Chapter 10

Composing Specifications

Systems are usually described in terms of their components. In the specifications we've written so far, the components have been represented as separate disjuncts of the next-state action. For example, the FIFO system pictured on page 35 is specified in module *InnerFIFO* on page 38 by representing the three components with the following disjuncts of the next-state action:

Sender: $\exists\, msg \in Message : SSend(msg)$

Buffer: $BufRcv \lor BufSend$

Receiver: $RRcv$

In this chapter, we learn how to specify the components separately and compose their specifications to form a single system specification. Most of the time, there's no point doing this. The two ways of writing the specification differ by only a few lines—a trivial difference in a specification of hundreds or thousands of lines. Still, you may encounter a situation in which it's better to specify a system as a composition.

First, we must understand what it means to compose specifications. We usually say that a TLA formula specifies the correct behavior of a system. However, as explained in Section 2.3 (page 18), a behavior actually represents a possible history of the entire universe, not just of the system. So, it would be more accurate to say that a TLA formula specifies a universe in which the system behaves correctly. Building a system that implements a specification F means constructing the universe so it satisfies F. (Fortunately, correctness of the system depends on the behavior of only a tiny part of the universe, and that's the only part we must build.) Composing two systems whose specifications are F and G means making the universe satisfy both F and G, which is the same as making it satisfy $F \land G$. Thus, the specification of the composition of two systems is the conjunction of their specifications.

Writing a specification as the composition of its components therefore means writing the specification as a conjunction, each conjunct of which can be viewed as the specification of a component. While the basic idea is simple, the details are not always obvious. To simplify the exposition, I begin by considering only safety properties, ignoring liveness and largely ignoring hiding. Liveness and hiding are discussed in Section 10.6.

10.1 Composing Two Specifications

Let's return once again to the simple hour clock, with no liveness or real-time requirement. In Chapter 2, we specified such a clock whose display is represented by the variable hr. We can write that specification as

$$(hr \in 1 \,..\, 12) \,\land\, \Box[HCN(hr)]_{hr}$$

where HCN is defined by

$$HCN(h) \;\triangleq\; h' \,=\, (h \,\%\, 12) + 1$$

Now let's write a specification *TwoClocks* of a system composed of two separate hour clocks, whose displays are represented by the variables x and y. (The two clocks are not synchronized and are completely independent of one another.) We can just define *TwoClocks* to be the conjunction of the two clock specifications

$$\begin{array}{lll} \mathit{TwoClocks} & \triangleq & \land\, (x \in 1 \,..\, 12) \,\land\, \Box[HCN(x)]_x \\ & & \land\, (y \in 1 \,..\, 12) \,\land\, \Box[HCN(y)]_y \end{array}$$

The following calculation shows how we can rewrite *TwoClocks* in the usual form as a "monolithic" specification with a single next-state action:[1]

$$\begin{array}{lll} & \mathit{TwoClocks} & \\ \equiv & \land\, (x \in 1 \,..\, 12) \,\land\, (y \in 1 \,..\, 12) & \\ & \land\, \Box[HCN(x)]_x \,\land\, \Box[HCN(y)]_y & \\ \equiv & \land\, (x \in 1 \,..\, 12) \,\land\, (y \in 1 \,..\, 12) & \text{Because } \Box(F \land G) \equiv (\Box F) \land (\Box G). \\ & \land\, \Box\,(\,[HCN(x)]_x \,\land\, [HCN(y)]_y\,) & \\ \equiv & \land\, (x \in 1 \,..\, 12) \,\land\, (y \in 1 \,..\, 12) & \text{By definition of } [\ldots]_x \text{ and } [\ldots]_y. \\ & \land\, \Box\,(\,\land\, HCN(x) \,\lor\, x' = x & \\ & \qquad \land\, HCN(y) \,\lor\, y' = y\,) & \end{array}$$

[1] This calculation is informal because it contains formulas that are not legal TLA—namely, ones of the form $\Box A$ where A is an action that doesn't have the syntactic form $[B]_v$. However, it can be done rigorously.

$$\begin{array}{l} \equiv \ \land\ (x \in 1\,..\,12)\ \land\ (y \in 1\,..\,12) \\ \quad \land\ \Box\,(\ \lor\ HCN(x) \land HCN(y) \\ \qquad\quad \lor\ HCN(x) \land (y' = y) \\ \qquad\quad \lor\ HCN(y) \land (x' = x) \\ \qquad\quad \lor\ (x' = x) \land (y' = y)\ ) \end{array}$$

Because:

$$\begin{pmatrix} \land \lor A_1 \\ \quad \lor A_2 \\ \land \lor B_1 \\ \quad \lor B_2 \end{pmatrix} \equiv \begin{pmatrix} \lor A_1 \land B_1 \\ \lor A_1 \land B_2 \\ \lor A_2 \land B_1 \\ \lor A_2 \land B_2 \end{pmatrix}$$

$$\begin{array}{l} \equiv \ \land\ (x \in 1\,..\,12)\ \land\ (y \in 1\,..\,12) \\ \quad \land\ \Box\,[\ \lor\ HCN(x) \land HCN(y) \\ \qquad\quad \lor\ HCN(x) \land (y' = y) \\ \qquad\quad \lor\ HCN(y) \land (x' = x)\ ]_{\langle x,\, y\rangle} \end{array}$$

By definition of $[\ldots]_{\langle x,\, y\rangle}$.

Thus, *TwoClocks* is equivalent to $Init \land \Box[TCNxt]_{\langle x,\, y\rangle}$ where the next-state action *TCNxt* is

$$\begin{array}{lll} TCnxt & \triangleq & \lor\ HCN(x) \land HCN(y) \\ & & \lor\ HCN(x) \land (y' = y) \\ & & \lor\ HCN(y) \land (x' = x) \end{array}$$

This next-state action differs from the ones we are used to writing because of the disjunct $HCN(x) \land HCN(y)$, which represents the simultaneous advance of the two displays. In the specifications we have written so far, different components never act simultaneously.

Up until now, we have been writing what are called *interleaving* specifications. In an interleaving specification, each step represents an operation of only one component. For example, in our FIFO specification, a (nonstuttering) step represents an action of either the sender, the buffer, or the receiver. For want of a better term, we describe as *noninterleaving* a specification that, like *TwoClocks*, does permit simultaneous actions by two components.

Suppose we want to write an interleaving specification of the two-clock system as the conjunction of two component specifications. One way is to replace the next-state actions $HCN(x)$ and $HCN(y)$ of the two components by two actions *HCNx* and *HCNy* so that, when we perform the analogous calculation to the one above, we get

$$\begin{pmatrix} \land\ (x \in 1\,..\,12)\ \land\ \Box[HCNx]_x \\ \land\ (y \in 1\,..\,12)\ \land\ \Box[HCNy]_y \end{pmatrix} \equiv \begin{pmatrix} \land\ (x \in 1\,..\,12)\ \land\ (y \in 1\,..\,12) \\ \land\ \Box\,[\ \lor\ HCNx \land (y' = y) \\ \qquad\quad \lor\ HCNy \land (x' = x)\ ]_{\langle x,\, y\rangle} \end{pmatrix}$$

From the calculation above, we see that this equivalence holds if the following three conditions are satisfied: (i) *HCNx* implies $HCN(x)$, (ii) *HCNy* implies $HCN(y)$, and (iii) $HCNx \land HCNy$ implies $x' = x$ or $y' = y$. (Condition (iii) implies that the disjunct $HCNx \land HCNy$ of the next-state action is subsumed by one of the disjuncts $HCNx \land (y' = y)$ and $HCNy \land (x' = x)$.) The common way

of satisfying these conditions is to let the next-state action of each clock assert that the other clock's display is unchanged. We do this by defining

$$HCNx \triangleq HCN(x) \land (y' = y) \qquad HCNy \triangleq HCN(y) \land (x' = x)$$

Another way to write an interleaving specification is simply to disallow simultaneous changes to both clock displays. We can do this by taking as our specification the formula

$$TwoClocks \land \Box[(x' = x) \lor (y' = y)]_{\langle x, y \rangle}$$

The second conjunct asserts that any step must leave x or y (or both) unchanged.

