable, and y be a *fresh* variable (that is, y is not a state, input, or output variable of C), such that the types of x and y are the same. Then the component obtained by *renaming x to y in C*, denoted $C\,[\,x \mapsto y\,]$, is the synchronous reactive component obtained by substituting the variable name x by y in the description of C.

With this notation, the component `Delay1` is defined as `Delay`$\,[\,\mathit{out} \mapsto \mathit{temp}\,]$. For the component `Delay1`, the set of input variables is $\{\mathit{in}\}$, the set of output variables is $\{\mathit{temp}\}$, the set of state variables is $\{x_1\}$, the initialization is $x_1 := 0$, and the reaction description is $\mathit{temp} := x_1;\ x_1 := \mathit{in}$. Similarly, the component `Delay2` is defined as `Delay`$\,[\,\mathit{in} \mapsto \mathit{temp}\,]$.

Observe that variable renaming does not change properties of a component. For instance, if a component is deterministic, so is its renamed instance, and if a component is event-triggered, so is its renamed instance.

2.3.3 Parallel Composition

The parallel composition operation combines two components into a single component whose behavior captures the synchronous interaction between the two components running concurrently.

Compatibility in Variable Names

Consider $C_1 = (I_1, O_1, S_1, \mathit{Init}_1, \mathit{React}_1)$ and $C_2 = (I_2, O_2, S_2, \mathit{Init}_2, \mathit{React}_2)$. Before we can compose these two components, we need to check for *compatibility* in their variable declarations. First, there should be no name conflicts concerning state variables. If x is a state variable of C_1, then no variable of C_2 should be called x. That is, the set S_1 should be disjoint from each of the sets I_2, O_2, and S_2; symmetrically, the set S_2 should be disjoint from each of I_1, O_1, and S_1. Note that the names of state variables are really private to a component. We can always rename them to avoid name conflicts before taking the composition. For instance, the variable name may be prefixed by the name of the component instance. Henceforth, we will assume that names of state variables are chosen according to a scheme that avoids name conflicts. Similarly, if the reaction description uses local variables, we will assume that the names of these local variables are unique and do not conflict with the names of other variables.

Second, a variable can be an input variable to both the components, and an output variable of one component can be an input variable to the other, but a variable cannot be an output variable of both the components. That is, the sets O_1 and O_2 should be disjoint. A consequence of this requirement is that only one component is responsible for controlling the value of any given variable.

Product Variables

When two components C_1 and C_2 are compatible, we want to define their parallel composition, denoted $C_1 \| C_2$, to be another synchronous reactive component C. We will also refer to the composition C as the *synchronous product* of the components C_1 and C_2. We proceed to describe how to construct the input variables, output variables, state variables, initialization, and reaction description of the product C.

Each state variable of a component is a state variable of the product. That is, the set S of state variables of C is the union $S_1 \cup S_2$. Each output variable of a component is an output variable of the product. That is, the set O of output variables of C is the union $O_1 \cup O_2$. Each input variable of a component is an input variable of the product, provided it is not an output variable of the other component. That is, the set I of input variables of C is the set $(I_1 \cup I_2) \setminus O$, denoting the difference of the two sets $I_1 \cup I_2$ and O.

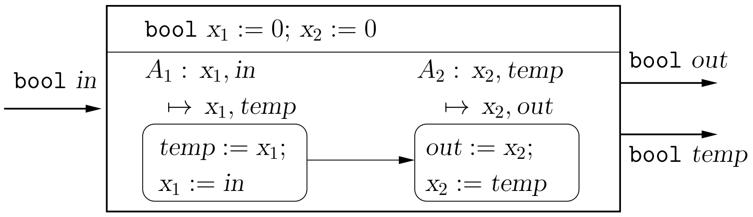
For example, the composition of the components `Delay1` and `Delay2` gives the component with state variables $\{x_1, x_2\}$, output variables $\{temp, out\}$, and input variables $\{in\}$.

Product States

A state of the product C assigns values to variables in S_1 as well as variables in S_2. The initial states of C are obtained by choosing the values for variables in S_1 according to the initialization $Init_1$ of the component C_1 and choosing the values for variables in S_2 according to the initialization $Init_2$ of the component C_2. If the two initializations $Init_1$ and $Init_2$ are given as sequences of assignment statements, then the initialization $Init$ for the product can be defined to be $Init_1; Init_2$ or, equivalently, $Init_2; Init_1$ since there can be no write-conflicts in the two blocks of initial assignments. In the example of composing the components `Delay1` and `Delay2`, the sole initial state of the product assigns the value 0 to both the state variables x_1 and x_2, and this can be described by the initialization $x_1 := 0;\ x_2 := 0$.

Reaction Description of the Product

Let us consider how to obtain the reaction description and the corresponding set of reactions of the product. If the reaction descriptions of the two components



bool $x_1 := 0$; $x_2 := 0$
bool *in*
$A_1 : x_1, in \mapsto x_1, temp$
$A_2 : x_2, temp \mapsto x_2, out$
$temp := x_1$; $x_1 := in$
$out := x_2$; $x_2 := temp$
bool *out*
bool *temp*

Figure 2.16: Parallel Composition of `Delay1` and `Delay2`

C_1 and C_2 use local variables L_1 and L_2, respectively, then the set of local variables for the reaction description of the product is $L_1 \cup L_2$.

If there is no communication between the two components, and this is the case when outputs of one component are not inputs to the other, then the two components can be executed independently. To obtain the reactions of the product, we can execute the update code of one followed by the other, and the order would not matter. However, when outputs of one component are inputs to the other, we have ordering constraints on how components should execute within a round. If the input/output connections are only one way, as is the case in the example of composing the components `Delay1` and `Delay2`, where the output of `Delay1` is an input to `Delay2` but not vice versa, then we can execute the updates of the components in the order suggested by the connections: we can first execute the component `Delay1` and then execute the component `Delay2`. In other words, the reaction description for the product consists of a task graph with two tasks, the task A_1 corresponding to the reaction description of `Delay1`, the task A_2 corresponding to the reaction description of `Delay2`, and a precedence edge from A_1 to A_2. The product is shown in figure 2.16. The reactions of the product are listed below, where in each state we list the values of x_1 and x_2, in that order, and in each output, we list the values of *temp* and *out*, in that order:

$$(0,0) \xrightarrow{0/(0,0)} (0,0);\quad (0,0) \xrightarrow{1/(0,0)} (1,0);\quad (0,1) \xrightarrow{0/(0,1)} (0,0);\quad (0,1) \xrightarrow{1/(0,1)} (1,0);$$
$$(1,0) \xrightarrow{0/(1,0)} (0,1);\quad (1,0) \xrightarrow{1/(1,0)} (1,1);\quad (1,1) \xrightarrow{0/(1,1)} (0,1);\quad (1,1) \xrightarrow{1/(1,1)} (1,1).$$

Composing Task Graphs

As another example of parallel composition, consider the composition shown in figure 2.17, in which the output *out* of the component `SplitDelay` is connected to the input of the component `Inverter` and vice versa. The component `Inverter` is a combinational component that sets its output to the negation of its input. This form of cyclic composition is called *feedback composition.* The result of the composition is the product component shown in figure 2.18. The product has no input variables, two output variables *in* and *out*, and one state

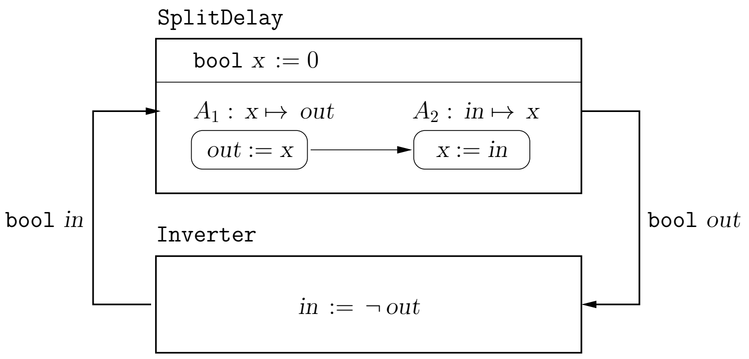


Figure 2.17: Feedback Composition of a `SplitDelay` with an `Inverter`

variable x. The task-graph description of the update for `SplitDelay` suggests an execution schedule in each round for the product. In each round, we first execute the task A_1 of `SplitDelay` to assign a value to the variable *out*. Now, the component `Inverter` can be executed as its input is available, and this assigns a value to the variable *in*. Subsequently, the task A_2 of `SplitDelay` can be executed using this value. In essence, we are constructing the task graph of the product by merging the task graphs of the two components. In this case, the component `Inverter` has a single task A, which reads *out* and writes *in*. We retain the original precedence constraints (the edge from A_1 to A_2 in the task graph of `SplitDelay`) and add additional precedence edges to reflect variable dependencies. The general rule for these additional cross-component edges is:

> If a task A belonging to one component reads a variable y, which is an output variable of the other component, then add a precedence edge from the unique task that writes y to the task A.

