

Chapter 1

Synchronous Models

1.1 Reactive Components

A reactive component interacts with other components via inputs and outputs, executing in a sequence of rounds. The component maintains an internal state. In each round, it reads its inputs, and based on its current state and the inputs, it computes to produce outputs and to update the internal state.

As a first example, consider the **Delay** component shown in Figure 1.1. The component has a Boolean input variable *in*, a Boolean output variable *out*, and an internal state, modeled by a Boolean variable *x*. To explain the behavior of the component, we first need to specify the initial value 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 *out* to the value of the state *x* at the beginning of the round, and then updates the state to the current input *in*. 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. To explain the various aspects of the definition of this component precisely, we need a bit of mathematical notation concerning variables, expressions over variables, and assignment of values to variables.

1.1.1 Variables, Valuations, and Expressions

We use *typed variables* to describe components. The commonly used types are:

- **nat** denoting the set of natural numbers;
- **int** denoting the set of integers;
- **real** denoting the set of real numbers;
- **bool** denoting the Boolean valued type $\{0, 1\}$;
- enumerated types such as $\{on, off\}$ that contain a finite number of symbolic constants;

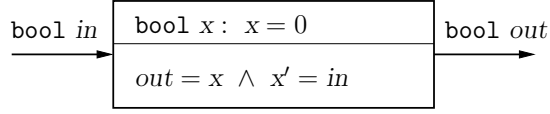


Figure 1.1: Reactive component Delay

- **event** denoting the enumerated type $\{\top, \perp\}$, where \top denotes that the event is present and \perp denotes that the event is absent. More generally, we allow the event type to be parameterized by another type, and the event can be absent, or can be present with a value of the parameter type. For example, the type `event(bool)` has 3 values 0, 1, and \perp .

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 an element of the type of v . We will use Q_V to denote the set of all valuations over V .

Typed expressions are formed from variables using operators. An *expression* e over a set V of variables consists of variables in V , constants, and primitive operations in types corresponding to these variables. Given a valuation q over V and an expression e over V , $q(e)$ denotes the value obtained by evaluating the expression according to the assignment of values to variables specified by q . For a Boolean expression e over variables V and a valuation q over V , we say that q *satisfies* e if $q(e) = 1$.

Over numerical types, we will use arithmetic operations such as addition and multiplication, and comparison operations such as equality and ordering. For Boolean expressions, we use logical operators such as conjunction \wedge , disjunction \vee , negation \neg , and implication \rightarrow . We also use *existential quantification*: if e is a Boolean expression over a set V of variables, and x is a variable in V , then $\exists x. e$ represents the Boolean expression over $V \setminus \{x\}$. A valuation q over $V \setminus \{x\}$ satisfies the quantified expression $\exists x. e$ if q can be extended by assigning some value to x to satisfy e , that is, there exists a valuation s over V such that $s(e) = 1$ and $s(y) = q(y)$ for each variable y in $V \setminus \{x\}$. If e_1 and e_2 are expressions of the same type, and b is a Boolean expression, then the *if-then-else* expression $b \rightarrow e_1 \mid e_2$ denotes an expression of the same type as e_1 and e_2 : for a valuation q , $q(b \rightarrow e_1 \mid e_2)$ equals $q(e_1)$ if $q(b) = 1$ and equals $q(e_2)$ otherwise. For an event variable x , the Boolean expression $x?$, meaning x is present, stands for $x \neq \perp$. Given an expression e over variables V and a variable x in V , if f is another expression of the same type as that of x , then we use $e[x \mapsto f]$ to denote the expression obtained from e by *substituting* every (free) occurrence of x by f .

For Boolean expressions, let us recall some standard concepts.

- **Satisfiability:** A Boolean expression e over variables V is satisfiable if at least one valuation q over V satisfies e .

- **Validity:** A Boolean expression e over variables V is valid if every valuation q over V satisfies e .
- **Equivalence:** Two expressions e_1 and e_2 over variables V are equivalent if for every valuation q over V , $q(e_1) = q(e_2)$.

The following operations over valuations will be used in our definitions:

- **Disjoint union:** Given two disjoint sets V and W of variables, a valuation s over V , and a valuation t over W , the combined valuation $[s, t]$ denotes the valuation over the union $V \cup W$ that assigns $s(v)$ to each variable $v \in V$ and $t(w)$ to each variable $w \in W$.
- **Restriction:** If q is a valuation over variables V , and $W \subseteq V$ is a subset of V , then $q(W)$ denotes the valuation over W obtained by restricting the domain of q to W .

One final bit of notation concerns primed variables. For a variable v , we use v' to denote a *primed* version of v with the same type as v . We will use this to distinguish the value of a state variable at the end of a round from its value at the beginning of the round. This carries over to sets of variables and valuations in a natural way: for a set V of variables, V' denotes the set of variables $\{v' \mid v \in V\}$; and for a valuation q over V , q' denotes the valuation over V' such that for every variable $v \in V$, $q'(v') = q(v)$.

1.1.2 Inputs, Outputs, and States

The component `Delay` 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 these 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 examples, we draw components as rectangular boxes. For each input variable, there is an incoming edge, and for each output variable there is an outgoing edge. These edges are labeled with the names and the 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, $Q_I = Q_O = Q_S = \{0, 1\}$.

1.1.3 Initialization and Update

To specify the dynamics of the component we must specify the initial states and how the component reacts to a given input in each state. In practice, a variety of programming styles are used to specify this, ranging from imperative style (for instance, SYSTEMC and ESTEREL), declarative equational style (for instance, LUSTRE), and hierarchical state machines (for instance, STATEFLOW). We will use Boolean expressions to specify the dynamics. This has the advantage of simplicity and clear mathematical semantics, and serves our purpose of explaining analysis techniques for reactive systems.

Initialization is expressed by a Boolean expression $Init$ over the state variables S . The set of initial states contains the states satisfying this expression, that is, a state $q \in Q_S$ is an initial state of the component if q satisfies $Init$. We require that there is at least one initial state: the expression $Init$ must be satisfiable. In general, this set can contain more than one state, and this allows modeling of situations where initial conditions are only partially known.

The initialization formula for `Delay` is $x = 0$, and the set of initial states is $\{0\}$.

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

If the component in state s , on 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*, and is an element of $Q_S \times Q_I \times Q_O \times Q_S$. We specify the reactions using a Boolean expression over state variables S , input variables I , output variables O , and primed state variables S' denoting the state at the end of the round. Thus, the dynamics is specified by a Boolean expression $React$ over variables $S \cup I \cup O \cup S'$ such that $s \xrightarrow{i/o} t$ is a reaction of the component precisely when the combined valuation $[s, i, o, t']$ satisfies the expression $React$.

For the `Delay` component, the reaction expression is

$$(out = x) \wedge (x' = in).$$

That is, in state s , on input i , the component outputs s and updates its state to i . In this case, $React$ has 4 possible reactions:

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

In our examples, the reaction expression is specified in the lower half of the component box.

We summarize the definition of a reactive component below:

SYNCHRONOUS REACTIVE COMPONENT

A *synchronous reactive component* C has a finite set I of typed input variables, a finite set O of typed output variables, a finite set S of typed state variables, such that these three sets are pair-wise disjoint, a satisfiable Boolean expression $Init$ over S , and a Boolean expression $React$ over $S \cup I \cup O \cup S'$. An *input* $i \in Q_I$ of C is a valuation over I ; an *output* $o \in Q_O$ of C is a valuation over O ; and a *state* $s \in Q_S$ of C is a valuation over S . A state $s \in Q_S$ is an *initial state* of C if s satisfies $Init$, and for states s, t , input i , and output o , $s \xrightarrow{i/o} t$ is a *reaction* of C if $[s, i, o, t']$ satisfies $React$.

1.1.4 Executions

The operational semantics of the component can be captured by defining its executions. An execution starts in an initial state, and proceeds for a finite number of rounds. It records the input, output, and updated state in each round.