Everything we have done for the two-clock system generalizes to any system comprising two components. The same calculation as above shows that if

$$({v_1}' = v_1) \land ({v_2}' = v_2) \equiv (v' = v)$$ This asserts that v is unchanged iff both v_1 and v_2 are.

then

$$(10.1)\quad \begin{pmatrix} \land\ I_1 \land \Box[N_1]_{v_1} \\ \land\ I_2 \land \Box[N_2]_{v_2} \end{pmatrix} \equiv \begin{pmatrix} \land\ I_1 \land\ I_2 \\ \land \Box\,[\lor\ N_1 \land N_2 \\ \qquad \lor\ N_1 \land ({v_2}' = v_2) \\ \qquad \lor\ N_2 \land ({v_1}' = v_1)\,]_v \end{pmatrix}$$

for any state predicates I_1 and I_2 and any actions N_1 and N_2. The left-hand side of this equivalence represents the composition of two component specifications if v_k is a tuple containing the variables that describe the k^{th} component, for $k = 1, 2$, and v is the tuple of all the variables.

The equivalent formulas in (10.1) represent an interleaving specification if the first disjunct in the next-state action of the right-hand side is redundant, so it can be removed. This is the case if $N_1 \land N_2$ implies that v_1 or v_2 is unchanged. The usual way to ensure that this condition is satisfied is by defining each N_k so it implies that the other component's tuple is left unchanged. Another way to obtain an interleaving specification is by conjoining the formula $\Box[({v_1}' = v_1) \lor ({v_2}' = v_2)]_v$.

10.2 Composing Many Specifications

We can generalize (10.1) to the composition of any set C of components. Because universal quantification generalizes conjunction, the following rule is a generalization of (10.1):

Composition Rule For any set C, if

$$(\forall\, k \in C : {v_k}' = v_k) \equiv (v' = v)$$ This asserts that v is unchanged iff all the v_k are.

then

$$
\begin{array}{l}
(\forall\, k \in C \,:\, I_k \,\wedge\, \Box[N_k]_{v_k}) \;\equiv \\
\quad \wedge\, \forall\, k \in C \,:\, I_k \\
\quad \wedge\, \Box \begin{bmatrix} \vee\, \exists\, k \in C \,:\, N_k \,\wedge\, (\forall\, i \in C \setminus \{k\} \,:\, {v_i}' = v_i) \\ \vee\, \exists\, i, j \in C \,:\, (i \neq j) \,\wedge\, N_i \,\wedge\, N_j \,\wedge\, F_{ij} \end{bmatrix}_v
\end{array}
$$

for some actions F_{ij}.

The second disjunct of the next-state action is redundant, and we have an interleaving specification, if each N_i implies that v_j is unchanged, for all $j \neq i$. However, for this to hold, N_i must mention v_j for components j other than i. You might object to this approach—either on philosophical grounds, because you feel that the specification of one component should not mention the state of another component, or because mentioning other component's variables complicates the component's specification. An alternative approach is simply to assert interleaving. You can do this by conjoining the following formula, which states that no step changes both v_i and v_j, for any i and j with $i \neq j$:

$$\Box\,[\,\exists\, k \in C \,:\, \forall\, i \in C \setminus \{k\} \,:\, {v_i}' = v_i\,]_v$$

This conjunct can be viewed as a global condition, not attached to any component's specification.

For the left-hand side of the conclusion of the Composition Rule to represent the composition of separate components, the v_k need not be composed of separate variables. They could contain different "parts" of the same variable that describe different components. For example, our system might consist of a set *Clock* of separate, independent clocks, where clock k's display is described by the value of $hr[k]$. Then v_k would equal $hr[k]$. It's easy to specify such an array of clocks as a composition. Using the definition of *HCN* on page 136 above, we can write the specification as

$$\textit{ClockArray} \;\triangleq\; \forall\, k \in \textit{Clock} \,:\, (hr[k] \in 1 \,..\, 12) \,\wedge\, \Box[\textit{HCN}(hr[k])]_{hr[k]} \tag{10.2}$$

This is a noninterleaving specification, since it allows simultaneous steps by different clocks.

Suppose we wanted to use the Composition Rule to express *ClockArray* as a monolithic specification. What would we substitute for v? Our first thought is to substitute hr for v. However, the hypothesis of the rule requires that v must be left unchanged iff $hr[k]$ is left unchanged, for all $k \in \textit{Clock}$. However, as explained in Section 6.5 on page 72, specifying the values of $hr[k]'$ for all $k \in \textit{Clock}$ does not specify the value of hr. It doesn't even imply that hr is a function. We must substitute for v the function *hrfcn* defined by

$$\textit{hrfcn} \;\triangleq\; [k \in \textit{Clock} \mapsto hr[k]] \tag{10.3}$$

The function *hrfcn* equals *hr* iff *hr* is a function with domain *Clock*. Formula *ClockArray* does not imply that *hr* is always a function. It specifies the possible values of $hr[k]$, for all $k \in Clock$, but it doesn't specify the value of *hr*. Even if we changed the initial condition to imply that *hr* is initially a function with domain *Clock*, formula *ClockArray* would not imply that *hr* is always a function. For example, it would still allow "stuttering" steps that leave each $hr[k]$ unchanged, but change *hr* in unknown ways.

We might prefer to write a specification in which *hr* is a function with domain *Clock*. One way of doing this is to conjoin to the specification the formula $\Box IsFcnOn(hr, Clock)$, where $IsFcnOn(hr, Clock)$ asserts that *hr* is an arbitrary function with domain *Clock*. The operator *IsFcnOn* is defined by

$$IsFcnOn(f, S) \triangleq f = [x \in S \mapsto f[x]]$$

We can view the formula $\Box IsFcnOn(hr, Clock)$ as a global constraint on *hr*, while the value of $hr[k]$ for each component k is described by that component's specification.