This rule gives us the edge from the task A_1 of `SplitDelay` to the task A of `Inverter` as the latter reads the variable *out* computed by A_1, and the edge from the task A of `Inverter` to the task A_2 of `SplitDelay` as the latter reads the variable *in* computed by A. In the first round, *out* is 0 and *in* is 1, and in every subsequent round, both of these values toggle. That is, the sequence of outputs produced by the product, listing the value of *in* first and *out* second, is $10, 01, 10, 01, 10, \ldots$

Now we can propose a precise definition for the reaction description of the product. We assume that the reaction descriptions for both the components C_1 and C_2 are given as task graphs. Recall that when the reaction description is not explicitly split into tasks, we interpret it as a single task that reads all the state and input variables and writes all the state and output variables, thus leading to await dependency of each of the output variables on all of the input variables. The set of tasks in the product is the union of the tasks of the two components. The precedence relation $\prec$ of the product is the union of the

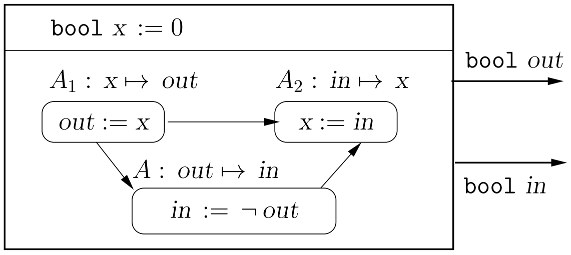


Figure 2.18: Parallel Composition of `SplitDelay` and `Inverter`

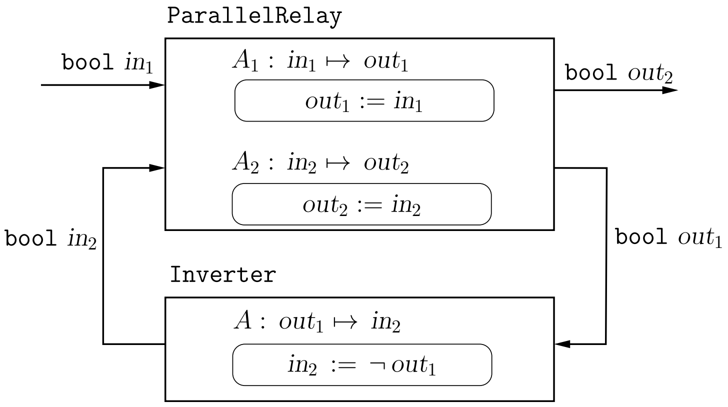


Figure 2.19: Parallel Composition of `ParallelDelay` and `Inverter`

precedence relations $\prec_1$ and $\prec_2$ of the component task graphs, together with the cross-component edges according to the rule above.

As another example, consider the composition of the components `ParallelRelay` and `Inverter`, shown in figure 2.19. The two tasks A_1 and A_2 are independent in `ParallelRelay`. Since the component `Inverter` reads out_1 and writes in_2, we get cross-component edges from A_1 to A and from A to A_2. This implies a new transitive precedence constraint: the task A_1 must be executed before A_2 in the product.

Acyclicity of Await Dependencies

The cross-component precedence edges can lead to a cycle among precedence constraints. This problem can be traced to cycles in the input/output await dependencies. Observe that in the feedback composition of the components `SplitDelay` and `Inverter` of figure 2.17, for the `Inverter` component, the variable *in* awaits the variable *out*, but for the `SplitDelay` component, there is no await dependency between the variables *out* and *in*. The absence of mutual

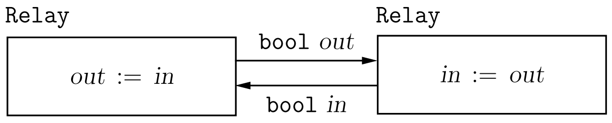


Figure 2.20: Ill-Formed Combinational Loop with Two `Relay` Components

await dependency is a key for well-formed behavior of the product. Composing components with mutually cyclic await dependencies can lead to unexpected behaviors, even when the individual components are deterministic. Let us illustrate two kinds of basic problems using two examples.

Figure 2.20 shows composition of two `Relay` components: the left component copies its input *in* to its output *out*, whereas the right component copies its input *out* to its output *in*. For one component, the variable *out* awaits the variable *in*, whereas for the other, the variable *in* awaits the variable *out*, and it is not possible to order the updates of the components in a consistent manner. If we just consider the set of reactions of the two components and mathematically compose these sets to obtain reactions that are consistent with the descriptions of both, then in each round, the product can produce two outputs: one possibility is that both the variables *in* and *out* are set to 0, and the other possibility is that both the variables are set to 1. Thus, if we were to allow composition of these two components, then we would obtain a nondeterministic component by composing deterministic ones.

A converse problem to the one of multiple possible consistent reactions arises in composition of an inverter component with a relay component shown in figure 2.21. The left component `Inverter` sets its output to the negation of its input, and the right component `Relay` copies its input to its output. For the left component, the variable *out* awaits the variable *in*, whereas for the right component, the variable *in* awaits the variable *out* causing cyclic await dependencies. In this case, there is no assignment of values to the two variables that is consistent with the reactions of the two components, and thus the product would have no behaviors.

Thus, it is imperative to detect cyclic await dependencies and rule out such ill-formed compositions. Let $\succ_1$ and $\succ_2$ represent the await dependencies of the two components. For instance, in figure 2.20, the await dependency for the top component is $out \succ_1 in$ and for the bottom component is $in \succ_2 out$. The union of the two relations gives the cycle $out \succ_1 in \succ_2 out$. We allow two components to be composed only when the union of the two await-dependency relations $\succ_1 \cup \succ_2$ is acyclic, that is, there do not exist common input/output variables $x_1, x_2, \ldots x_n$, with $x_n = x_1$, such that for each $1 \leq j < n$, x_{j+1} awaits x_j according to either one of the two await-dependency relations. This condition can be checked automatically by the compiler for the modeling language.

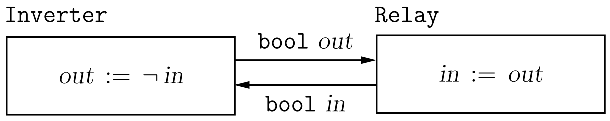


Figure 2.21: Ill-Formed Combinational Loop with a `Relay` and an `Inverter`

The compatibility conditions for two components to be composable are summarized in the definition below.

Component Compatibility

The components C_1 with input variables I_1, output variables O_1, and input/output await-dependency relation $\succ_1 \subseteq O_1 \times I_1$, and C_2 with input variables I_2, output variables O_2, and input/output await-dependency relation $\succ_2 \subseteq O_2 \times I_2$, are said to be *compatible* if (1) the sets O_1 and O_2 are disjoint, and (2) the relation $(\succ_1 \cup \succ_2)$ is acyclic.

We can take parallel composition of two components only when they are compatible by the above definition. Observe that by this definition, in figure 2.18, if we replace the component `SplitDelay` with the component `Delay`, then the composition is not allowed as the compatibility check would fail. This is because the component `Delay` has a single task, and thus its output *out* awaits its input *in*. Thus, our approach to ensuring well-behaved composition is conservative as it relies on the syntactic decomposition of the reaction description into tasks given by the designer.

Interfaces

One appealing feature of the compatibility check is that it refers only to input/output variables and their await dependencies. We can think of the inputs I, outputs O, and await-dependency relation $\succ \subseteq O \times I$ as an *interface* for the component. To form block diagrams consisting of multiple components, the designer can focus only on the interfaces of components to ensure compatibility and consistent usage and does not need to know internal details such as state variables and task graphs.