COMPONENT EXECUTION

An *execution* of a synchronous reactive component C consists of a finite sequence of the form

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

where

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

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

1.1.5 Exercises

1. Consider a modified version of the **Delay** component, called **OddDelay**, with a Boolean input variable in , a Boolean output variable out , two Boolean state variables x and y , with the initialization expression

$$x = 0 \wedge y = 0,$$

and the reaction expression

$$out = (y \rightarrow x \mid 0) \wedge x' = in \wedge y' = \neg y.$$

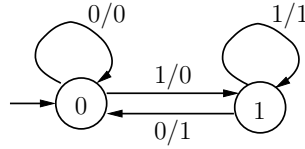


Figure 1.2: Mealy machine corresponding to the component `Delay`

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

- We want to design a reactive component with three Boolean input variables x , y , and $reset$, and a Boolean output variable z . The desired specification is the following. By default the output z is low (0). The component waits till 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 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. 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. Try to use as few states as possible. List all the possible reactions of the component.

1.2 Properties of Components

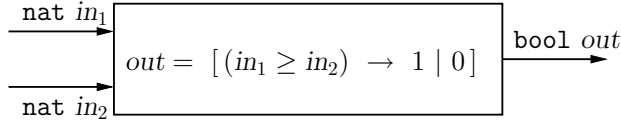
1.2.1 Finite-state Components

In many embedded applications, it suffices to consider types with only finitely many values. The Boolean type and enumerated types are finitely valued. In this case, 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 component `Delay`. Such components are called *finite-state components*, and are amenable to powerful 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.

For finite-state components, the 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, 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.

Figure 1.3: Combinational component `Comparator`

The Mealy machine representation of the `Delay` component is shown in Figure 1.2.

1.2.2 Combinational Components

Consider the `Comparator` component shown in Figure 1.3. The component has two input variables, in_1 and in_2 , both of which are of type of natural numbers, and has a Boolean output variable out . In each round, the component reads the inputs in_1 and in_2 , and sets the output out to 1 if the value of in_1 is greater than or equal to the value of in_2 , and 0 otherwise.

The component does not need to maintain any internal state, and hence, has no state variables. When S 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, and the initialization expression $Init$ implicitly is the constant 1. When there are no state variables, we will only specify the reaction expression.

The reaction expression for `Comparator` is shown in Figure 1.3. For every pair of natural numbers $m, n \in \mathbf{nat}$, if $m \geq n$, `Comparator` has a reaction $s_\emptyset \xrightarrow{(m,n)/1} s_\emptyset$, and if $m < n$, it has a reaction $s_\emptyset \xrightarrow{(m,n)/0} s_\emptyset$. A possible execution of the component is

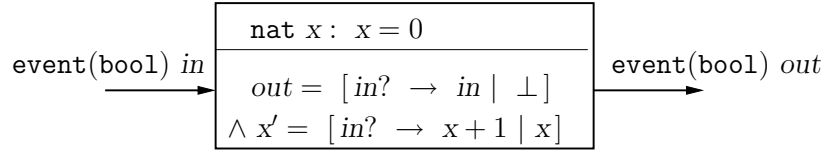
$$s_\emptyset \xrightarrow{(2,3)/0} s_\emptyset \xrightarrow{(5,1)/1} s_\emptyset \xrightarrow{(40,40)/1} s_\emptyset$$

Components such as `Comparator` without any internal state 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 expressions can be modeled as combinational components, and the component `Comparator` corresponds to the Boolean expression $in_1 \geq in_2$. 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 is 1 precisely when both input variables are 1. Whether a desired expression is modeled as a combinational component is a design choice, influenced by the primitives supported by the language used to specify the reactions.

Figure 1.4: Event-triggered component `TriggeredCopy`

1.2.3 Event-triggered Components

For execution 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. For this purpose, we use input variables of type `event`.

As an example, consider the component `TriggeredCopy` that copies its input to output, see Figure 1.4. 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 output, thus, is also `event(bool)`. The component `TriggeredCopy` does maintain a state variable `x`: initially `x` is 0, and whenever the input is present, `x` is incremented. Thus, the value of `x` shows the number of past rounds in which the input was present. For every natural number n , the component has 3 reactions:

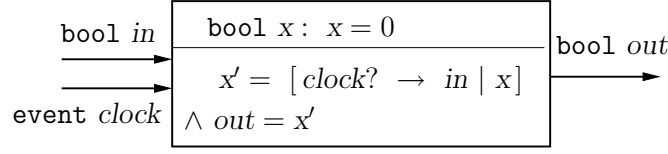
$$n \xrightarrow{\perp/\perp} n; n \xrightarrow{0/0} n + 1; n \xrightarrow{1/1} n + 1.$$

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 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, and the component does not need to be aware of the number of successive global rounds in which the trigger is absent.

As another example, consider the clock-triggered component `ClockedCopy` shown in Figure 1.5. It has a Boolean input variable `in`, and an event input variable `clock` that acts as a trigger. Every time the clock event is present, the component updates its state `x` to the current input `in`. Any changes to the input `in` in rounds in which `clock` is absent are not processed. The output `out` is a

Figure 1.5: Event-triggered component `ClockedCopy`

Boolean variable, and its value equals the updated state. The output always has a value, even in rounds in which the trigger *clock* is absent, and equals the value of the input *in* in the most recent round in which *clock* was present. One possible execution of the component is the following (input is listed as a pair with the value of *in* followed by the value of *clock*):

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

We call an output variable such as *out* of `ClockedCopy` as a *latched* output: its value equals the updated value of one of the state variables. More formally, for a synchronous reactive component $C = (I, O, S, Init, React)$, an output variable $y \in O$ is said to be *latched* if there exists a state variable $x \in S$ such that in every reaction $s \xrightarrow{i/o} t$ of the component, $o(y) = t(x)$. In the implementation of a component, a latched output does not need to be explicitly stored, the corresponding state variable needs to be made accessible to other components.

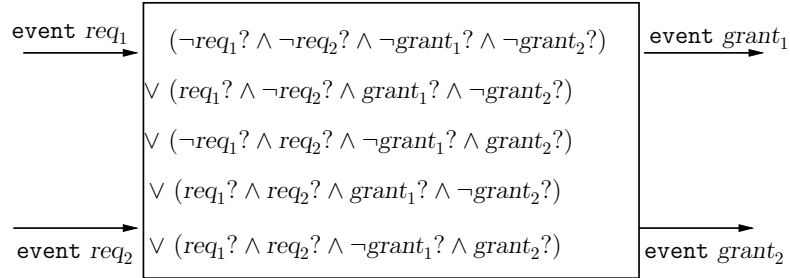
Now we can define the notion of an event-triggered component in its generality. Each of its output variables should be either latched or be an event. When the triggering input events are absent, the state should stay unchanged (and as a result, the latched outputs also stay unchanged) and event outputs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

EVENT-TRIGGERED COMPONENT

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

1. all input variables in J are of event type,
2. every non-event output variable in O is latched, 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 all event output variables y .

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 .

Figure 1.6: Nondeterministic component **Arbiter**

1.2.4 Nondeterministic Components

In our examples so far, the components were *deterministic*: given a sequence of inputs, the component produces 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.

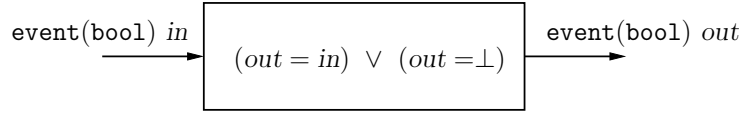
DETERMINISTIC COMPONENT

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 input i , there is precisely one output o and one state t such that $s \xrightarrow{i/o} t$ is a reaction of C .

All of **Delay**, **Comparator**, **TriggeredCopy**, and **ClockedCopy** are deterministic. Determinism is a desirable property for components that are meant to be implemented.