Now, suppose we want to write an interleaving specification of the array of clocks as the composition of specifications of the individual clocks. In general, the conjunction in the Composition Rule is an interleaving specification if each N_k implies that v_i is unchanged, for all $i \neq k$. So, we want the next-state action N_k of clock k to imply that $hr[i]$ is unchanged for every clock i other than k. The most obvious way to do this is to define N_k to equal

$$\begin{array}{l} \land\ hr'[k] = (hr[k] \,\%\, 12) + 1 \\ \land\ \forall\, i \in Clock \setminus \{k\} : hr'[i] = hr[i] \end{array}$$

We can express this formula more compactly using the EXCEPT construct. This construct applies only to functions, so we must choose whether or not to require *hr* to be a function. If *hr* is a function, then we can let N_k equal

The EXCEPT construct is explained in Section 5.2 on page 48.

$$(10.4)\quad hr' = [hr \text{ EXCEPT } ![k] = (hr[k] \,\%\, 12) + 1]$$

As noted above, we can ensure that *hr* is a function by conjoining the formula $\Box IsFcnOn(hr, Clock)$ to the specification. Another way is to define the state function *hrfcn* by (10.3) on the preceding page and let $N(k)$ equal

$$hrfcn' = [hrfcn \text{ EXCEPT } ![k] = (hr[k] \,\%\, 12) + 1]$$

A specification is just a mathematical formula; as we've seen before, there are often many equivalent ways of writing a formula. Which one you choose is usually a matter of taste.

10.3 The FIFO

Let's now specify the FIFO, described in Chapter 4, as the composition of its three components—the Sender, the Buffer, and the Receiver. We start with the

internal specification, in which the variable q occurs—that is, q is not hidden. First, we decide what part of the state describes each component. The variables *in* and *out* are channels. Recall that the *Channel* module (page 30) specifies a channel *chan* to be a record with *val*, *rdy*, and *ack* components. The *Send* action, which sends a value, modifies the *val* and *rdy* components; the *Rcv* action, which receives a value, modifies the *ack* component. So, the components' states are described by the following state functions:

Sender: $\langle in.val,\ in.rdy \rangle$

Buffer: $\langle in.ack,\ q,\ out.val,\ out.rdy \rangle$

Receiver: $out.ack$

Unfortunately, we can't reuse the definitions from the *InnerFIFO* module on page 38 for the following reason. The variable q, which is hidden in the final specification, is part of the Buffer component's internal state. Therefore, it should not appear in the specifications of the Sender or Receiver component. The Sender and Receiver actions defined in the *InnerFIFO* module all mention q, so we can't use them. We therefore won't bother reusing that module. However, instead of starting completely from scratch, we can make use of the *Send* and *Rcv* actions from the *Channel* module on page 30 to describe the changes to *in* and *out*.

Let's write a noninterleaving specification. The next-state actions of the components are then the same as the corresponding disjuncts of the *Next* action in module *InnerFIFO*, except that they do not mention the parts of the states belonging to the other components. These contain *Send* and *Rcv* actions, instantiated from the *Channel* module, which use the EXCEPT construct. As noted above, we can apply EXCEPT only to functions—and to records, which are functions. We therefore add to our specification the conjunct

Section 5.2 on page 48 explains why records are functions.

$$\Box(IsChannel(in) \land IsChannel(out))$$

where $IsChannel(c)$ asserts that c is a channel—that is a record with *val*, *ack*, and *rdy* fields. Since a record with *val*, *ack*, and *rdy* fields is a function whose domain is {"val", "ack", "rdy"}, we can define $IsChannel(c)$ to equal $IsFcnOn(c, \{\text{"val"}, \text{"ack"}, \text{"rdy"}\})$. However, it's just as easy to define formula $IsChannel(c)$ directly by

$$IsChannel(c) \triangleq c = [ack \mapsto c.ack,\ val \mapsto c.val,\ rdy \mapsto c.rdy]$$

In writing this specification, we face the same problem as in our original FIFO specification of introducing the variable q and then hiding it. In Chapter 4, we solved this problem by introducing q in a separate *InnerFIFO* module, which is instantiated by the *FIFO* module that defines the final specification. We do essentially the same thing here, except that we introduce q in a submodule

instead of in a completely separate module. All the symbols declared and defined at the point where the submodule appears can be used within it. The submodule itself can be instantiated in the containing module anywhere after it appears. (Submodules are used in the *RealTimeHourClock* and *RTMemory* specifications on pages 121 and 126 of Chapter 9.)

There is one small problem to be solved before we can write a composite specification of the FIFO—how to specify the initial predicates. It makes sense for the initial predicate of each component's specification to specify the initial values of its part of the state. However the initial condition includes the requirements $in.ack = in.rdy$ and $out.ack = out.rdy$, each of which relates the initial states of two different components. (These requirements are stated in module *InnerFIFO* by the conjuncts $InChan!Init$ and $OutChan!Init$ of the initial predicate *Init*.) There are three ways of expressing a requirement that relates the initial states of multiple components:

- Assert it in the initial conditions of all the components. Although symmetric, this seems needlessly redundant.
- Arbitrarily assign the requirement to one of the components. This intuitively suggests that we are assigning to that component the responsibility of ensuring that the requirement is met.
- Assert the requirement as a conjunct separate from either of the component specifications. This intuitively suggests that it is an assumption about how the components are put together, rather than a requirement of either component.

When we write an open-system specification, as described in Section 10.7 below, the intuitive suggestions of the last two approaches can be turned into formal requirements. I've taken the last approach and added

$$(in.ack = in.rdy) \land (out.ack = out.rdy)$$

as a separate condition. The complete specification is in module *CompositeFIFO* of Figure 10.1 on the next page. Formula *Spec* of this module is a noninterleaving specification; for example, it allows a single step that is both an $InChan!Send$ step (the sender sends a value) and an $OutChan!Rcv$ step (the receiver acknowledges a value). Hence, it is not equivalent to the interleaving specification *Spec* of the *FIFO* module on page 41, which does not allow such a step.

10.4 Composition with Shared State

Thus far, we have been considering *disjoint-state compositions*—ones in which the components are represented by disjoint parts of the state, and a compo-

MODULE *CompositeFIFO*

EXTENDS *Naturals, Sequences*
CONSTANT *Message*
VARIABLES *in, out*

$InChan \triangleq$ INSTANCE *Channel* WITH *Data* $\leftarrow$ *Message*, *chan* $\leftarrow$ *in*
$OutChan \triangleq$ INSTANCE *Channel* WITH *Data* $\leftarrow$ *Message*, *chan* $\leftarrow$ *out*

$SenderInit \triangleq (in.rdy \in \{0, 1\}) \land (in.val \in Message)$
$Sender \triangleq SenderInit \land \Box [\exists msg \in Message : InChan!Send(msg)]_{\langle in.val, in.rdy \rangle}$

The Sender's specification.

MODULE *InnerBuf*

VARIABLE *q*

$BufferInit \triangleq \land in.ack \in \{0, 1\}$
$\land q = \langle \rangle$
$\land (out.rdy \in \{0, 1\}) \land (out.val \in Message)$

The Buffer's internal specification, with *q* visible.

$BufRcv \triangleq \land InChan!Rcv$
$\land q' = Append(q, in.val)$
$\land$ UNCHANGED $\langle out.val, out.rdy \rangle$

$BufSend \triangleq \land q \neq \langle \rangle$
$\land OutChan!Send(Head(q))$
$\land q' = Tail(q)$
$\land$ UNCHANGED $in.ack$

$InnerBuffer \triangleq BufferInit \land \Box[BufRcv \lor BufSend]_{\langle in.ack, q, out.val, out.rdy \rangle}$

$Buf(q) \triangleq$ INSTANCE *InnerBuf*
$Buffer \triangleq \exists q : Buf(q)!InnerBuffer$

The Buffer's external specification with *q* hidden.