As an example of compatibility check using interfaces, consider the components shown in figure 2.22. The interface of the component C_1 corresponds to the task graph illustrated in figure 2.13. The interface simply shows the input variables in_1 and in_2, the output variables out_1, out_2, and out_3, and the dependencies that the output out_1 awaits in_1 and the output out_3 awaits both in_1 and in_2. The block-diagram connects this component C_1 with another component C_2. The interface of C_2 shows its input variables to be out_1, out_2, and out_3, its output variables to be in_1 and in_2, and the dependencies that in_1 awaits out_2 and in_2 awaits both out_1 and out_2. Verify that there is no cycle in the combined

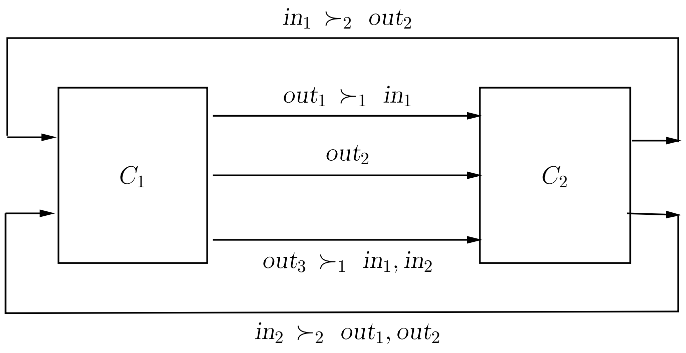


Figure 2.22: Composing Interfaces

dependencies and the interfaces are compatible. This suffices for us to conclude that the composition illustrated in figure 2.22 is well defined.

The next proposition asserts that indeed absence of cycles in the union of the await-dependency relations over input/output variables implies absence of cycles in the precedence constraints in the task graph of the product component.

Proposition 2.1 [Await Compatibility Implies Acyclic Product Task Graph] *Let C_1 and C_2 be compatible reactive components. Then the task graph over the set of tasks of C_1 and C_2 obtained by retaining the precedence edges in the individual components and adding cross-component edges from a task A_1 of one component to a task A_2 of another component whenever A_1 writes a variable read by A_2, is acyclic.*

Proof. Consider two compatible components, C_1 and C_2. Let $\prec_1$ and $\prec_2$ be their respective precedence relations over their task sets, and let $\succ_1$ and $\succ_2$ be the corresponding input/output await dependencies. Consider the combined task graph over the tasks of both the components obtained by retaining edges of the individual precedence relations and cross-component edges from a task A_1 of one component to a task A_2 of another component if A_2 reads a variable written by A_1. We will prove that if this task graph contains a cycle, then the union relation $(\succ_1 \cup \succ_2)$ over input and output variables also contains a cycle, thereby contradicting the assumption that the two components are compatible.

Consider a cycle in the combined task graph. This type of cycle must alternate between stretches of tasks, such that each stretch contains one or more tasks of a single component, and there is a cross-component edge from the last task of one stretch to the first task of the next stretch. Let $(A_1, B_1), (A_2, B_2), \ldots (A_k, B_k)$ be the pairs of tasks connected by cross-component edges in the order in which these cross-component edges appear in the cycle. For each j, tasks A_j and B_j belong to different components. Let x_j be the variable written by task A_j and read by task B_j. Thus, x_j must be an output variable of the component to which the task A_j belongs and an input variable of the other component to which the task B_j belongs. From each task B_j, there is a stretch of the cycle to the task A_{j+1} within the same component (define $A_{k+1} = A_1$ for the cycle to wrap around). Consider the case when the task B_j belongs to the component C_1. Then either $B_j = A_{j+1}$ or $B_j \prec_1^+ A_{j+1}$. The variable x_j is an input to C_1 belonging to the read-set of B_j, and the variable x_{j+1} is an output of C_1 belonging to the write-set of A_{j+1}. Thus, the component C_1 cannot produce the output variable x_{j+1} before the input variable x_j is available. By definition of await dependencies, $x_{j+1} \succ_1 x_j$. The case when the task B_j belongs to the component C_2 is symmetric and leads to $x_{j+1} \succ_2 x_j$. Note that in this argument, the task A_{k+1} is the same as A_1, and thus the variable x_{k+1} is the same as x_1. This gives a cycle of await dependencies, alternating between $\succ_1$ and $\succ_2$, among the sequence of input/output variables $x_1, x_2, \ldots x_k, x_1$.

Thus, we have established that the existence of a cycle in the task graph of the product implies the existence of a cycle in the combined await dependencies and, thus, incompatibility of the two components. ■

The following definition summarizes the parallel composition operation.

Component Composition

Let $C_1 = (I_1, O_1, S_1, \mathit{Init}_1, \mathit{React}_1)$ and $C_2 = (I_2, O_2, S_2, \mathit{Init}_2, \mathit{React}_2)$ be compatible synchronous reactive components. Suppose the reaction description React_1 is given using local variables L_1 by a task graph with the set $\mathcal{A}_1$ of tasks and the precedence relation $\prec_1$, and the reaction description React_2 is given using local variables L_2 by a task graph with the set $\mathcal{A}_2$ of tasks and the precedence relation $\prec_2$. Then the *parallel composition* $C_1 \| C_2$ is a synchronous reactive component C such that:

- the set S of state variables is $S_1 \cup S_2$;
- the set O of output variables is $O_1 \cup O_2$;
- the set I of input variables is $(I_1 \cup I_2) \setminus O$;
- the initialization for a state variable x is given by Init_1 for $x \in S_1$ and by Init_2 for $x \in S_2$; and
- the reaction description of C uses the local variables $L_1 \cup L_2$ and is given by the task graph such that (1) the set of tasks is $\mathcal{A}_1 \cup \mathcal{A}_2$, and (2) the precedence relation is the union of $\prec_1$ and $\prec_2$ and task pairs (A_1, A_2), such that A_1 and A_2 are tasks of different components with some variable occurring in both the write-set of A_1 and the read-set of A_2.

Properties of Parallel Composition

Let C_1 and C_2 be compatible components. Then by the above definition, the product $C_1 \| C_2$ is the same as the product $C_2 \| C_1$. Thus, the parallel composition operation is *commutative.*

The parallel composition is also associative. Suppose two components, C_1 and C_2, are compatible, and their product, $C_1 \| C_2$, is compatible with a third component, C_3. Then compatibility also holds for components C_2 and C_3 and for C_1 and $C_2 \| C_3$. Furthermore, $(C_1 \| C_2) \| C_3$ is the same as $C_1 \| (C_2 \| C_3)$. Thus, if we want to compose multiple components, then we can compose two, compose the result with a third one, and so on, and we get the same final result irrespective of the order of composition. At some step, we may discover incompatibility due to either common outputs or cyclic await dependencies, and we may not be able to compose all the components, but this failure does not depend on the order in which the components are composed.

If both the components C_1 and C_2 are finite-state, then so is the product $C_1 \| C_2$. If C_1 has n_1 states and C_2 has n_2 states, then $C_1 \| C_2$ has $n_1 * n_2$ states. For example, in the composition of the components `Delay1` and `Delay2`, each component has two states, and the product has four states. If we were to compose n instances of the component `Delay` in a chain to construct a component that outputs, in each round, the value of the input n rounds earlier, then it will have 2^n states. The fact that the number of states grows *exponentially* with the number of components is sometimes referred to as the *state-space explosion problem*, and it poses a challenge to analysis tools in terms of scalability.

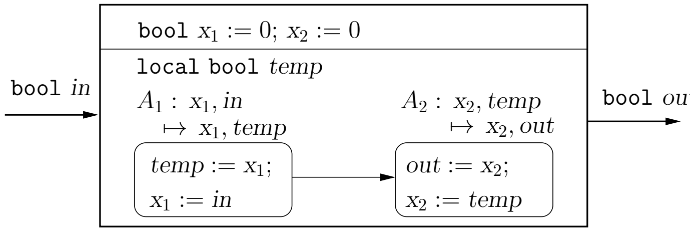
Note that when all the tasks of two compatible components C_1 and C_2 are deterministic, then the product $C_1 \| C_2$ is guaranteed to be deterministic. Similarly, if all the tasks of two compatible components C_1 and C_2 are input-enabled, then the product $C_1 \| C_2$ is guaranteed to be input-enabled.

2.3.4 Output Hiding

The final operation needed to define the semantics of block diagrams is hiding of output variables. If y is an output variable of a component C, then the result of *hiding* y in C, denoted $C \setminus y$, gives a component that behaves the same way as the component C, but y is no longer an output that is observable outside. This is achieved by removing y from the set of output variables and declaring y to be a local variable in the reaction description.