Nondeterministic components, on the other hand, can respond with different outputs for the same input. 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 1.6. 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 supposed to resolve contention among incoming requests. When only one of the requests is present, the component issues the corresponding grant output; that is, if req_1 is present and req_2 is absent, then the component sets $grant_1$ to present and $grant_2$ to absent (and vice versa). If both requests are absent, then both the outputs are absent. However, if both requests are present, then there are two possible computations 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 specification captures what an arbiter should do, namely, a grant output should be issued only when requested, and at most one grant output should be issued

Figure 1.7: Nondeterministic component `LossyCopy`

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 1.7. It has an event variable `in` and an event output variable `out`. The specification of the reaction expression says that in each round, either the input is copied to the output, or the output is absent. Such a component can be used to model loss of messages along a network link. One possible execution of the component is

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

1.2.5 Input-enabled Components

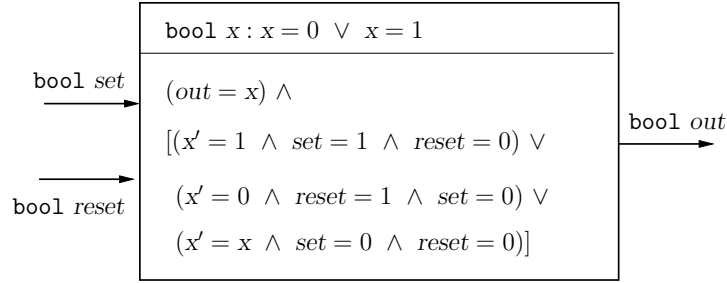
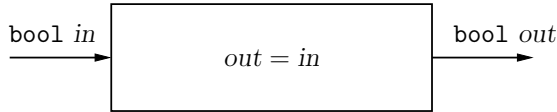
All the components we have seen so far have the following property: in every state and for every input, the component has at least one reaction. This property makes the components *input-enabled*. For every sequence of inputs, an input-enabled component has at least one corresponding execution. By definition,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

INPUT-ENABLED COMPONENT

For a synchronous reactive component C , its 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 `Latch` component, of Figure 1.8, which maintains an internal state using a state variable x . The initial value of x is unspecified, so the latch has 2 initial states. The component has two Boolean input variables `set` and `reset`: when `set` is high (1), the latch state should be set to 1, and when `reset` is high, the latch state should be reset to 0. The latch does not expect both `set` and `reset` to be high simultaneously. We express this as a clearly stated assumption on inputs:

$$\text{Input } i \text{ to } \text{Latch} \text{ satisfies } \neg(\text{set} = 1 \wedge \text{reset} = 1).$$

Figure 1.8: Component **Latch** with input assumptionsFigure 1.9: The combinational component **Relay**

Now we can specify the reactions of **Latch** only for inputs satisfying this input assumption. The output out is set to the current value of the state variable x . The update of x is defined by: if $set = 1$ and $reset = 0$ then x is updated to 1; conversely, if $set = 0$ and $reset = 1$ then x is updated to 0; and if both set and $reset$ are 0, the value of x stays unchanged.

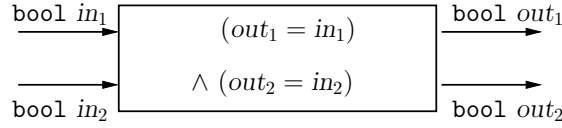
In general, the input assumption can be a constraint on the sequence of inputs supplied to the component. For instance, it may be useful to assume that a particular input variable does not stay high for two consecutive rounds. When a component with input assumptions is used as part of a larger system, we need to check that its input assumptions are indeed satisfied.

1.2.6 Output-Input Await Dependencies

Let us consider a component **Relay** that has a Boolean input variable in and a Boolean output variable out , see Figure 1.9. The component **Relay** is a combinational component without any state, and in each round, simply copies the input to the output: its reactions are $s_\emptyset \xrightarrow{0/0} s_\emptyset$ and $s_\emptyset \xrightarrow{1/1} s_\emptyset$.

Let us compare the components **Relay** and **Delay**. Observe that they have identical input/output connections. In any round, the component **Delay** can produce its output without waiting for the input in that round, while the component **Relay** can produce its output only after reading the input for the current round.

We formalize such dependence by defining *awaits-dependencies* among output and input variables. For simplicity of presentation, let us restrict the discussion

Figure 1.10: Component `ParallelRelay`: out_1 awaits in_1 and out_2 awaits in_2

to deterministic components. For a deterministic component, the value of an output variable in a round is a function of the state at the beginning of the round and the input. The awaits-dependency \succ is a relation between output variables O and input variables I such that $y \succ x$ means that the output variable y depends on the input variable x , and its value in a given round cannot be determined without knowing the value of x . The dependency has the property that the function that determines the value of y depends on the state at the beginning of the round and values of only those input variables that y depends on. More precisely, consider a state s and two inputs i_1 and i_2 such that the inputs i_1 and i_2 agree on all the input variables that y depends on (that is, if $y \succ x$ then $i_1(x) = i_2(x)$). Then, if o_1 is the output of the unique reaction of state s to input i_1 , and if o_2 is the output of the unique reaction of state s to input i_2 , then these outputs should agree on the value of y : $o_1(y) = o_2(y)$.

For the component `Delay`, the awaits-dependency relation is $out \succ in$; for the component `Relay`, the awaits-dependency relation is empty; for the component `Comparator`, the awaits-dependency relation is $out \succ in_1$ and $out \succ in_2$; for the component `TriggeredCopy`, the awaits-dependency relation is $out \succ in$; and for the component `ClockedCopy`, the awaits-dependency relation is $out \succ in$ and $out \succ clock$.

AWAITS DEPENDENCY

Let $C = (I, O, S, Init, React)$ be a deterministic synchronous reactive component. A relation $\succ \subseteq O \times I$ between the output and input variables of C is an *awaits-dependency* relation if for all output variables y , for all states s , and for all pairs of reactions $s \xrightarrow{i_1/o_1} t_1$ and $s \xrightarrow{i_2/o_2} t_2$ of the component C , if $i_1(x) = i_2(x)$ for all input variables x such that $y \succ x$, then $o_1(y) = o_2(y)$.

Note by definition, the complete relation $O \times I$, where every output variable awaits every input variable, is an awaits-dependency relation. It is more useful to capture the *minimal* set of dependencies.

An output variable may await only some of the input variables, and different output variables may await different input variables. To illustrate this point, consider the combinational component `ParallelRelay` (see Figure 1.10) 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 out_1 , and copies

the input variable in_2 to output out_2 . Then, out_1 awaits in_1 , but not in_2 , and out_2 awaits in_2 , but not in_1 .

The awaits-dependencies provide useful information for determining if a network of components is well-formed, and also for code generation. This information can be extracted from syntactic analysis of the source code describing the reactions of the component. A better approach is to explicitly specify the awaits-dependencies along with the declaration of the variables, and use typing rules to enforce that the description of the reaction relation uses variables in a manner consistent with the declared dependencies.

1.2.7 Exercises

1. Consider the component `OddDelay` from Exercise 1.1.5.1. Is the component finite-state? Is the component deterministic? Draw the Mealy machine corresponding to `OddDelay`.
2. Describe the AND, OR, and NOT gates of synchronous circuits as combinational synchronous reactive components. The component `SyncAnd` should have two Boolean input variables in_1 and in_2 , and a Boolean output variable out , which should reflect the logical conjunction of the two inputs. The component `SyncOr` is similar, with its output denoting the logical disjunction of its inputs. The component `SyncNot` has a Boolean input variable in and a Boolean output variable out , which should be set the negation of its input.
3. Design an event-triggered, deterministic, 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 inputs x and y in rounds in which $clock$ is present.
4. Design an event-triggered deterministic component `SecondToMinute` with the event input variable $second$ and the event output variable $minute$ such that $minute$ is present every 60^{th} time the event $second$ is present.
5. Design a component `ClockedDelay` with a Boolean input variable x , input event variable $clock$, and output variable y of type `event(bool)` with the following behavior: if $clock$ is present in rounds, say, $n_1 < n_2 < n_3 < \dots$ 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 where $clock$ is absent, output should be absent too.
6. Design a nondeterministic component `LatchEnv` that supplies inputs to the latch of Figure 1.8. The component `LatchEnv` has no inputs, and 3 Boolean output variables in , set , and $reset$. It should produce all possible combinations of outputs as long as (1) in a given round, it is never the

case that both *set* and *reset* are 1 simultaneously, (2) *set* is never 1 in two consecutive rounds, and (3) *reset* is never 1 in two consecutive rounds.