$ReceiverInit \triangleq out.ack \in \{0, 1\}$
$Receiver \triangleq ReceiverInit \land \Box[OutChan!Rcv]_{out.ack}$

The Receiver's specification.

$IsChannel(c) \triangleq c = [ack \mapsto c.ack, val \mapsto c.val, rdy \mapsto c.rdy]$

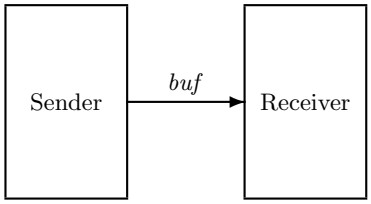
$Spec \triangleq \land \Box(IsChannel(in) \land IsChannel(out))$ Asserts that *in* and *out* are always records.
$\land (in.ack = in.rdy) \land (out.ack = out.rdy)$ Relates different components' initial states.
$\land Sender \land Buffer \land Receiver$ Conjoins the three specifications.

Figure 10.1: A noninterleaving composite specification of the FIFO.

nent's next-state action describes changes only to its part of the state.[2] We now consider the case when this may not be possible.

10.4.1 Explicit State Changes

We first examine the situation in which some part of the state cannot be partitioned among the different components, but the state change that each component performs is completely described by the specification. As an example, let's again consider a Sender and a Receiver that communicate with a FIFO buffer. In the system we studied in Chapter 4, sending or receiving a value required two steps. For example, the Sender executes a *Send* step to send a value, and it must then wait until the buffer executes a *Rcv* step before it can send another value. We simplify the system by replacing the Buffer component with a variable *buf* whose value is the sequence of values sent by the Sender but not yet received by the Receiver. This replaces the three-component system pictured on page 35 with this two-component one:



The Sender sends a value by appending it to the end of *buf*; the Receiver receives a value by removing it from the head of *buf*.

In general, the Sender performs some computation to produce the values that it sends, and the Receiver does some computation on the values that it receives. The system state consists of *buf* and two tuples s and r of variables that describe the Sender and Receiver states. In a monolithic specification, the system's next-state action is a disjunction $Sndr \lor Rcvr$, where *Sndr* and *Rcvr* describe steps taken by the Sender and Receiver, respectively. These actions are defined by

$$
\begin{array}{l}
Sndr \triangleq \\
\quad \lor \land buf' = Append(buf, \ldots) \\
\quad\quad \land SComm \\
\quad\quad \land \text{UNCHANGED } r \\
\quad \lor \land SCompute \\
\quad\quad \land \text{UNCHANGED } \langle buf, r \rangle
\end{array}
\qquad
\begin{array}{l}
Rcvr \triangleq \\
\quad \lor \land buf \neq \langle\,\rangle \\
\quad\quad \land buf' = Tail(buf) \\
\quad\quad \land RComm \\
\quad\quad \land \text{UNCHANGED } s \\
\quad \lor \land RCompute \\
\quad\quad \land \text{UNCHANGED } \langle buf, s \rangle
\end{array}
$$

[2] In an interleaving composition, a component specification may assert that the state of other components is *not* changed.

for some actions *SComm*, *SCompute*, *RComm*, and *RCompute*. For simplicity, we assume that neither *Sndr* nor *Rcvr* allows stuttering actions, so *SCompute* changes s and *RCompute* changes r. We now write the specification as the composition of separate specifications of the Sender and Receiver.

Splitting the initial predicate is straightforward. The initial conditions on s belong to the Sender's initial predicate; those on r belong to the Receiver's initial predicate; and the initial condition $buf = \langle\,\rangle$ can be assigned arbitrarily to either of them.

Now let's consider the next-state actions *NS* and *NR* of the Sender and Receiver components. The trick is to define them by

$$NS \triangleq Sndr \lor (\sigma \land (s' = s)) \qquad NR \triangleq Rcvr \lor (\rho \land (r' = r))$$

where σ and ρ are actions containing only the variable buf. Think of σ as describing possible changes to buf that are not caused by the Sender, and ρ as describing possible changes to buf that are not caused by the Receiver. Thus, *NS* permits any step that is either a *Sndr* step or one that leaves s unchanged and is a change to buf that can't be "blamed" on the Sender.

Suppose σ and ρ satisfy the following three conditions:

- $\forall\, d : (buf' = Append(buf, d)) \Rightarrow \rho$
  A step that appends a value to buf is not caused by the Receiver.
- $(buf \neq \langle\,\rangle) \land (buf' = Tail(buf)) \Rightarrow \sigma$
  A step that removes a value from the head of buf is not caused by the Sender.
- $(\sigma \land \rho) \Rightarrow (buf' = buf)$
  A step that is caused by neither the Sender nor the Receiver cannot change buf.

Using obvious relations such as[3]

$$(buf' = buf) \land (buf \neq \langle\,\rangle) \land (buf' = Tail(buf)) \equiv \text{FALSE}$$

a computation like the one by which we derived (10.1) shows

$$\Box[NS]_{\langle buf,\, s\rangle} \land \Box[NR]_{\langle buf,\, r\rangle} \equiv \Box[Sndr \lor Rcvr]_{\langle buf,\, s,\, r\rangle}$$

Thus, *NS* and *NR* are suitable next-state actions for the components, if we choose σ and ρ to satisfy the three conditions above. There is considerable freedom in that choice. The strongest possible choices of σ and ρ are ones that describe exactly the changes permitted by the other component:

$$\sigma \triangleq (buf \neq \langle\,\rangle) \land (buf' = Tail(buf))$$
$$\rho \triangleq \exists\, d : buf' = Append(buf,\, d)$$

[3]These relations are true only if buf is a sequence. A rigorous calculation requires the use of an invariant to assert that buf actually is a sequence.

We can weaken these definitions any way we want, so long as we maintain the condition that $\sigma \wedge \rho$ implies that *buf* is unchanged. For example, we can define σ as above and let ρ equal $\neg\sigma$. The choice is a matter of taste.

I've been describing an interleaving specification of the Sender/Receiver system. Now let's consider a noninterleaving specification—one that allows steps in which both the Sender and the Receiver are computing. In other words, we want the specification to allow $SCompute \wedge RCompute$ steps that leave *buf* unchanged. Let *SSndr* be the action that is the same as *Sndr* except it doesn't mention r, and let *RRcvr* be defined analogously. We then have

$$Sndr \;\equiv\; SSndr \wedge (r' = r) \qquad\qquad Rcvr \;\equiv\; RRcvr \wedge (s' = s)$$

A monolithic noninterleaving specification has the next-state action

$$Sndr \;\vee\; Rcvr \;\vee\; (SSndr \wedge RRcvr \wedge (buf' = buf))$$

It is the conjunction of component specifications having the next-state actions *NS* and *NR* defined by

$$NS \;\triangleq\; SSndr \vee (\sigma \wedge (s' = s)) \qquad\qquad NR \;\triangleq\; RRcvr \vee (\rho \wedge (r' = r))$$

where σ and ρ are as above.