Let us revisit the component `Delay1` $\|$ `Delay2` (see figure 2.16). If we hide the intermediate output *temp*, then we get the desired product component `DoubleDelay`: the set of state variables is $\{x_1, x_2\}$, the set of output variables is $\{out\}$, and the set of input variables is $\{in\}$. The resulting component is shown in figure 2.23. Note that the initial state of the component `DoubleDelay` is $(0, 0)$, and its reactions are:

$$(0,0) \xrightarrow{0/0} (0,0); \quad (0,0) \xrightarrow{1/0} (1,0); \quad (0,1) \xrightarrow{0/1} (0,0); \quad (0,1) \xrightarrow{1/1} (1,0);$$
$$(1,0) \xrightarrow{0/0} (0,1); \quad (1,0) \xrightarrow{1/0} (1,1); \quad (1,1) \xrightarrow{0/1} (0,1); \quad (1,1) \xrightarrow{1/1} (1,1).$$



bool $x_1 := 0$; $x_2 := 0$
local bool $temp$
bool in
$A_1 : x_1, in \mapsto x_1, temp$
$A_2 : x_2, temp \mapsto x_2, out$
bool out
$temp := x_1$; $x_1 := in$
$out := x_2$; $x_2 := temp$

Figure 2.23: The Component `DoubleDelay`

Hiding preserves all the following properties of components: being finite-state, combinational, deterministic, input-enabled, and event-triggered.

When we want to hide multiple output variables, the order in which we apply the hiding operator does not matter. If x and y are two output variables of a component C, then the components $(C \setminus x) \setminus y$ and $(C \setminus y) \setminus x$ are exactly the same, and we can use $C \setminus \{x, y\}$ as an abbreviation to indicate hiding of both the output variables.

Exercise 2.15: Consider the component `ClockedDelay` from exercise 2.7. The component `ClockDelayComparator` is defined as follows:

$$(\,\texttt{Comparator}[out \mapsto x] \parallel \texttt{ClockedDelay}\,) \setminus x$$

Describe the input-output behavior of the component `ClockDelayComparator`. ■

Exercise 2.16: Consider the component `DoubleSplitDelay` defined as

$$(\texttt{SplitDelay}[\,out \mapsto temp\,] \parallel \texttt{SplitDelay}[\,in \mapsto temp\,]) \setminus temp$$

This component is similar to the component `DoubleDelay` except we use instances of the component `SplitDelay` instead of `Delay`. Show the "compiled" version of `DoubleSplitDelay`, that is, list its state, input, output, and local variables, tasks, and precedence constraints. What are the await dependencies among output and input variables for `DoubleSplitDelay`? ■

Exercise 2.17: Recall the event-triggered component `SecondToMinute` from exercise 2.6 with the input event variable *second* and the output event variable *minute* such that *minute* is present every 60^{th} time the event *second* is present. Now suppose we want to design an event-triggered component `SecondToHour` with an input event variable *second* and an output event variable *hour*, such that the output event *hour* is present every 3600^{th} time the event *second* is present. Show how to construct the desired component `SecondToHour` from the component `SecondToMinute` using the operations of parallel composition, instantiation, and output hiding. ■

2.4 Synchronous Designs

Before we consider some illustrative design problems in the synchronous model, let us recap the salient features and assumptions of the model.

In the classical functional model of computation, the component reads its input and then computes, producing the output on termination. The desired behavior of the component is described as a function from inputs to outputs. Reactive components, in contrast, interact with their environment via inputs and outputs in an ongoing manner. In principle, the component never terminates. The desired behavior is described by the *sequence* of outputs that the component should produce in response to a given sequence of inputs.

In *synchronous* reactive computation, the computation proceeds in a well-defined sequence of rounds. All the components, along with the environment that is supplying the inputs, agree on what constitutes a round. Event-triggered modeling can be used to describe the situation where a component may not be interested in every round and actively participates only in those rounds in which one of the trigger events is present. The key assumption of the synchronous model is that the computation of all the tasks within a round and all the inter-task communication necessary to determine the values of all the variables logically happens *instantaneously*. The external inputs do not change during a round, and when the inputs do change, a new round is initiated with all the tasks ready to process the new inputs. This assumption is called the *synchrony hypothesis*. This idealized assumption leads to simplicity and predictability of designs.

The computation of a component within a round can be split into multiple tasks. The precedence constraints among tasks capture read/write dependencies among its variables and lead to await dependencies among output and input variables. While composing components, absence of mutually cyclic await dependencies, a condition that can be checked at design time, ensures well-behaved execution of the product. During a round, the order in which tasks execute does not affect the resulting reaction. Nondeterminism, that is, multiple reactions in response to the same input, needs to be explicitly programmed within the description of a task and is not an artifact of the interaction model. In particular, for deterministic components, the behavior is repeatable: if we execute the component again with the same sequence of inputs, then we will observe the same sequence of outputs. This is valuable in debugging and analysis of complex designs.

During implementation, one needs to ensure that the implementation faithfully implements the synchronous semantics. This is the case, for instance, if the upper bound on the time needed to compute a reaction, which may require inter-component communication, is less than the minimum delay between changes to the input. Real-time scheduling theory, to be studied in chapter 8, offers ways of checking this.

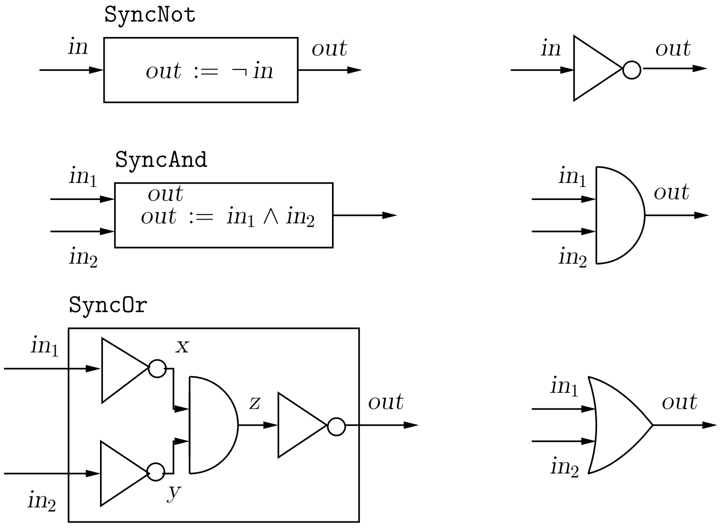


Figure 2.24: Synchronous Not, And, and Or Gates

2.4.1 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. The design of synchronous circuits offers an excellent case study of how to build complex systems by putting together simpler components in a hierarchical manner. We can construct synchronous circuits from three basic building blocks: as basic logic gates we use the Not and And gates, and as basic memory cell we use a latch component that models a set-reset flip-flop. These building blocks are then combined to obtain circuits by applying the three operations of parallel composition, variable renaming, and output hiding.

Combinational Circuits

Figure 2.24 defines three deterministic and combinational synchronous reactive components for modeling Not, And, and Or gates. In the description of synchronous circuits, all variables are implicitly assumed to be of type `bool`.

The component `SyncNot` is the same as the component `Inverter` of figure 2.17 and models a Not gate, which takes a Boolean input variable *in* and produces a Boolean output *out*. The reaction description sets the output to the logical negation of the input value. Note that the output *out* awaits the input *in*. The component `SyncAnd` models an And gate in a similar fashion. The component takes two Boolean input variables in_1 and in_2 and produces a Boolean output

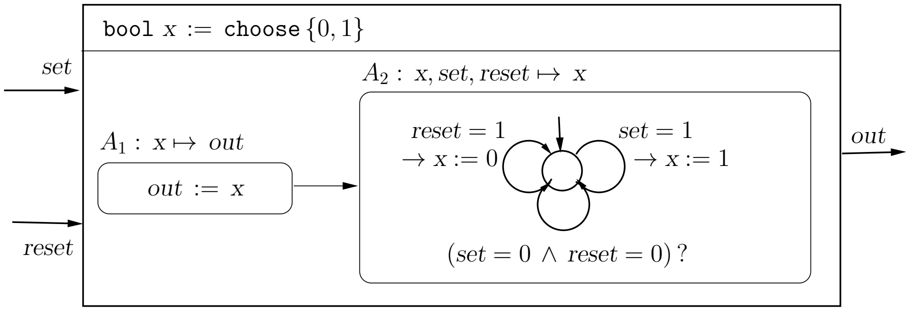


Figure 2.25: Synchronous `Latch` Component

out. The output is set to the logical conjunction of the two inputs and awaits both the input variables.