1.3 Composing Components

1.3.1 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 output is 0, and in every subsequent round $n > 2$, 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 should suffice. The resulting design of the component `DoubleDelay` is shown in Figure 1.11. 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. We see 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 of the input/output variables. For example, `Delay1` is exactly like `Delay` except its output variable is called *temp* instead of *out*. We also need to implicitly rename state variables as they act like local variables not meaningful or accessible outside the component.
- **Parallel Composition:** The two components `Delay1` and `Delay2` run in parallel. The block diagram shows that the output of `Delay1` is same as the input of `Delay2`, and this achieves communication among the two components. The communication is synchronous. In each round, `Delay1` reads its input *in*, produces output *temp*, and updates its internal state to record the current value of *in*. In the same round, `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:** From the description of `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 *temp* is not exported to the outside world.

The component `DoubleDelay` is formally defined as

$$(\text{Delay}[out \mapsto temp] \parallel \text{Delay}[in \mapsto temp]) \setminus temp$$

We proceed to discuss the three operations in the above expression in more details.

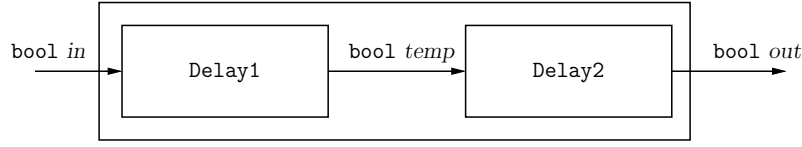


Figure 1.11: Block diagram for DoubleDealy from two Delay components

1.3.2 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 input/output connections. We assume that the renaming of state variables is implicit. For instance, in Figure 1.11, we can assume that the state variable of `Delay1` is called x_1 instead of x , and the state variable of `Delay2` is called x_2 . The renaming of input/output variables needs to be defined explicitly since it is central to the communication pattern:

INPUT VARIABLE RENAMING

Let $C = (I, O, S, Init, React)$ be a synchronous reactive component, $x \in I$ be an input variable, and y be a variable such that the types of x and y are the same and $y \notin I \cup O \cup S$. Then the component obtained by *renaming x to y in C* , denoted $C[x \mapsto y]$, is the synchronous reactive component $(I \setminus \{x\} \cup \{y\}, O, S, Init, React[x \mapsto y])$.

For a component C , an output variable x , and a variable y such that y has same type as x and y is not a variable of C , the renamed component $C[x \mapsto y]$ is defined in an analogous manner: simply replace x by y in the set of output variables and in the reaction expression.

With this notation, in Figure 1.11, `Delay1` is `Delay[out \mapsto temp]` and `Delay2` is `Delay[in \mapsto temp]`. For the component `Delay1`, the set of input variables is $\{in\}$, the set of output variables is $\{temp\}$, the set of state variables is $\{x_1\}$, the initialization expression is $x_1 = 0$, and the reaction expression is $temp = x_1 \wedge x_1' = in$.

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.

1.3.3 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. Two components can be composed only if

their variable declarations are mutually consistent: there are no name conflicts concerning state variables, and the output variables are disjoint. These requirements capture the assumption that only one component is responsible for controlling the value of any given variable. Input variables of one can be either input or output variables of the other.

COMPONENT COMPATIBILITY

The two components $C_1 = (I_1, O_1, S_1, Init_1, React_1)$ and $C_2 = (I_2, O_2, S_2, Init_2, React_2)$ are *compatible* if (1) S_1 is disjoint from each of I_2 , O_2 , and S_2 ; (2) S_2 is disjoint from each of I_1 , O_1 , and S_1 ; and (3) O_1 and O_2 are disjoint.

If either the condition (1) or (2) in the compatibility requirement is violated, we assume that state variables are implicitly renamed to avoid the conflict. For instance, the variable name may be prefixed by the name of the component instance.

The composition of two compatible components is defined below:

COMPONENT COMPOSITION

Let $C_1 = (I_1, O_1, S_1, Init_1, React_1)$ and $C_2 = (I_2, O_2, S_2, Init_2, React_2)$ be compatible components. Then the *parallel composition* $C_1 || C_2$ is a synchronous reactive component $C = (I, O, S, Init, React)$ such that

- each state variable of a component is a state variable of the composite: $S = S_1 \cup S_2$;
- each output variable of a component is an output variable of the composite: $O = O_1 \cup O_2$;
- each input variable of a component is an input variable of the composite, provided it is not an output variable of the other component: $I = (I_1 \cup I_2) \setminus O$;
- the initialization expression for the composite is the conjunction of the components' initialization constraints: $Init = Init_1 \wedge Init_2$; and
- the reaction expression for the composite is the conjunction of the components' reaction constraints: $React = React_1 \wedge React_2$.

Observe that the set of initial states of the composite is the product of the set of initial states of the components: a valuation s for the variables $S_1 \cup S_2$ is an initial state of C precisely when it is consistent with both initialization expressions, that is, (1) $s(S_1)$ is an initial state of C_1 and (2) $s(S_2)$ is an initial state of C_2 . The set of reactions of the composite is also a product of the sets of components' reactions. A state s of the composite on its input i can react with an output o transitioning to a state t precisely when this reaction is consistent

with both the reaction expressions, that is, (1) in component C_1 , state $s(S_1)$ on input $[i, o](I_1)$ can produce output $o(O_1)$ transitioning to state $t(S_1)$, and (2) in component C_2 , state $s(S_2)$ on input $[i, o](I_2)$ can produce output $o(O_2)$ transitioning to state $t(S_2)$.

For example, the composition of `Delay1` and `Delay2` gives the component with state variables $\{x_1, x_2\}$, output variables $\{temp, out\}$, input variables $\{in\}$, initialization expression $x_1 = 0 \wedge x_2 = 0$, and reaction expression

$$temp = x_1 \wedge x'_1 = in \wedge out = x_2 \wedge x'_2 = temp.$$

The only initial state of the composed component is $(0, 0)$, and its reactions are:

$$\begin{aligned} (0, 0) \xrightarrow{0/(0,0)} (0, 0); & (0, 0) \xrightarrow{1/(0,0)} (1, 0); & (0, 1) \xrightarrow{0/(0,1)} (0, 0); & (0, 1) \xrightarrow{1/(0,1)} (1, 0); \\ (1, 0) \xrightarrow{0/(1,0)} (0, 1); & (1, 0) \xrightarrow{1/(1,0)} (1, 1); & (1, 1) \xrightarrow{0/(1,1)} (0, 1); & (1, 1) \xrightarrow{1/(1,1)} (1, 1). \end{aligned}$$

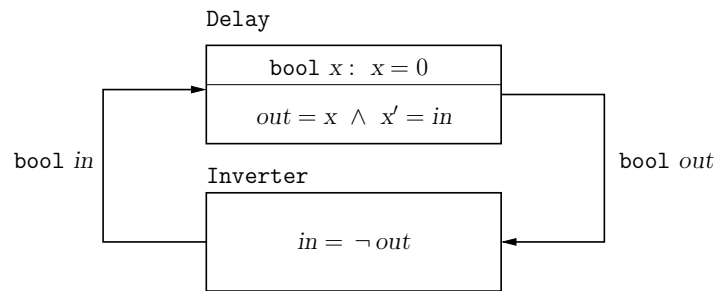
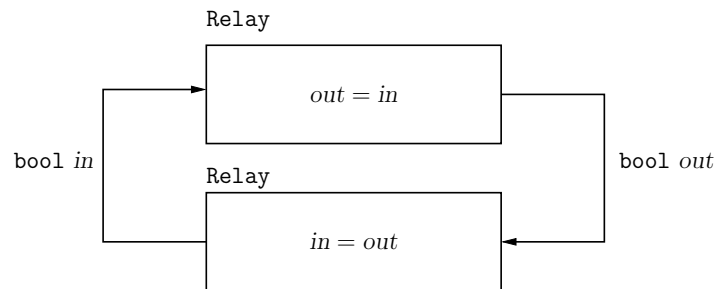
Let $C_1 = (I_1, O_1, S_1, Init_1, React_1)$ and $C_2 = (I_2, O_2, S_2, Init_2, React_2)$ be compatible components, and let $C_1 \parallel C_2$ be $(I, O, S, Init_1 \wedge Init_2, React_1 \wedge React_2)$. Then $C_2 \parallel C_1$ will be the component $(I, O, S, Init_2 \wedge Init_1, React_2 \wedge React_1)$. Since the conjunction operation is commutative in terms of its meaning, the two composites have identical inputs, outputs, states, initial states, and reactions. As a result, we can consider the parallel composition operation to be *commutative*.