This two-process situation generalizes to the composition of any set C of components that share a variable or tuple of variables w. The interleaving case generalizes to the following rule, in which N_k is the next-state action of component k, the action μ_k describes all changes to w that are attributed to some component other than k, the tuple v_k describes the private state of k, and v is the tuple formed by all the v_k:

Shared-State Composition Rule The four conditions

1. $(\forall\, k \in C : {v_k}' = v_k) \;\equiv\; (v' = v)$

   v is unchanged iff the private state v_k of every component is unchanged.

2. $\forall\, i, k \in C : N_k \wedge (i \neq k) \Rightarrow ({v_i}' = v_i)$

   The next-state action of any component k leaves the private state v_i of all other components i unchanged.

3. $\forall\, i, k \in C : N_k \wedge (w' \neq w) \wedge (i \neq k) \Rightarrow \mu_i$

   A step of any component k that changes w is a μ_i step, for any other component i.

4. $(\forall\, k \in C : \mu_k) \equiv (w' = w)$

   A step is caused by no component iff it does not change w.

imply

$$\begin{aligned}(\forall\, k \in C : I_k \wedge \Box[N_k \vee (\mu_k \wedge ({v_k}' = v_k))]_{\langle w,\, v_k\rangle}) \\ \equiv\; (\forall\, k \in C : I_k) \wedge \Box\,[\exists\, k \in C : N_k]_{\langle w,\, v\rangle}\end{aligned}$$

Assumption 2 asserts that we have an interleaving specification. If we drop that assumption, then the right-hand side of the conclusion may not be a sensible specification, since a disjunct N_k may allow steps in which a variable of some other component assumes arbitrary values. However, if each N_k correctly determines the new values of component k's private state v_k, then the left-hand side will be a reasonable specification, though possibly a noninterleaving one (and not necessarily equivalent to the right-hand side).

10.4.2 Composition with Joint Actions

Consider the linearizable memory of Chapter 5. As shown in the picture on page 45, it is a system consisting of a collection of processors, a memory, and an interface represented by the variable *memInt*. We now take it to be a two-component system, where the set of processors forms one component, called the *environment*, and the memory is the other component. Let's neglect hiding for now and consider only the internal specification, with all variables visible. We want to write the specification in the form

(10.5) $(IE \land \Box[NE]_{vE}) \land (IM \land \Box[NM]_{vM})$

where E refers to the environment component (the processors) and M to the memory component. The tuple vE of variables includes *memInt* and the variables of the environment component; the tuple vM includes *memInt* and the variables of the memory component. We must choose the formulas IE, NE, etc. so that (10.5), with internal variables hidden, is equivalent to the memory specification *Spec* of module *Memory* on page 53.

In the memory specification, communication between the environment and the memory is described by an action of the form

$$Send(p,\, d,\, memInt,\, memInt') \quad \text{or} \quad Reply(p,\, d,\, memInt,\, memInt')$$

where *Send* and *Reply* are unspecified operators declared in the *MemoryInterface* module (page 48). The specification says nothing about the actual value of *memInt*. So, not only do we not know how to split *memInt* into two parts that are each changed by only one of the components, we don't even know exactly how *memInt* changes.

The trick to writing the specification as a composition is to put the *Send* and *Reply* actions in the next-state actions of both components. We represent the sending of a value over *memInt* as a *joint action* performed by both the memory and the environment. The next-state actions have the following form:

$$NM \triangleq \exists\, p \in Proc : MRqst(p) \lor MRsp(p) \lor MInternal(p)$$
$$NE \triangleq \exists\, p \in Proc : ERqst(p) \lor ERsp(p)$$

where an $MRqst(p) \land ERqst(p)$ step represents the sending of a request by processor p (part of the environment) to the memory, an $MRsp(p) \land ERsp(p)$ step represents the sending of a reply by the memory to processor p, and an $MInternal(p)$ step is an internal step of the memory component that performs the request. (There are no internal steps of the environment.)

The sending of a reply is controlled by the memory, which chooses what value is sent and when it is sent. The enabling condition and the value sent are therefore specified by the $MRsp(p)$ action. Let's take the internal variables of the memory component to be the same variables *mem*, *ctl*, and *buf* as in the internal monolithic memory specification of module *InternalMemory* on pages 52 and 53. We can then let $MRsp(p)$ be the same as the action $Rsp(p)$ defined in that module. The $ERsp(p)$ action should always be enabled, and it should allow any legal response to be sent. A legal response is an element of *Val* or the special value *NoVal*, so we can define $ERsp(p)$ to equal[4]

$$\begin{array}{l} \land\ \exists\, rsp \in Val \cup \{NoVal\} \,:\, Reply(p,\, rsp,\, memInt,\, memInt') \\ \land\ \ldots \end{array}$$

where the "..." describes the new values of the environment's internal variables.

The sending of a request is controlled by the environment, which chooses what value is sent and when it is sent. Hence, the enabling condition should be part of the $ERqst(p)$ action. In the monolithic specification of the *InternalMemory* module, that enabling condition was $ctl[p] = \text{“rdy”}$. However, if *ctl* is an internal variable of the memory, it can't also appear in the environment specification. We therefore have to add a new variable whose value indicates whether a processor is allowed to send a new request. Let's use a Boolean variable *rdy*, where $rdy[p]$ is true iff processor p can send a request. The value of $rdy[p]$ is set false when p sends a request and is set true again when the corresponding response to p is sent. We can therefore define $ERqst(p)$, and complete the definition of $ERsp(p)$, as follows:

$$\begin{array}{lcl} ERqst(p) & \triangleq & \land\ rdy[p] \\ & & \land\ \exists\, req \in MReq \,:\, Send(p,\, req,\, memInt,\, memInt') \\ & & \land\ rdy' = [rdy \text{ EXCEPT } ![p] = \text{FALSE}] \\ ERsp(p) & \triangleq & \land\ \exists\, rsp \in Val \cup \{NoVal\} \,: \\ & & \qquad Reply(p,\, rsp,\, memInt,\, memInt') \\ & & \land\ rdy' = [rdy \text{ EXCEPT } ![p] = \text{TRUE}] \end{array}$$

The memory's $MRqst(p)$ action is the same as the $Req(p)$ action of the *InternalMemory* module, except without the enabling condition $ctl[p] = \text{“rdy”}$.

[4]The bound on the $\exists$ isn't necessary. We can let the processor accept any value, not just a legal one, by taking $\exists\, rsp : Reply(p,\, rsp,\, memInt,\, memInt')$ as the first conjunct. However, it's generally better to use bounded quantifiers when possible.

Finally, the memory's internal action $MInternal(p)$ is the same as the $Do(p)$ action of the *InternalMemory* module.

The rest of the specification is easy. The tuples vE and vM are $\langle memInt, rdy \rangle$ and $\langle memInt, mem, ctl, buf \rangle$, respectively. Defining the initial predicates IE and IM is straightforward, except for the decision of where to put the initial condition $memInt \in InitMemInt$. We can put it in either IE or IM, in both, or else in a separate conjunct that belongs to neither component's specification. Let's put it in IM, which then equals the initial predicate $IInit$ from the *InternalMemory* module. The final environment specification is obtained by hiding rdy in its internal specification; the final memory component specification is obtained by hiding mem, ctl, and buf in its internal specification. The complete specification appears in Figure 10.2 on the next page. I have not bothered to define IM, $MRsp(p)$, or $MInternal(p)$, since they equal $IInit$, $Rsp(p)$, and $Do(p)$ from the *InternalMemory* module, respectively.