From `SyncNot` and `SyncAnd` gates we can build all combinational circuits. For example, by de Morgan's law, an Or gate can be defined by composing an And gate with three Not gates that negate both inputs and the output of the And gate. The block diagram is shown in figure 2.24. Note that instances of the components `SyncNot` and `SyncAnd` are shown by the corresponding symbolic representations commonly used in circuit diagrams. The resulting component `SyncOr` has two Boolean input variables in_1 and in_2 and produces a Boolean output *out*. The local variables x, y, and z of `SyncOr` represent internal wires that connect the four component gates. The component `SyncOr` is deterministic and combinational, and its output awaits both its input variables. The component `SyncOr` can be equivalently described using the operations of instantiation, parallel composition, and hiding:

$$\begin{aligned}
\texttt{SyncNot1} &= \texttt{SyncNot}[in \mapsto in_1][out \mapsto x],\\
\texttt{SyncNot2} &= \texttt{SyncNot}[in \mapsto in_2][out \mapsto y],\\
\texttt{SyncNot3} &= \texttt{SyncNot}[in \mapsto z],\\
\texttt{SyncAnd1} &= \texttt{SyncAnd}[in_1 \mapsto x][in_2 \mapsto y][out \mapsto z],\\
\texttt{SyncOr} &= (\texttt{SyncNot1} \,\|\, \texttt{SyncNot2} \,\|\, \texttt{SyncAnd1} \,\|\, \texttt{SyncNot3}) \setminus \{x, y, z\}.
\end{aligned}$$

Sequential Circuits

The combinational circuits are stateless. To model sequential circuits, we need a component that can store a value in its state from one round to the next. Figure 2.25 defines a nondeterministic component for modeling a unit-delay latch. The latch takes two Boolean input variables *set* and *reset* and produces a Boolean output *out*. The latch has a Boolean state, which is represented by the state variable x. The initial value of the state is unconstrained, and this is expressed using the `choose` construct in the initialization. In every round, the latch first issues its state as output and then waits for the input values

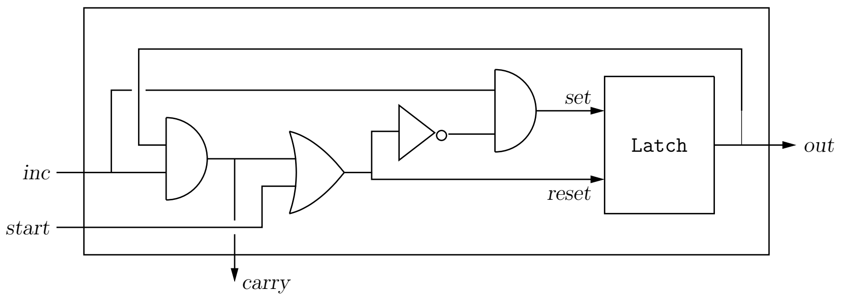


Figure 2.26: Synchronous Component `1BitCounter`

to compute its next state. For this purpose, the reaction description is split into two tasks: the task A_1 for computing the output *out* and the task A_2 for updating x. The update of the state variable x is described using a single-mode extended-state machine with three mode-switches. If the value of *set* is 1, then the latch can change its state to 1 using the mode-switch "$set = 1 \to x := 1$"; and if the value of *reset* is 1, then the latch can change its state to 0 using the mode-switch "$reset = 1 \to x := 0$". If both input variables have value 1, then the guards of both mode-switches are satisfied, and one of them is executed nondeterministically; thus, the next state of the latch may be either 0 or 1. If both input variables have value 0, then the state stays unchanged using the mode-switch "$set = 0 \wedge reset = 0$".

Note that the component `Latch` is nondeterministic and finite-state, and its output does not await either of its input variables. The fact that the output of the latch is available before the values of its inputs are known is essential for composing latches with logic gates, which in every round (clock cycle) provide the latch inputs dependent on the latch outputs.

Binary Counter

As an example of a sequential circuit, we design a three-bit binary counter. The counter has two Boolean input variables *start* and *inc*, for starting and incrementing the counter, respectively. 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. When the input *start* is 1, the counter value is reset to 0 independent of the value of the other input *inc*. Otherwise when the input *inc* is 1, the counter value increases by 1. If the counter value is 7, then an increment changes the counter value to 0. In every round, the counter issues its value as output—the low bit on the output variable out_0, the middle bit on the output variable out_1, and the high bit on the output variable out_2.

Figure 2.26 shows a possible design of a one-bit counter. It uses one `Latch` component to store one bit of state, and its logic is implemented using two And

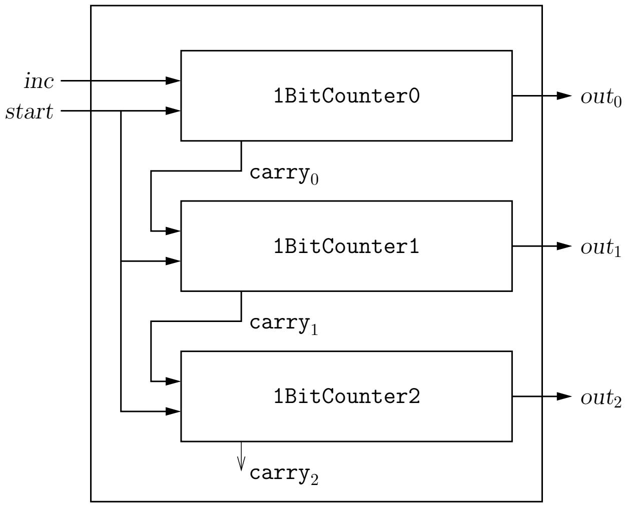


Figure 2.27: Synchronous Component `3BitCounter`

gates, one Not gate, and one Or gate. Verify that there are no awaits cycles and, thus, the components are compatible. The value of the output *out* equals the state of `Latch` at the beginning of the round. Let us consider all possible cases to understand how this circuit works.

Suppose the state of the latch (that is, the value of the counter) is 0. Then the output *carry*, which indicates overflow in the counter value, is 0. The value of the local variable *reset* equals the input *start*, and the value of the local variable *set* equals the conjunction $inc \wedge \neg\, start$. Observe that both *set* and *reset* cannot be 1 simultaneously. When *start* is 1, only *reset* is 1, and in this case, the latch state is reset to 0. When *start* is 0, *set* equals *inc*: if *inc* is 1, then the latch state is updated to 1; otherwise, the latch state stays 0.

Suppose the state of the latch is 1 and the input *inc* is 0. Then again, the value of the output *carry* is 0. The value of the local variable *reset* equals the input *start*, and the value of the local variable *set* equals 0. When *start* is 1, the latch state is reset to 0; otherwise, the latch state stays 1.

Finally, suppose the state of the latch is 1 and the input *inc* is 1. In this case, the value of the output *carry* is 1, indicating overflow. The variable *reset* equals *start* and the variable *set* equals $\neg\, start$. Thus, no matter what the value of *start* is, the latch state is updated to 0.

Figure 2.27 shows the block diagram for connecting three instances of the 1-bit counter to implement a 3-bit counter in a natural way. The input variable *start* acts as the command to reset to all the three instances. The input variable *inc* acts as the command to increment only to the 1-bit counter `1BitCounter0` corresponding to the least significant bit. The *carry* output of `1BitCounter0` is used as the command to increment the next significant bit stored in `1BitCounter1`, whose carry overflow output acts as the command to increment the most significant bit.

Exercise 2.18: An XOR (Exclusive-Or) gate has two Boolean inputs in_1 and in_2, and a boolean output *out*. The output is 1 when exactly one of its two inputs are 1 and is 0 otherwise. Define the combinational component `SyncXor` to capture this desired functionality by composing And, Or, and Not gates. ■

Exercise 2.19: A *parity* circuit has n boolean input variables $in_1, in_2, \ldots in_n$ and a boolean output *out*. The value of the output should be 1 if an odd number of input variables have the value 1 and should be 0 otherwise. Construct the component $\texttt{Parity}_n$ that computes the parity of n input variables by composing instances of the component `SyncXor` defined in exercise 2.18. ■

Exercise 2.20: Design a 1-bit synchronous adder `1BitAdder` by composing instances of And, Or, Not, and Xor gates. The component `1BitAdder` has three input variables x, y, and *carry-in* and two output variables z and *carry-out*. In each round, the value encoded by the two output bits z and *carry-out*, where z is the least significant bit, should equal the sum of the values of three input variables. Then, design a 3-bit synchronous adder `3BitAdder` by composing three instances of the component `1BitAdder`. The component `3BitAdder` has input variables x_0, x_1, x_2, y_0, y_1, y_2, and *carry-in* and has output variables z_0, z_1, z_2, and *carry-out*. In each round, the 4-bit number encoded by the output variables z_0, z_1, z_2, and *carry-out* should equal the sum of the 3-bit number encoded by the input variables x_0, x_1, and x_2, the 3-bit number encoded by the input variables y_0, y_1, and y_2, and the input value of *carry-in*. ■

2.4.2 Cruise Control System

We will illustrate concepts of top-down component-based design using a simplified design of a cruise-control system for a car.