By a similar argument, parallel composition is also associative: $(C_1 \parallel C_2) \parallel C_3$ is same as $C_1 \parallel (C_2 \parallel C_3)$. Thus, if we want to compose multiple components, 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 common outputs, and may not be able to proceed with taking parallel composition, but this also does not depend on the order in which the components are composed.

If both C_1 and C_2 are finite-state components, then so is the composite $C_1 \parallel C_2$. If C_1 has n_1 states and C_2 has n_2 states, then $C_1 \parallel C_2$ has $n_1 * n_2$ states. For example, in the composition of `Delay1` and `Delay2`, each component has 2 states, and the composite has 4 states. If we were to compose n `Delay` components in a chain to construct a component that outputs, in each round, the value of the input n rounds ago, 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.

1.3.4 Well-formed Composition

Our definition allows composition of components as long as their outputs are disjoint. In Figure 1.11, outputs of one component are connected to inputs of another. As another example, consider the composition shown in Figure 1.12 in which the output *out* of a `Delay` component is connected to the input of an

Figure 1.12: Feedback composition of a **Delay** with a **Relay** componentFigure 1.13: Ill-formed combinational loop with two **Relay** components

Inverter component, and vice versa. The **Inverter** component is a combinational component that sets its output to the negation of its input. This form of cyclic composition is called *feedback composition*. The composite has no input variables, two output variables in and out , and one state variable x . In the first round out is 0 and in is 1, and in every subsequent round, both these values toggle. That is, the sequence of outputs produced by the composite, listing the value of in first and out second, is 10,01,10,01,10,... The composite is deterministic.

Observe that in the feedback composition of **Delay** and **Inverter** of Figure 1.12, for the **Inverter**, in awaits out , but for the **Delay**, there is no awaits-dependency between out and in . The absence of mutual awaits-dependency is a key for the deterministic behavior of the composite. Composing components with mutually cyclic awaits-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 1.13 shows composition of two **Relay** components: the top component copies its input in to its output out , while the bottom component copies its input out to its output in . Both are deterministic: for one component out awaits in ,

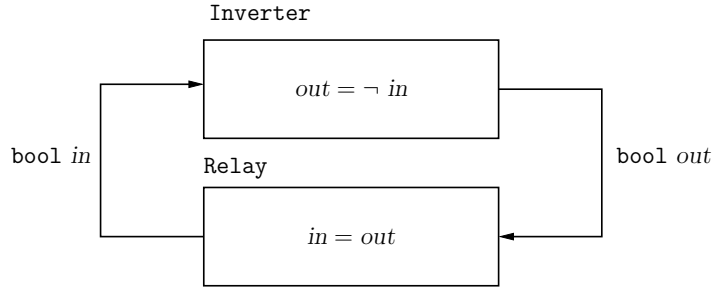


Figure 1.14: Ill-formed combinational loop with a Relay and an Inverter

while for the other in awaits out . By our definition, the composite component has no state variables, and no input variables. Its output variables are in and out . The conjoined reaction expression is equivalent to $in = out$. Consistent with this, in each round, it can produce two outputs: one possibility is that both in and out are set to 0, and the other possibility is that both in and out are set to 1. Thus, the composition is not deterministic.

A converse of multiple possible consistent reactions arises in composition of an inverter component with a relay component shown in Figure 1.14. The top component is an **Inverter** that sets its output to the negation of its input, and the bottom component is a **Relay**. Both are deterministic: for the top component out awaits in , while for the bottom component in awaits out causing a combinational cycle. The conjoined reaction expression is $in = \neg out \wedge out = in$, which is not satisfiable. Thus, the component has no possible reactions, and is not input-enabled.

It is preferable to detect and rule out such ill-formed compositions. One possible approach to composing deterministic components relies on checking for cycles in the union of awaits-dependencies of the two components. If a cycle is found, the composition is disallowed, or at least the designer is warned about the cyclic dependence. For instance, in Figure 1.13, the awaits-dependency for the top **Relay** is $out \succ_1 in$ and for the bottom component is $in \succ_2 out$. The union of the two relations gives the cycle $out \succ_1 in \succ_2 out$. In general, the cyclicity requirement means that there are common input/output variables x_1, x_2, \dots, x_n , with $x_n = x_1$ such that for each $1 \leq j < n$, x_j awaits x_{j+1} according to either one of the two awaits-dependency relations: either $x_j \succ_1 x_{j+1}$ or $x_j \succ_2 x_{j+1}$. The absence of such a cycle corresponds to the transitive closure of the union of the two awaits-dependency relations $\succ_1 \cup \succ_2$ being irreflexive. We call this condition awaits-compatibility:

AWAITS COMPATIBILITY FOR DETERMINISTIC COMPONENTS

Two deterministic components $C_1 = (I_1, O_1, S_1, Init_1, React_1)$ and $C_2 = (I_2, O_2, S_2, Init_2, React_2)$ are said to be *awaits-compatible* if they are compatible, and there exist relations $\succ_1 \subseteq O_1 \times I_1$ and $\succ_2 \subseteq O_2 \times I_2$ such that (1) \succ_1 is an awaits-dependency relation for C_1 , (2) \succ_2 is an awaits-dependency relation for C_2 , and (3) the transitive closure of $(\succ_1 \cup \succ_2)$ is irreflexive.

Theorem 1.1 [Well-formed Composition of Awaits-Compatible Components]
If C_1 and C_2 are awaits-compatible deterministic components then the parallel composition $C_1 \parallel C_2$ is deterministic.

Proof. Let $C_1 = (I_1, O_1, S_1, Init_1, React_1)$ and $C_2 = (I_2, O_2, S_2, Init_2, React_2)$ be compatible deterministic components, and let \succ_1 and \succ_2 be the corresponding awaits-dependency relations such that the transitive closure of $(\succ_1 \cup \succ_2)$ is irreflexive. Let $C = (I, O, S, Init, React)$ be the parallel composition $C_1 \parallel C_2$. We want to prove that C is deterministic.

Consider a state s and an input i of C . We will show that there is a unique output o and a unique updated state t such that $s \xrightarrow{i/o} t$ is a reaction of C . For this purpose, we will determine the values of output variables in O one by one.

Consider all the output variables of C , that is, all the variables in $O_1 \cup O_2$. Let us order them x_1, x_2, \dots, x_n in such a way that if either $x \succ_1 y$ or $x \succ_2 y$ according to the awaits-dependencies of one of the two components, then x appears *after* y in the chosen ordering. Since the transitive closure of the union of the awaits-dependencies is acyclic, such a total ordering is always possible. We will assign unique values to all these variables in this order. Consider a variable x_j , for $1 \leq j \leq n$, and assume that we have already assigned unique values to all the variables x_1, \dots, x_{j-1} that appear before x_j in the chosen order. Assume that x_j is an output variable of C_1 (the other possibility is symmetric).

Every reaction of C is consistent with the reaction expression of C_1 . And the reaction of C_1 assigns a value to x_j as a function of the values of state variables in S_1 and the values of those input variables in I_1 that x_j awaits according to \succ_1 . The state of C_1 at the beginning of the round is $s(S_1)$. A variable in I_1 that x_j awaits according to \succ_1 is either an input variable of C , and thus has a value determined by the input i , or is an output variable of C , in which case, appears before x_j in our ordering, and thus has already been assigned a unique value. We can now choose the unique value for x_j that is consistent with $React_1$ and the values assigned so far.