What we've just done for the environment-memory system generalizes naturally to joint-action specifications of any two-component system in which part of the state cannot be considered to belong to either component. It also generalizes to systems in which any number of components share some part of the state. For example, suppose we want to write a composite specification of the linearizable memory system in which each processor is a separate component. The specification of the memory component would be the same as before. The next-state action of processor p would now be

$$ERqst(p) \lor ERsp(p) \lor OtherProc(p)$$

where $ERqst(p)$ and $ERsp(p)$ are the same as above, and an $OtherProc(p)$ step represents the sending of a request by, or a response to, some processor other than p. Action $OtherProc(p)$ represents p's participation in the joint action by which another processor q communicates with the memory component. It is defined to equal

$$\begin{aligned} \exists\, q \in Proc \setminus \{p\} : &\lor \exists\, req \in MReq : Send(q, req, memInt, memInt') \\ &\lor \exists\, rsp \in Val \cup \{NoVal\} : \\ &\qquad Reply(q, rsp, memInt, memInt') \end{aligned}$$

This example is rather silly because each processor must participate in communication actions that concern only other components. It would be better to change the interface to make $memInt$ an array, with communication between processor p and the memory represented by a change to $memInt[p]$. A sensible example would require that a joint action represent a true interaction between all the components—for example, a barrier synchronization operation in which the components wait until they are all ready and then perform a synchronization step together.

```
MODULE JointActionMemory
EXTENDS MemoryInterface
  MODULE InnerEnvironmentComponent
    VARIABLE rdy
    IE ≜ rdy = [p ∈ Proc ↦ TRUE]
    ERqst(p) ≜ ∧ rdy[p]
               ∧ ∃ req ∈ MReq : Send(p, req, memInt, memInt′)
               ∧ rdy′ = [rdy EXCEPT ![p] = FALSE]
    ERsp(p) ≜ ∧ ∃ rsp ∈ Val ∪ {NoVal} : Reply(p, rsp, memInt, memInt′)
              ∧ rdy′ = [rdy EXCEPT ![p] = TRUE]
    NE ≜ ∃ p ∈ Proc : ERqst(p) ∨ ERsp(p)
    IESpec ≜ IE ∧ □[NE]⟨memInt, rdy⟩
  MODULE InnerMemoryComponent
    EXTENDS InternalMemory
    MRqst(p) ≜ ∧ ∃ req ∈ MReq : ∧ Send(p, req, memInt, memInt′)
                                ∧ buf′ = [buf EXCEPT ![p] = req]
                                ∧ ctl′ = [ctl EXCEPT ![p] = "busy"]
               ∧ UNCHANGED mem
    NM ≜ ∃ p ∈ Proc : MRqst(p) ∨ Do(p) ∨ Rsp(p)
    IMSpec ≜ IInit ∧ □[NM]⟨memInt, mem, ctl, buf⟩
  IEnv(rdy) ≜ INSTANCE InnerEnvironmentComponent
  IMem(mem, ctl, buf) ≜ INSTANCE InnerMemoryComponent
  Spec ≜ ∧ ∃ rdy : IEnv(rdy)!IESpec
         ∧ ∃ mem, ctl, buf : IMem(mem, ctl, buf)!IMSpec
```

Figure 10.2: A joint-action specification of a linearizable memory.

10.5 A Brief Review

The basic idea of composing specifications is simple: a composite specification is the conjunction of formulas, each of which can be considered to be the specification of a separate component. This chapter has presented several techniques for writing a specification as a composition. Before going further, let's put these techniques in perspective.

10.5.1 A Taxonomy of Composition

We have seen three different ways of categorizing composite specifications:

The word *interleaving* is standard; there is no common terminology for the other concepts.

Interleaving versus noninterleaving. An interleaving specification is one in which each (nonstuttering) step can be attributed to exactly one component. A noninterleaving specification allows steps that represent simultaneous operations of two or more different components.

Disjoint-state versus shared-state. A disjoint-state specification is one in which the state can be partitioned, with each part belonging to a separate component. A part of the state can be a variable v, or a "piece" of that variable such as $v.c$ or $v[c]$ for some fixed c. Any change to a component's part of the state is attributed to that component. In a shared-state specification, some part of the state can be changed by steps attributed to more than one component.

Joint-action versus separate-action. A joint-action specification is a noninterleaving one in which some step attributed to one component must occur simultaneously with a step attributed to another component. A separate-action specification is simply one that is not a joint-action specification.

These are independent ways of classifying specifications, except that a joint-action specification must be noninterleaving.

10.5.2 Interleaving Reconsidered

Should we write interleaving or noninterleaving specifications? We might try to answer this question by asking: can different components really take simultaneous steps? However, this question makes no sense. A step is a mathematical abstraction; real components perform operations that take a finite amount of time. Operations performed by two different components could overlap in time. We are free to represent this physical situation either with a single simultaneous step of the two components, or with two separate steps. In the latter case, the specification usually allows the two steps to occur in either order. (If the two operations must occur simultaneously, then we have written a joint-action specification.) It's up to you whether to write an interleaving or a noninterleaving specification. You should choose whichever is more convenient.

The choice is not completely arbitrary if you want one specification to implement another. A noninterleaving specification will not, in general, implement an interleaving one because the noninterleaving specification will allow simultaneous actions that the interleaving specification prohibits. So, if you want to write a lower-level specification that implements a higher-level interleaving specification, then you'll have to use an interleaving specification. As we've seen, it's easy

to turn a noninterleaving specification into an interleaving one by conjoining an interleaving assumption.

10.5.3 Joint Actions Reconsidered

The reason for writing a composite specification is to separate the specifications of the different components. The mixing of actions from different components in a joint-action specification destroys this separation. So, why should we write such a specification?

Joint-action specifications arise most often in highly abstract descriptions of inter-component communication. In writing a composite specification of the linearizable memory, we were led to use joint actions because of the abstract nature of the interface. In real systems, communication occurs when one component changes the state and another component later observes that change. The interface described by the *MemoryInterface* module abstracts away those two steps, replacing them with a single one that represents instantaneous communication—a fiction that does not exist in the real world. Since each component must remember that the communication has occurred, the single communication step has to change the private state of both components. That's why we couldn't use the approach of Section 10.4.1 (page 144), which requires that any change to the shared interface change the nonshared state of just one component.

The abstract memory interface simplifies the specification, allowing communication to be represented as one step instead of two. But this simplification comes at the cost of blurring the distinction between the two components. If we blur this distinction, it may not make sense to write the specification as the conjunction of separate component specifications. As the memory system example illustrates, decomposing the system into separate components communicating with joint actions may require the introduction of extra variables. There may occasionally be a good reason for adding this kind of complexity to a specification, but it should not be done as a matter of course.