Top-Level Specification

The inputs and outputs of the system are shown in figure 2.28. The driver interacts with the cruise-controller with three buttons: one to turn the cruise controller on and off, one to increment the desired speed, and one to decrement the desired speed. These are modeled by three input event variables *cruise*, *inc*, and *dec*. Presence of the event *cruise* should toggle the controller between on and off modes. When it is turned on, the desired cruising speed should be set to the current speed, and the events *inc* and *dec*, when present, should cause the desired cruising speed to increment and decrement, respectively. We should ensure that this value stays within a reasonable cruising range, given by a minimum value, denoted `minSpeed`, and a maximum value, denoted `maxSpeed`.

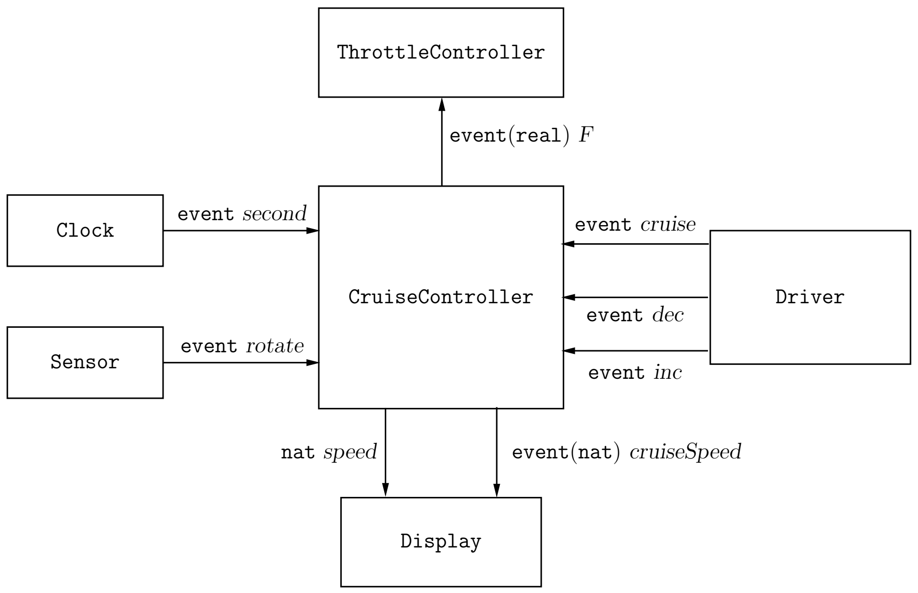


Figure 2.28: Inputs and Outputs of the Cruise-Control System

The cruise controller needs to measure the current speed to make its decisions. This is achieved using two input events: *rotate* and *second*. Whenever the wheel completes a rotation, a sensor associated with the wheel-shaft issues the input event *rotate*, and every second a system-wide clock issues the input event *second*. Thus, the controller can count the number of rotations every second and compute the current speed.

The controller should send information to the display regarding the current settings. This is modeled by output variables *speed*, denoting the current speed, and *cruiseSpeed*. The value of *cruiseSpeed* is absent if the cruise control is turned off and, when on, equals the current cruising speed set by the driver.

Finally, the output F is sent to the throttle control system and corresponds to the force needed to adjust the throttle to regulate the current speed so as to track the desired cruising speed.

Decomposing into Subsystems

As a next step in the design, we decompose the controller into three subsystems: the component `MeasureSpeed` to compute the current speed based on the inputs *rotate* and *second*, the component `SetSpeed` to keep track of the desired cruise settings based on the inputs from the driver and the current speed, and the component `ControlSpeed` to process the differential between the current speed and the desired speed in order to compute the output force. The interconnections among these subcomponents are shown in figure 2.29. The design of the

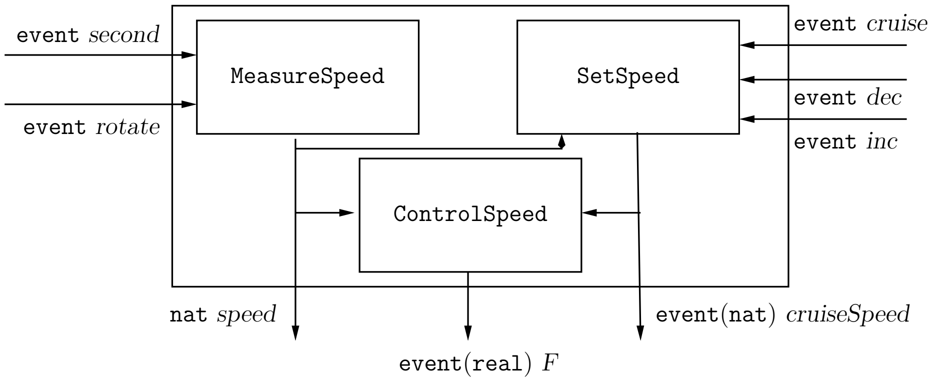


Figure 2.29: Components of the Cruise-Control System `CruiseController`

component `ControlSpeed` requires understanding the dynamics of the car and control theory, a topic to be discussed in chapter 6 on dynamical systems. We proceed to design the other two components.

Tracking Speed

The task for the component `MeasureSpeed` is to output the current speed of the car based on the two input event variables, *rotate* and *second*. The component is shown in figure 2.30. The component has a state variable *count* that counts the number of times the event *rotate* has occurred since the most recent occurrence of the event *second*. The initial value of *count* is 0. The state variable *s* remembers the current speed: it is 0 initially, and in every round during which the event *second* is present, the current value of *count* is used to update *s*.

More precisely, the rules for updating the state are as follows. If *rotate* event is present, then the component increments *count*. If *second* event is present, then the current value of *count* indicates the number of rotations of the wheel during a time interval of a second. To compute the speed, this value is multiplied by a constant, denoted k, which depends on the circumference of the wheel and rounded to the nearest integer (the function `round-off` returns the integer nearest to its argument). In this case, the value of *count* is reset to 0. Note that if both input events are absent, then the state stays unchanged. If both events are present, then *count* is first incremented, then used to compute the value of *s*, and then reset to 0.

The component has an output variable *speed*: in each round, the output is set to the updated value of *s*. Thus, it is a latched output. The display as well as the speed controller can access this output in any round. The component `MeasureSpeed` is deterministic and event-triggered.

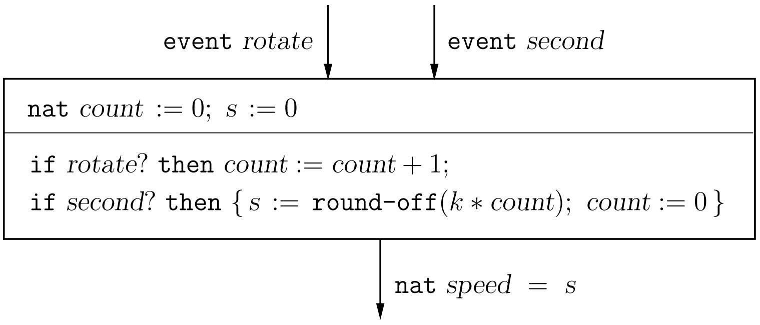

event $rotate$
event $second$
nat $count := 0;\ s := 0$
if $rotate$? then $count := count + 1$;
if $second$? then $\{\, s :=$ round-off$(k * count);\ count := 0 \,\}$
nat $speed = s$

Figure 2.30: Component `MeasureSpeed`

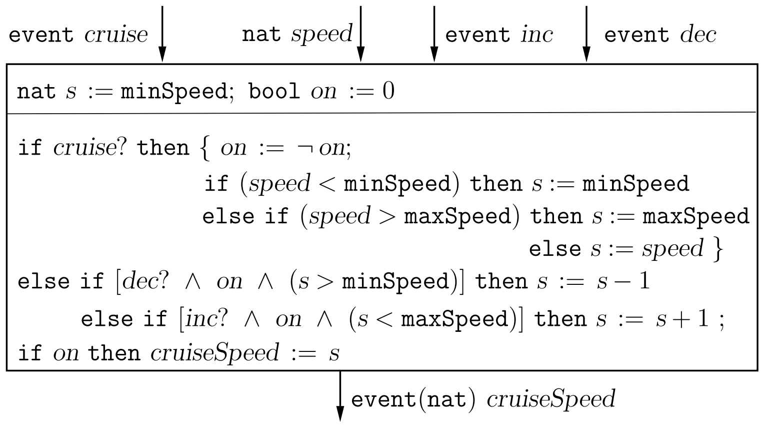
Tracking Cruise Settings

Now consider the component `SetSpeed` shown in figure 2.31. The output variable *cruiseSpeed* is an event variable that is either absent (when the controller is off) or indicates the current desired speed. The component maintains two state variables: a Boolean variable *on* that keeps track of whether the controller is switched on and the current desired speed *s*.