Let o be the unique output of C that has been determined by the above process. Now the values of all the variables in I_1 as well as I_2 have been determined. The input for C_1 in this reaction is $i_1 = [i, o](I_1)$, and for C_2 is $i_2 = [i, o](I_2)$. The updated state t_1 for C_1 is the unique updated state in response to input i_1 in state $s(S_1)$ according to the reaction expression $React_1$, and similarly, the

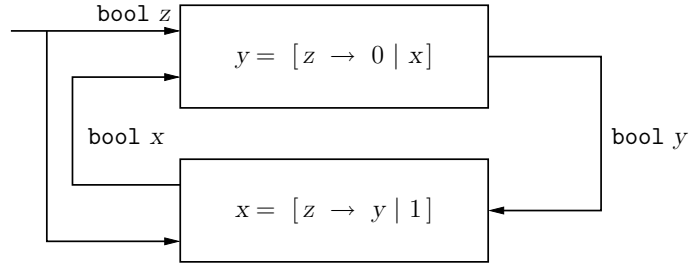


Figure 1.15: Well-formed composition in presence of cyclic awaits-dependencies

updated state t_2 for C_2 is the unique updated state in response to input i_2 in state $s(S_2)$ according to the reaction expression $React_2$. This gives the updated state $[t_1, t_2]$ for the composite component. ■

In practice, the condition of awaits-compatibility is sometimes too strong. To understand that parallel composition can be well-formed even in presence of cyclic (static) awaits-dependencies, consider the composition shown in Figure 1.15. The top component has two input variables x and z , and output variable y . If z is 0, it sets its output y equal to its input x , else it sets output y to 0 independent of the input x . Symmetrically, the bottom component has input variables z and y , and output x . If z is 1, it copies the input y to its output x , and else, it sets its output x to 1. The reactions of the individual components have the following awaits-dependencies: for the top component y awaits both x and z , and for the bottom component x awaits both y and z . Thus, there is an awaits-cycle in the union, and the two components are not awaits-compatible. However, the composition is well-formed. It has a single input variable z , and two output variables x and y . When z is 0, the bottom component sets x to 1, and then the top component copies it to y , and when z is 1, the top component sets y to 0, and then the bottom component copies it to x . We have precisely two reactions in the composition: $s_\emptyset \xrightarrow{0/(1,1)} s_\emptyset$, and $s_\emptyset \xrightarrow{1/(0,0)} s_\emptyset$. The composition is deterministic.

1.3.5 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 , the result of *hiding* y in C gives a component that behaves exactly like C , but y is no longer an output that is observable outside. Syntactically, this is achieved by removing y from the set of outputs, and existentially quantifying y from the reaction expression.

OUTPUT HIDING

If $C = (I, O, S, Init, React)$ is a synchronous reactive component, and $y \in O$ is an output variable, then *hiding y in C* , gives the component, denoted $C \setminus y$, with the set I of input variables, the set $O \setminus \{y\}$ of output variables, the set S of state variables, the initialization expression $Init$, and the reaction expression $\exists y. React$.

Let us revisit the component $\mathbf{Delay1} \parallel \mathbf{Delay2}$. If we hide the intermediate output $temp$, we get the desired composite component $\mathbf{DoubleDelay}$: the set of state variables is $\{x_1, x_2\}$, the set of output variables is $\{out\}$, the set of input variables is $\{in\}$, the initialization expression is

$$x_1 = 0 \wedge x_2 = 0,$$

and the reaction expression is

$$\exists temp. (temp = x_1 \wedge x'_1 = in \wedge out = x_2 \wedge x'_2 = temp),$$

which can be simplified to an equivalent expression

$$x'_1 = in \wedge out = x_2 \wedge x'_2 = x_1.$$

The initial state of the resulting component is $(0, 0)$, and its reactions are:

$$\begin{aligned} (0, 0) &\xrightarrow{0/0} (0, 0); (0, 0) \xrightarrow{1/0} (1, 0); (0, 1) \xrightarrow{0/1} (0, 0); (0, 1) \xrightarrow{1/1} (1, 0); \\ (1, 0) &\xrightarrow{0/0} (0, 1); (1, 0) \xrightarrow{1/0} (1, 1); (1, 1) \xrightarrow{0/1} (0, 1); (1, 1) \xrightarrow{1/1} (1, 1). \end{aligned}$$

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

1.3.6 Exercises

1. Consider the component $\mathbf{ClockedDelay}$ from Exercise 1.2.7.5. The composite component $\mathbf{ClockDelayComparator}$ is defined as follows:

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

Describe the input-output behavior of $\mathbf{ClockDelayComparator}$.

2. First show that if two compatible components C_1 and C_2 are event-triggered, it is not necessarily the case that the composition $C_1 \parallel C_2$ is event-triggered. Then prove that the parallel composition of deterministic awaits-compatible event-triggered components is event-triggered.
3. Suppose we simplify the definition of awaits-compatibility for deterministic components as follows: two deterministic compatible components with awaits-dependency relations \succ_1 and \succ_2 are awaits-compatible if there do

not exist two variables x and y such that $x \succ_1 y$ and $y \succ_2 x$, or $x \succ_2 y$ and $y \succ_1 x$. Show that this condition is not sufficient to ensure that parallel composition preserves determinacy. That is, find two deterministic components C_1 and C_2 that are awaits-compatible according to this modified definition, but $C_1 \parallel C_2$ is not deterministic.

1.4 Synchronous Designs

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

In the classical functional model of computation, the component reads its input, then computes, producing the output upon termination. The specification of the component, that is, the desired behavior, is described as a function from inputs to outputs. Reactive components, on the other hand, interact with their environment, via inputs and outputs in an ongoing manner. In principle, the component never terminates. The desired behavior is described, in general, 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 allows to describe the situation where a component may not be interested in every round, and actively participates only in those rounds in which the triggering events are present. The key assumption of the synchronous model is that the computation of all the components within a round and all the inter-component 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 components ready to process the new inputs. This assumption is called the *synchrony hypothesis*. This idealized assumption leads to simplicity and predictability of designs. During a round, the order in which components execute does not affect the resulting reaction. Nondeterminism, that is, multiple reactions to the same input, needs to be explicitly programmed within the description of a component, 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, 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 separation between changes to the input.

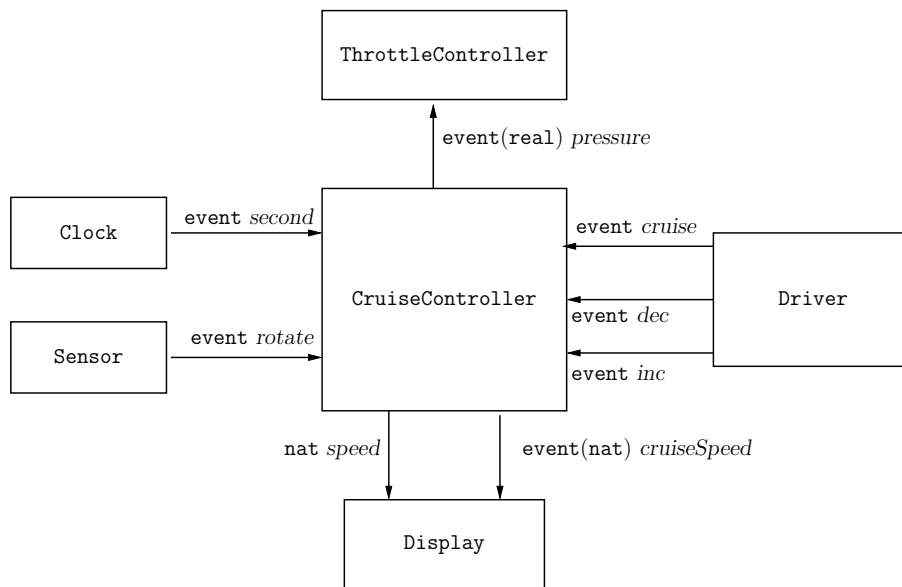


Figure 1.16: Inputs and outputs of the cruise control system

1.4.1 Top-down Design

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 1.16. 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 cruising speed should be set to the current speed, and the events *inc* and *dec*, when present, should cause the desired speed to increment and decrement, respectively. We should ensure though that the desired speed stays within a reasonable cruising range, given by a minimum value, denoted *minSpeed*, and a maximum value, denoted *maxSpeed*.