10.6 Liveness and Hiding

10.6.1 Liveness and Machine Closure

Thus far, the discussion of composition has neglected liveness. In composite specifications, it is usually easy to specifying liveness by placing fairness conditions on the actions of individual components. For example, to specify an array of clocks that all keep ticking forever, we would modify the specification

ClockArray of (10.2) on page 139 to equal

$$\forall\, k \in \mathit{Clock} :\\ (hr[k] \in 1 \,..\, 12) \;\wedge\; \Box[\mathit{HCN}(hr[k])]_{hr[k]} \;\wedge\; \mathrm{WF}_{hr[k]}(\mathit{HCN}(hr[k]))$$

When writing a weak or strong fairness formula for an action A of a component c, there arises the question of what the subscript should be. The obvious choices are (i) the tuple v describing the entire specification state, and (ii) the tuple v_c describing that component's state. The choice can matter only if the safety part of the specification allows the system to reach some state in which an A step could leave v_c unchanged while changing v. Although unlikely, this could conceivably be the case in a joint-action specification. If it is, we probably don't want the fairness condition to be satisfied by a step that leaves the component's state unchanged, so we would use the subscript v_c.

Fairness conditions for composite specifications do raise one important question: if each component specification is machine closed, is the composite specification necessarily machine closed? Suppose we write the specification as $\forall\, k \in C : S_k \wedge L_k$, where each pair S_k, L_k is machine closed. Let S be the conjunction of the S_k and L the conjunction of the L_k, so the specification equals $S \wedge L$. The conjunction of safety properties is a safety property,[5] so S is a safety property. Hence, we can ask if the pair S, L is machine closed.

Machine closure is defined in Section 8.9.2 on page 111.

In general, S, L need not be machine closed. But, for an interleaving composition, it usually is. Liveness properties are usually expressed as the conjunction of weak and strong fairness properties of actions. As stated on page 112, a specification is machine closed if its liveness property is the conjunction of fairness properties for subactions of the next-state action. In an interleaving composition, each S_k usually has the form $I_k \wedge \Box[N_k]_{v_k}$ where the v_k satisfy the hypothesis of the Composition Rule (page 138), and each N_k implies ${v_i}' = v_i$, for all i in $C \setminus \{k\}$. In this case, the Composition Rule implies that a subaction of N_k is also a subaction of the next-state action of S. Hence, if we write an interleaving composition in the usual way, and we write machine-closed component specifications in the usual way, then the composite specification is machine closed.

It is not so easy to obtain a machine-closed noninterleaving composition—especially with a joint-action composition. We have actually seen an example of a joint-action specification in which each component is machine closed but the composition is not. In Chapter 9, we wrote a real-time specification by conjoining one or more *RTBound* formulas and an *RTnow* formula to an untimed specification. A pathological example was the following, which is formula (9.2) on page 131:

$$\mathit{HC} \;\wedge\; \mathit{RTBound}(hr' = hr - 1,\, hr,\, 0,\, 3600) \;\wedge\; \mathit{RTnow}(hr)$$

HC is the hour-clock specification from Chapter 2.

[5] Recall that a safety property is one that is violated by a behavior iff it is violated at some particular point in the behavior. A behavior violates a conjunction of safety properties S_k iff it violates some particular S_k, and that S_k is violated at some specific point.

We can view this formula as the conjunction of three component specifications:

1. HC specifies a clock, represented by the variable hr.
2. $RTBound(hr' = hr - 1,\ hr,\ 0,\ 3600)$ specifies a timer, represented by the hidden (existentially quantified) timer variable.
3. $RTnow(hr)$ specifies real time, represented by the variable now.

The formula is a joint-action composition, with two kinds of joint actions:

- Joint actions of the first and second components that change both hr and the timer.
- Joint actions of the second and third components that change both the timer and now.

The first two specifications are trivially machine closed because they assert no liveness condition, so their liveness property is TRUE. The third specification's safety property asserts that now is a real number that is changed only by steps that increment it and leave hr unchanged; its liveness property NZ asserts that now increases without bound. Any finite behavior satisfying the safety property can easily be extended to an infinite behavior satisfying the entire specification, so the third specification is also machine closed. However, as we observed in Section 9.4, the composite specification is Zeno, meaning that it's not machine closed.

10.6.2 Hiding

Suppose we can write a specification S as the composition of two component specifications S_1 and S_2. Can we write $\exists\, h : S$, the specification S with variable h hidden, as a composition—that is, as the conjunction of two separate component specifications? If h represents state that is accessed by both components, then the answer is no. If the two components communicate through some part of the state, then that part of the state cannot be made internal to the separate components.

The simplest situation in which h doesn't represent shared state is when it occurs in only one of the component specifications—say, S_2. If h doesn't occur in S_1, then the equivalence

$$(\exists\, h\ :\ S_1 \wedge S_2) \ \equiv\ S_1 \wedge (\exists\, h\ :\ S_2)$$

provides the desired decomposition.

Now suppose that h occurs in both component specifications, but does not represent state accessed by both components. This can be the case only if different "parts" of h occur in the two component specifications. For example,

h might be a record with components $h.c1$ and $h.c2$, where S_1 mentions only $h.c1$ and S_2 mentions only $h.c2$. In this case, we have

$$(\exists\, h \,:\, S_1 \wedge S_2) \;\equiv\; (\exists\, h1 \,:\, T_1) \wedge (\exists\, h2 \,:\, T_2)$$

where T_1 is obtained from S_1 by substituting the variable $h1$ for the expression $h.c1$, and T_2 is defined similarly. Of course we can use any variables in place of $h1$ and $h2$; in particular, we can replace them both by the same variable.

We can generalize this result as follows to the composition of any finite number[6] of components:

> **Compositional Hiding Rule** If the variable h does not occur in the formula T_i, and S_i is obtained from T_i by substituting $h[i]$ for q, then
>
> $$(\exists\, h \,:\, \forall\, i \in C \,:\, S_i) \;\equiv\; (\forall\, i \in C \,:\, \exists\, q \,:\, T_i)$$
>
> for any finite set C.

The assumption that h does not occur in T_i means that the variable h occurs in formula S_i only in the expression $h[i]$. This in turn implies that the composition $\forall\, i \in C : S_i$ does not determine the value of h, just of its components $h[i]$ for $i \in C$. As noted in Section 10.2 on page 138, we can make the composite specification determine the value of h by conjoining the formula $\Box \mathit{IsFcnOn}(h, C)$ to it, where *IsFcnOn* is defined on page 140. The hypotheses of the Compositional Hiding Rule imply

$$(\exists\, h \,:\, \Box \mathit{IsFcnOn}(h, C) \wedge \forall\, i \in C \,:\, S_i) \;\equiv\; (\forall\, i \in C \,:\, \exists\, q \,:\, T_i)$$

Now consider the common case in which $\forall\, i \in C : S_i$ is an interleaving composition, where each specification S_i describes changes to $h[i]$ and asserts that steps of component i leave $h[j]$ unchanged for $j \neq i$. We cannot apply the Compositional Hiding Rule because S_i must mention other components of h besides $h[i]$. For example, it probably contains an expression of the form

$$h' = [h \text{ EXCEPT } ![i] = \mathit{exp}] \tag{10.6}$$

which mentions all of h. However, we can transform S_i into a specification $\widehat{S_i}$ that describes only the changes to $h[i]$ and makes no assertions about other components. For example, we can replace (10.6) with $h'[i] = \mathit{exp}$, and we can replace an assertion that h is unchanged by the assertion that $h[i]$ is unchanged. The composition $\forall\, i \in C : \widehat{S_i}$ may allow steps that change two different components $h[i]$ and $h[j]$, while leaving all other variables unchanged, making it a noninterleaving specification. It will then not be equivalent to $\forall\, i \in C : S_i$, which requires that the changes to $h[i]$ and $h[j]$ be performed by different steps. However, it can be shown that hiding h hides this difference, making the two specifications equivalent. We can then apply the Compositional Hiding Rule with S_i replaced by $\widehat{S_i}$.