Since there are three input events for the component, each of which can be present or absent, we should process all their combinations. In our design, we avoid this blow-up by considering these events in a priority order: first *cruise*, then *dec*, and then *inc*. If the driver presses two or more buttons simultaneously, then the effect will be equivalent to pressing a single button, with the highest priority among those pressed simultaneously. Alternatively, we could make an assumption about the environment that at any instant, at most one of the input events can be present.

The component updates the state variable *on* according to the following rule: every time the event *cruise* occurs, the variable *on* is toggled. The rule for updating the desired speed *s* is as follows. If the event *cruise* is present, then the variable *s* is set to the current speed, provided the current speed is within the legal cruising range from `minSpeed` to `maxSpeed`. Otherwise, if the event *dec* is present, then the variable *s* is decremented, provided its value is above the minimum threshold and the controller is on. Finally, if the event *inc* is present, then the variable *s* is incremented, provided its value is below the maximum threshold and the controller is on. If none of the rules applies, then the desired speed stays unchanged. After updating the state, the component decides on its output based on the following rule: if the updated value of *on* is 1, then the output *cruiseSpeed* is set to the updated value of *s* or else it is absent.

Note that the component `SetSpeed` is deterministic and event-triggered. Its output variable awaits all the four input variables.



```
nat s := minSpeed; bool on := 0

if cruise? then { on := ¬ on;
                  if (speed < minSpeed) then s := minSpeed
                  else if (speed > maxSpeed) then s := maxSpeed
                                              else s := speed }
else if [dec? ∧ on ∧ (s > minSpeed)] then s := s − 1
    else if [inc? ∧ on ∧ (s < maxSpeed)] then s := s + 1 ;
if on then cruiseSpeed := s
```

Figure 2.31: Component `SetSpeed`

Exercise 2.21 : Consider the design of the component `SetSpeed` of figure 2.31. Suppose we want to add another input control for the driver, *pause*, with the following desired behavior. When the cruise controller is on, if the driver presses *pause*, then the controller is temporarily turned off. In the resulting paused state, the output *cruiseSpeed* should be absent, and the events *inc* and *dec* should be ignored. Pressing *pause* again in this paused state should resume the operation of the cruise controller, restoring the desired speed on pausing. Pressing *cruise* in the paused state should switch the system off, and when the controller is off, pressing *pause* should have no effect. Redesign the component `SetSpeed` with this additional input event *pause* to capture the above specification. ■

2.4.3 Synchronous Networks *

In a synchronous network, communication happens in a sequence of time slots. The network topology determines the one-hop directed connectivity among network nodes. In each time slot, a node sends a message to all its neighbors connected by outgoing edges and receives messages from all its neighbors connected by incoming edges. We can model such networks as synchronous reactive components.

Modeling a Network Node

The design of an individual node should be independent of the network topology so that instances of the node can be connected in different ways to form different networks. For this purpose, each network node is modeled as a component `NetwkNode` with an input variable *in* and an output variable *out* as shown in figure 2.32. If the type of messages that a node sends in each round is `msg`, then

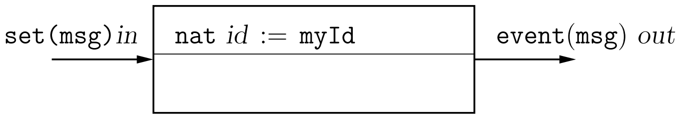


Figure 2.32: Schematic of a Synchronous Network Node `NetwkNode`

the type of the output variable *out* is `event`(`msg`) since in each round, a node may or may not send a message. The type of the input variable *in* is `set(msg)`, and a value of this type is a set of messages of type `msg`. We want to design the component so that there is no await dependency between the output *out* and the input *in*: during each round, the component decides on its output message based on its state and then updates the state in response to the input that contains the set of messages it receives.

The description of the component `NetwkNode` is parameterized by an identifier `myId`. To form a desired network of components, we create as many instances of the component `NetwkNode` as needed. Each instance is given a unique identifier, which is used to instantiate `myId` and rename the input and output variables to avoid name conflicts.

Modeling the Interconnections

The communication network is modeled as a combinational component `Network`. It has one input and one output variable for each instance of `NetwkNode`.

As a concrete example, consider the communication network shown in figure 2.33 over four nodes with identifiers 1, 3, 5, and 8. The edges show connectivity: for example, node 3 has two outgoing edges connecting it to nodes 5 and 8 and two incoming edges connecting from nodes 1 and 5. In a single round, if node 3 chooses to send a message, then it will be delivered to both nodes 5 and 8, and the set of messages it receives contains messages sent by nodes 1 and 5 in this round.

Figure 2.34 shows the composition of components. There are four instances of the component `NetwkNode` corresponding to the four nodes. The network is captured by `Network` with input variables out_1, out_3, out_5, and out_8, each of type `event`(`msg`), each of which is connected to the output of the corresponding node component. It has output variables in_1, in_3, in_5, and in_8, each of type `set(msg)`, connected to the input of the corresponding node component. In each round, the network reads the messages from all its input variables out_n, and for each node n, it collects the messages that are present on the incoming links for node n and delivers the corresponding set of messages by updating the output variable in_n. The reaction description in figure 2.34 first sets all the output sets in_n to empty sets. Then it checks all the input events one by one, adding it to the appropriate output sets. For instance, if the input message

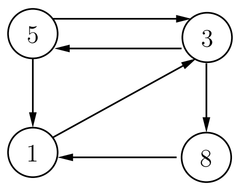


Figure 2.33: Example Communication Network with Four Nodes

out_3 is present, since the node 3 has outgoing links to nodes 5 and 8, then the message out_3 is added to the output sets in_5 and in_8.

More generally, let P be a set of node identifiers and let $E \subset P \times P$ denote the directed one-hop connectivity edges among nodes. Then for each $n \in P$, let $\texttt{NetwkNode}_n$ be the instance of `NetwkNode` obtained by instantiating `myID` to n and renaming each input and output variable x to x_n. The component $\texttt{Network}_{P,E}$ is a deterministic combinational component with the set $\{out_n \mid n \in P\}$ of input variables and the set $\{in_n \mid n \in P\}$ of output variables. In each round, for each $n \in P$, the output variable in_n equals the set containing the input values out_m, such that (1) the set E of network connections has an edge from node m to node n, and (2) the event out_m is present. The desired system is the parallel composition of all the components $\texttt{NetwkNode}_n$, for $n \in P$, and the interconnection network component $\texttt{Network}_{P,E}$.

Leader Election

To illustrate the design of algorithms for synchronous networks, let us consider the classical coordination problem of *leader election*: the nodes should exchange messages to decide on a unique leader. More precisely, let us assume that each node component has an output variable *status* that ranges over the enumerated type {`unknown`, `leader`, `follower`}. The nodes exchange messages updating the *status* so that (1) eventually every component sets the *status* output to be either `leader` or `follower`, and (2) exactly one component changes the *status* output to the value `leader`.