The cruise controller needs to measure the current speed to make its decisions. This is done using two input events: *rotate* and *second*. Whenever a 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.

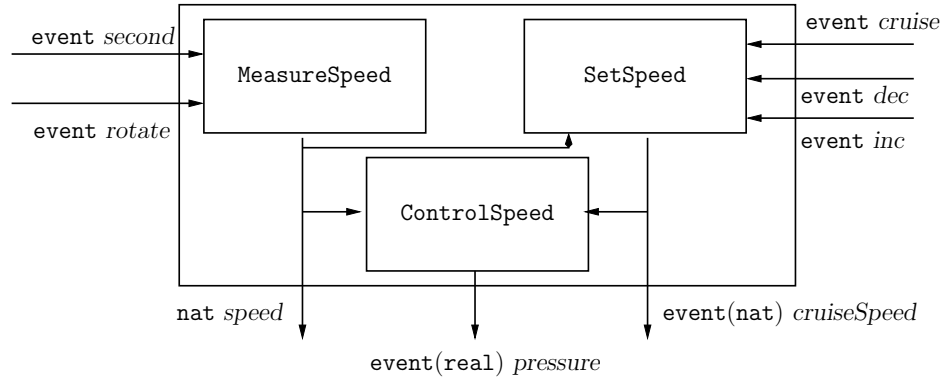


Figure 1.17: Components of the cruise control system `CruiseControl`

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 *pressure*, which may be absent, is sent to the throttle control system to adjust the throttle to regulate the current speed in order to track the desired cruising speed.

Decomposing into subsystems

As a next step in the design, we break down the controller in three subsystems: `MeasureSpeed` to compute the current speed based on the inputs *rotate* and *second*; `SetSpeed` to keep track of the desired cruise settings based on the inputs from the driver and the current speed, and `ControlSpeed` to process the differential between the current speed and desired speed and compute the output *pressure*. The interconnections among these subcomponents are shown in Figure 1.17. The design of the component `ControlSpeed` requires understanding the dynamics of the car and control theory, and we postpone its discussion. We proceed to design the other two components.

Tracking speed

The task for `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 1.18. The component has a state variable *count* that counts the number of times the event *rotate* has occurred since the most recent *second* event. The initial value of *count* is 0. The state variable *s* remembers the current speed: it is 0 initially, and every time *second* occurs, the current value of *count* is used to update *s*. More precisely, the rules for updating the state are as follows:

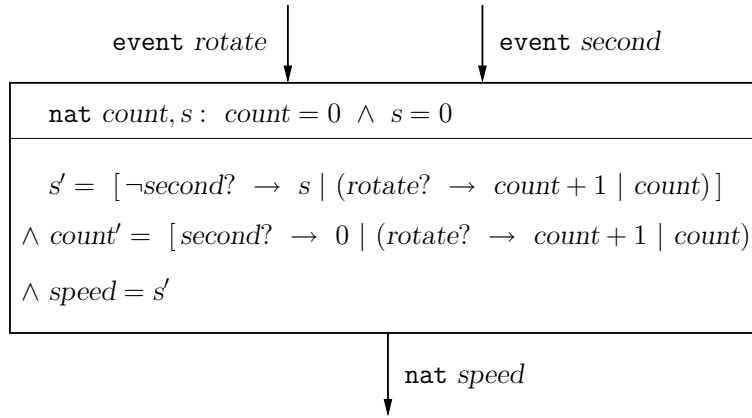


Figure 1.18: Component MeasureSpeed

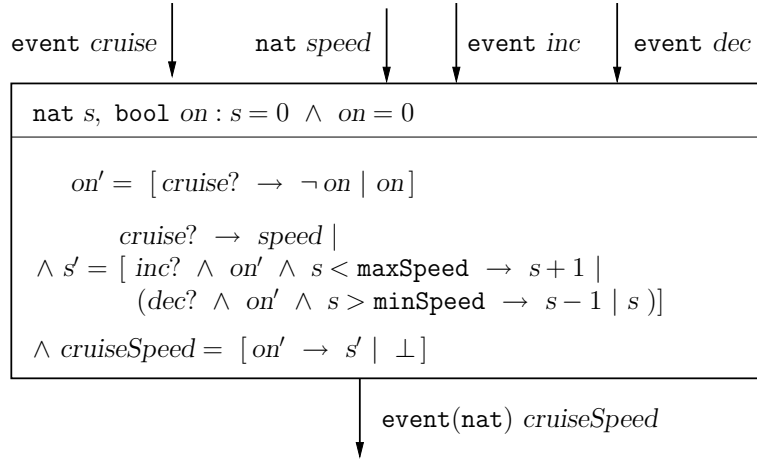
1. In each round, if both input events are absent, the component does nothing, that is, it leaves its state unchanged.
2. If only *rotate* event is present, then the component just increments *count* and leaves *s* unchanged.
3. If only *second* event is present, then the component updates the speed *s* to the current value of *count*, and resets the *count* to 0.
4. We need to also consider the case when both input events are present simultaneously. In such a case, the component sets the speed *s* to the current *count* incremented by 1, and then resets the count.

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

Tracking cruise settings

Now consider the component **SetSpeed**, see Figure 1.19. 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*. Initially, both state variables are set to 0.

Since there are 3 input events for the component, each of which can be present or absent, we need to worry about all their combinations. In our design, we avoid this blow-up by considering these events in a priority order: first *cruise*, then *inc*,

Figure 1.19: Component **SetSpeed**

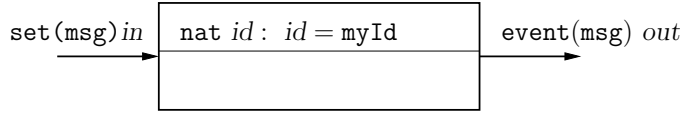
and then *dec*. If the driver presses two or more buttons simultaneously, the effect will be equivalent to pressing one button with the highest priority among those pressed simultaneously. Alternatively, we could make an assumption about the environment that at any instant, at most one of the input events can be present.

The component updates *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 *cruise* is present then *s* is set to the current speed, otherwise, if *inc* is present, then *s* is incremented, provided it is below the maximum threshold and the controller is on, and finally, if *dec* is present, then *s* is decremented, provided it is above the minimum threshold and the controller is on. By default, 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 *cruiseSpeed* is set to the updated value of *s*, else it is set to \perp .

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

1.4.2 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.

Figure 1.20: Schematic of a synchronous network node `SyncNetwkNode`

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 `SyncNetwkNode` with an input variable `in` and an output variable `out`, see Figure 1.20. If the type of messages that a node sends in each round is `msg`, then the type of `out` is `event(msg)` since in each round, a node may or may not send a message. The type of `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 awaits-dependency between `out` and `in`: in each round, the component decides on the value of `out` based on its state, and updates the state in response to the input, that contains the set of messages it receives. The description of the component `SyncNetwkNode` is parameterized by an identifier `myId`.

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

Modeling the interconnections

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

As a concrete example, consider the communication network shown in Figure 1.21 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 2 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 5 and 8, and the set of messages it receives contains messages sent by nodes 1 and 5 in this round.