Since each node has a unique identifier, it is natural to use these identifiers for choosing the leader, say, the one with the highest value of the identifier. At the beginning, a node does not know which other nodes are part of the network, and the purpose of exchanging messages is to identify this highest identifier. We want the algorithm to work in as many networks as possible. Consider the algorithm shown in figure 2.35, which relies on two assumptions:

1. The network is *strongly connected*: for every pair of nodes m and n, there is a directed path from node m to node n.

2. Each node knows an upper bound N on the total number of nodes in the network.

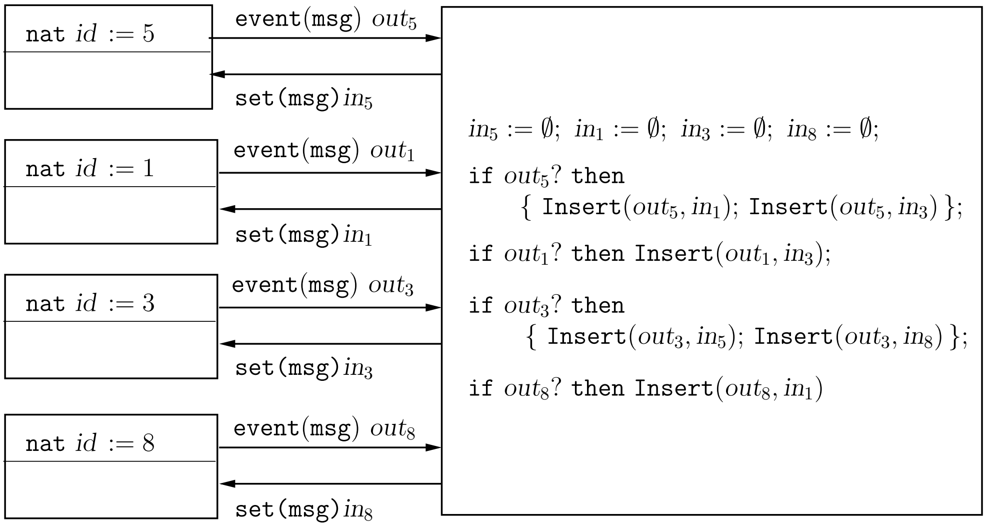

nat id := 5
event(msg) out_5
set(msg) in_5
nat id := 1
event(msg) out_1
set(msg) in_1
nat id := 3
event(msg) out_3
set(msg) in_3
nat id := 8
event(msg) out_8
set(msg) in_8
$in_5 := \emptyset$; $in_1 := \emptyset$; $in_3 := \emptyset$; $in_8 := \emptyset$;
if out_5? then { Insert(out_5, in_1); Insert(out_5, in_3) };
if out_1? then Insert(out_1, in_3);
if out_3? then { Insert(out_3, in_5); Insert(out_3, in_8) };
if out_8? then Insert(out_8, in_1)

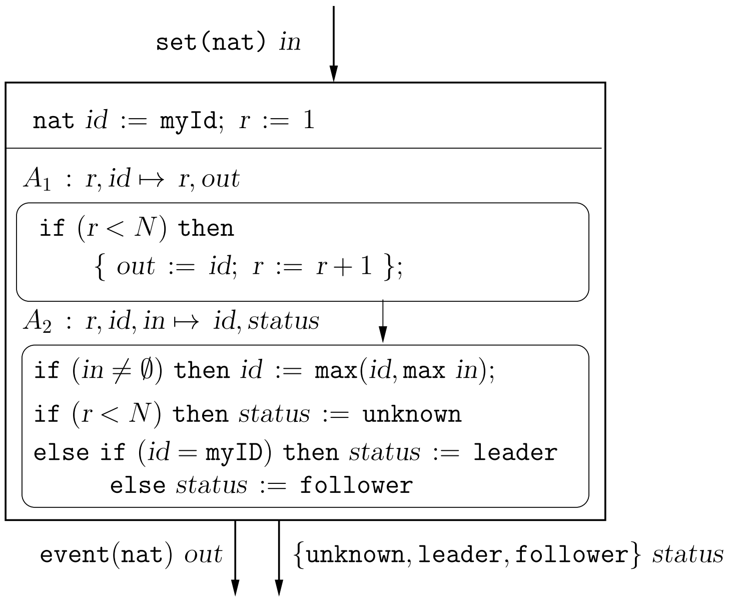
Figure 2.34: The Synchronous Network Component `Network`

The first condition is needed for information to flow from one node to another, and the second is used for termination.

In the algorithm of figure 2.35, called the *flooding* algorithm, a node maintains a state variable *id* that equals the highest identifier it knows so far. Initially, the value of *id* equals the node's own unique identifier. In each round, the node outputs this identifier to its neighbors, and if it receives any identifier higher than the current value of *id*, it updates this value. The reaction description is split into two tasks: the task A_1 computes the output *out* and updates the state variable *r*, and the task A_2 updates the state variable *id* and computes the output *status*. Note that the first task does not need the input, and thus only the output variable *status* awaits the input variable *in*.

If the total number of nodes in a strongly connected network is N, then between each pair of nodes, there is a path with at most $N-1$ hops. Hence, after $N-1$ rounds, each node can be sure that its identifier has had a chance to propagate to every other node. More precisely, if a node's unique identifier is n, and if the shortest path from this node to another node m is of length j, then after j rounds, the value of *id* variable of node m will be n or higher. As a result, after $N-1$ rounds, the value of *id* variable of each node will be equal to the highest identifier in the network. At this point, each node can decide: if the value of its *id* variable equals its original identifier, then it is the leader, otherwise it is a follower.

Consider the four-node network shown in figure 2.34 so that each component is the instantiated version of the leader election component `SyncLENode` of fig-



```
nat id := myId; r := 1

A_1 : r, id ↦ r, out
    if (r < N) then
        { out := id; r := r + 1 };

A_2 : r, id, in ↦ id, status
    if (in ≠ ∅) then id := max(id, max in);
    if (r < N) then status := unknown
    else if (id = myID) then status := leader
        else status := follower
```

Figure 2.35: Component `SyncLENode` for Synchronous Leader Election

ure 2.35. Here is how their executions proceed:

1. In round 1, each of the nodes 1, 3, 5, and 8 output their original identifiers. The node 1 receives $\{5, 8\}$ and updates its *id* variable to 8; the node 3 receives $\{1, 5\}$ and updates its *id* variable to 5; the node 5 receives $\{3\}$, and its *id* variable remains 5; the node 8 receives $\{3\}$, and its *id* variable remains 8. All the nodes set the output *status* to the value `unknown`.

2. In round 2, nodes 1 and 8 output 8, and nodes 3 and 5 output 5. As a result, the *id* variable of node 3 gets updated to 8, and other nodes do not change their respective *id* variables. Again, all the nodes set the output *status* to the value `unknown`.

3. In round 3, nodes 1, 3, and 8 output 8, and node 5 outputs 5. The node 5 updates its *id* variable to 8. The updated value of the round-counting variable r equals $N = 4$; as a result, all the nodes decide based on the updated values of their respective *id* variables: the node 8 sets its output *status* to the value `leader`, and the rest set their output *status* to the value `follower`.

Observe that if the *diameter* of the network is D, that is, between every pair of nodes there is a path of length D or less, then after D rounds, the value of *id* variable of each node will be equal to the highest identifier in the network. An upper bound on D is $N - 1$, but D can be much less than this upper bound.

If the diameter D is known to the nodes in advance, then a node can decide at the end of round D.

Exercise 2.22: Consider the leader election algorithm in synchronous networks (figure 2.35). Argue that if the value of *id* does not change in a given round, then there is no need to send it in the following round (that is, the output *out* can be absent in the next round). This can reduce the number of messages sent. Modify the description of the component `SyncLENode` to implement this change. ■

Exercise 2.23*: In a strongly connected network, for each network node n, let D_n be the smallest integer j such that for every node m, there is a directed path of at most j links from m to n. For example, if the network is a complete graph (for every pair of nodes m and n, there is a link from m to n), D_n is 1 for every node n; if the network is unidirectional ring connecting all nodes in a single cycle, D_n is $N - 1$ for every node n, where N is the total number nodes; and in the network of figure 2.33, $D_5 = 3$, $D_3 = 2$, $D_1 = 2$, and $D_8 = 2$. Design an algorithm for synchronous networks so that each node n can figure out the value of D_n. As in the case of leader election, assume that each node has a unique identifier, and it knows the bound N on the total number of nodes. The algorithm should work for any network as long as it is strongly connected. Explain how your algorithm works. ■

Bibliographic Notes

The term *reactive computation*, as opposed to the classical functional computation, was introduced in [HP85]. Since the 1980s, a number of formal models of synchronous reactive computation have been introduced and studied. Prominent examples include Esterel [BG88], Lustre [CPHP87], and Statecharts [Har87]. All of these have resulted in industrial-strength programming environments; see [BCE$^+$03] for a survey of synchronous languages.

In theory of concurrency, a rich variety of formal models of reactive and concurrent computation with alternative forms of interaction among components has been studied. Example formalisms include CSP [Hoa85], CCS [Mil89], UNITY [CM88], data-flow networks [Kah74, LP95], I/O automata [Lyn96], TLA [Lam02], and BIP [Sif13].

The model of synchronous reactive components studied in this chapter is a simplified version of *Reactive Modules* [AH99b]. The presentation of the model follows the outline in [AH99a]. The description of the leader election algorithm is based on [Lyn96], which contains rigorous descriptions of algorithms for a large variety of distributed coordination problems in synchronous networks.