Figure 1.22 shows the composition of components. There are four instances of `SyncNetwkNode` corresponding to the 4 nodes. The network is captured by `SyncNetwork` with inputs `out1`, `out3`, `out5`, and `out8`, each of type `event(msg)`, connected to the output of the corresponding node component. It has outputs `in1`, `in3`, `in5`, and `in8`, each of type `set(msg)`, connected to the input of the

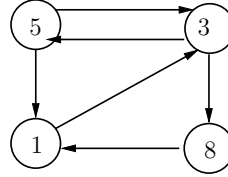


Figure 1.21: Example communication network with 4 nodes

corresponding node component. In each round, the network reads the messages from all its inputs out_n , and for each node n , collects the messages that are present on its incoming links, and delivers the corresponding set of messages on the output in_n . For instance, the rule for in_3 is: if both out_1 and out_5 are present then in_3 is assigned $\{out_1, out_5\}$; if only out_1 is present then in_3 is assigned $\{out_1\}$; if only out_5 is present then in_3 is assigned $\{out_5\}$; and if both are absent then in_3 is assigned the empty set.

More generally, let P be a set of node identifiers and let $E \subset P \times P$ denote the directed edges among nodes. Then, for each $n \in P$, let SyncNetwkNode_n be the instance of SyncNetwkNode obtained by instantiating myID to n , and renaming each input and output variable x to x_n . The component $\text{SyncNetwork}_{P,E}$ is a deterministic combinational component with the set $\{out_n \mid n \in P\}$ of input variables, the set $\{in_n \mid n \in P\}$ of output variables, and the reaction expression given by

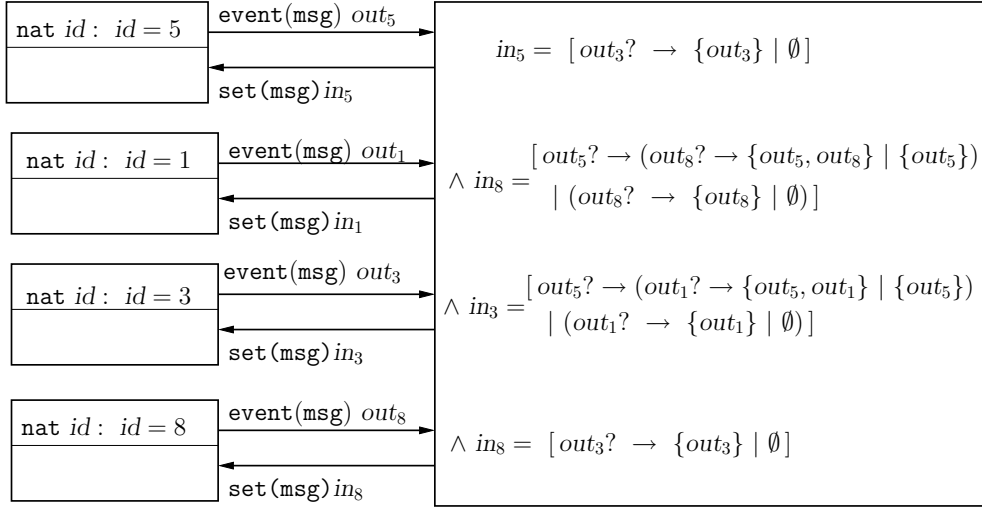
$$\bigwedge_{n \in P} in_n = \{ out_m \mid (m, n) \in E \wedge out_m? \}$$

The desired system is the parallel composition of all the nodes SyncNetwkNode_n , for $n \in P$, and the connecting network $\text{SyncNetwork}_{P,E}$.

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 $\{\text{unknown}, \text{leader}, \text{follower}\}$. The nodes exchange messages updating the $status$ so that (1) eventually every component outputs the $status$ to be either leader or follower , and (2) exactly one component outputs the $status$ to be 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 the 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 1.23 that relies on two assumptions:

Figure 1.22: The synchronous network component `SyncNetwork`

1. The network is strongly connected: for every pair of nodes m and n , there is a directed path from m to n .
2. Each node knows an upper bound N on the total number of nodes in the network.

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

In the algorithm of Figure 1.23, called the *flooding* algorithm, a node maintains a state variable id that is the highest identifier it has seen so far. Initially, 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 it. If the total number of nodes in a strongly connected network is N , then between any 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 of node m will be n or higher. As a result, after $N - 1$ rounds, the value of id in each node will be equal to the highest identifier in the network. At this point, each node can decide: if id equals the node's original identifier, it is the leader, otherwise it is a follower.

Consider the 4-node network shown in Figure 1.22 so that each component is the instantiated version of the leader election component `SyncLENode`. Here is how their executions proceed:

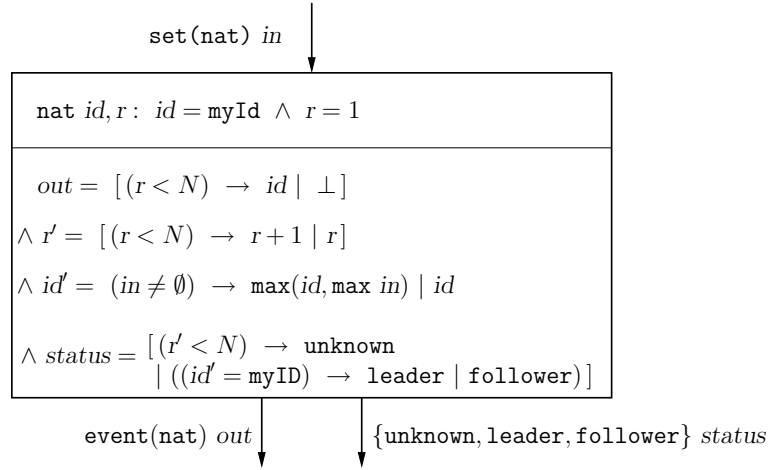


Figure 1.23: Leader Election in synchronous networks

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 to 8; the node 3 receives $\{1, 5\}$, and updates its id to 5; the node 5 receives $\{3\}$, and keeps its id to 5; the node 8 receives $\{3\}$, and keeps its id to 8. All the nodes output $status$ to be **unknown**.
2. In round 2, nodes 1 and 8 send 8, and 3 and 5 send 5. As a result, the id variable of node 3 will be updated to 8, and other nodes do not change their respective id values. Again, all the nodes output $status$ to be **unknown**.
3. In round 3, nodes 1, 3, and 8 output 8, and 5 outputs 5. The node 5 will update its id to 8. The updated value of the round-counting variable r equals $N = 4$, and as a result, all the nodes decide based on the updated value of their respective id variables: the node 8 outputs its $status$ to be **leader**, and the rest output their $status$ to be **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 in 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 , or an upper bound on it, is known to the nodes in advance, then a node can decide at the end of round D .

1.4.3 Exercises

1. Design of synchronous circuits offers an excellent example of bottom-up design using synchronous reactive components. The synchronous latch

component **Latch**, and the combinational gates **SyncNot**, **SyncAnd**, and **SyncOr** (see Exercise 1.2.7.2), suffice to build synchronous circuits. Suppose we want to design a three-bit binary counter. The counter has two Boolean input variables, *start* and *inc*, for starting and incrementing the counter. The counter value ranges from 0 to 7, and is represented by three bits. Setting *start* to 1 should reset the counter value to 0 overriding the value of *inc* in that round, and setting *inc* to 1 should increment the counter by 1 (if the value is 7, it should be changed to 0). The output is represented by 3 Boolean variables *out*₀, *out*₁ and *out*₂ representing the 3 bits of the counter.

First design a 1-bit counter **OneBitCounter** as a synchronous block diagram with Boolean input variables *start* and *inc*, and Boolean output variables *out* and *carry*. You should use one instance of **Latch** and as many instances of the gates **SyncNot**, **SyncAnd**, and **SyncOr** as needed. Make sure that the circuit is designed so that the input assumption for the **Latch** is satisfied. Then design the desired three-bit counter as a synchronous block diagram using 3 instances of **OneBitCounter**.

2. Consider the design of the component **SetSpeed** (see Figure 1.19). 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*, the controller is temporarily turned off. In the resulting paused state, the output *speed* 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 upon 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 event input variable *pause* to capture the above specification.
3. Consider the leader election algorithm in synchronous networks (Figure 1.23). 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 definition of the component **SyncLENode** to implement this change.
4. In a strongly directed 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 *n* to *m*. 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. Design an algorithm for synchronous networks so that each node *n* can figure out the value of D_n . As in 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.