[6] The Compositional Hiding Rule is not true in general if C is an infinite set; but the examples in which it doesn't hold are pathological and don't arise in practice.

10.7 Open-System Specifications

A specification describes the interaction between a system and its environment. For example, the FIFO buffer specification of Chapter 4 specifies the interaction between the buffer (the system) and an environment consisting of the sender and receiver. So far, all the specifications we have written have been complete-system specifications, meaning that they are satisfied by a behavior that represents the correct operation of both the system and its environment. When we write such a specification as the composition of an environment specification E and a system specification M, it has the form $E \wedge M$.

Open-system specifications are sometimes called *rely-guarantee* or *assume-guarantee* specifications.

An open-system specification is one that can serve as a contract between a user of the system and its implementer. An obvious choice of such a specification is the formula M that describes the correct behavior of the system component by itself. However, such a specification is unimplementable. It asserts that the system acts correctly no matter what the environment does. A system cannot behave as expected in the face of arbitrary behavior of its environment. It would be impossible to build a buffer that satisfies the buffer component's specification regardless of what the sender and receiver did. For example, if the sender sends a value before the previous value has been acknowledged, then the buffer could read the value while it is changing, causing unpredictable results.

A contract between a user and an implementer should require the system to act correctly only if the environment does. If M describes correct behavior of the system and E describes correct behavior of the environment, such a specification should require that M be true if E is. This suggests that we take as our open-system specification the formula $E \Rightarrow M$, which is true if the system behaves correctly or the environment behaves incorrectly. However, $E \Rightarrow M$ is too weak a specification for the following reason. Consider again the example of a FIFO buffer, where M describes the buffer and E the sender and receiver. Suppose now that the buffer sends a new value before the receiver has acknowledged the previous one. This could cause the receiver to act incorrectly, possibly modifying the output channel in some way not allowed by the receiver's specification. This situation is described by a behavior in which both E and M are false—a behavior that satisfies the specification $E \Rightarrow M$. However, the buffer should not be considered to act correctly in this case, since it was the buffer's error that caused the receiver to act incorrectly. Hence, this behavior should not satisfy the buffer's specification.

An open-system specification should assert that the system behaves correctly at least as long as the environment does. To express this, we introduce a new temporal operator $\stackrel{+}{\triangleright}$, where $E \stackrel{+}{\triangleright} M$ asserts that M remains true at least one step longer than E does, remaining true forever if E does. Somewhat more precisely, $E \stackrel{+}{\triangleright} M$ asserts that

- E implies M.

- If the safety property of E is not violated by the first n states of a behavior, then the safety property of M is not violated by the first $n+1$ states, for any natural number n. (Recall that a safety property is one that, if violated, is violated at some definite point in the behavior.)

A more precise definition of $\stackrel{+}{\triangleright}$ appears in Section 16.2.4 (page 314). If E describes the desired behavior of the environment and M describes the desired behavior of the system, then we take as our open-system specification the formula $E \stackrel{+}{\triangleright} M$.

Once we write separate specifications of the components, we can usually transform a complete-system specification into an open-system one by simply replacing conjunction with $\stackrel{+}{\triangleright}$. This requires first deciding whether each conjunct of the complete-system specification belongs to the specification of the environment, of the system, or of neither. As an example, consider the composite specification of the FIFO buffer in module *CompositeFIFO* on page 143. We take the system to consist of just the buffer, with the sender and receiver forming the environment. The closed-system specification *Spec* has three main conjuncts:

$Sender \wedge Buffer \wedge Receiver$
: The conjuncts *Sender* and *Receiver* are clearly part of the environment specification, and *Buffer* is part of the system specification.

$(in.ack = in.rdy) \wedge (out.ack = out.rdy)$
: These two initial conjuncts can be assigned to either, depending on which component we want to blame if they are violated. Let's assign to the component sending on a channel c the responsibility for establishing that $c.ack = c.rdy$ holds initially. We then assign $in.ack = in.rdy$ to the environment and $out.ack = out.rdy$ to the system.

$\Box(IsChannel(in) \wedge IsChannel(out))$
: This formula is not naturally attributed to either the system or the environment. We regard it as a property inherent in our way of modeling the system, which assumes that *in* and *out* are records with *ack*, *val*, and *rdy* components. We therefore take the formula to be a separate conjunct of the complete specification, not belonging to either the system or the environment.

We then have the following open-system specification for the FIFO buffer:

$$\begin{array}{l} \wedge\ \Box(IsChannel(in)\ \wedge\ IsChannel(out)) \\ \wedge\ (in.ack = in.rdy)\ \wedge\ Sender\ \wedge\ Receiver\ \stackrel{+}{\triangleright} \\ \qquad (out.ack = out.rdy)\ \wedge\ Buffer \end{array}$$

As this example suggests, there is little difference between writing a composite complete-system specification and an open-system specification. Most of the

specification doesn't depend on which we choose. The two differ only at the very end, when we put the pieces together.

10.8 Interface Refinement

An *interface refinement* is a method of obtaining a lower-level specification by refining the variables of a higher-level specification. Let's start with two examples and then discuss interface refinement in general.

10.8.1 A Binary Hour Clock

In specifying an hour clock, we described its display with a variable hr whose value (in a behavior satisfying the specification) is an integer from 1 to 12. Suppose we want to specify a *binary hour clock*. This is an hour clock for use in a computer, where the display consists of a four-bit register that displays the hour as one of the twelve values 0001, 0010, ..., 1100. We can easily specify such a clock from scratch. But suppose we want to describe it informally to someone who already knows what an hour clock is. We would simply say that a binary hour clock is the same as an ordinary hour clock, except that the value of the display is represented in binary. We now formalize that description.

We begin by describing what it means for a four-bit value to represent a number. There are several reasonable ways to represent a four-bit value mathematically. We could use a four-element sequence, which in TLA$^+$ is a function whose domain is $1 \,..\, 4$. However, a mathematician would find it more natural to represent an $(n+1)$-bit number as a function from $0 \,..\, n$ to $\{0, 1\}$, the function b representing the number $b[0] * 2^0 + b[1] * 2^1 + \ldots + b[n] * 2^n$. In TLA$^+$, we can define $BitArrayVal(b)$ to be the numerical value of such a function b by

We can also write $\{0, 1\}$ as $0 \,..\, 1$.

$$
\begin{array}{l}
BitArrayVal(b) \triangleq \\
\quad \text{LET } n \triangleq \text{CHOOSE } i \in Nat : \text{DOMAIN } b = 0 \,..\, i \\
\qquad val[i \in 0 \,..\, n] \triangleq \quad \text{Defines } val[i] \text{ to equal } b[0] * 2^0 + \ldots + b[i] * 2^i. \\
\qquad\quad \text{IF } i = 0 \text{ THEN } b[0] * 2^0 \text{ ELSE } b[i] * 2^i + val[i-1] \\
\quad \text{IN } val[n]
\end{array}
$$

To specify a binary hour clock whose display is described by the variable $bits$, we would simply say that $BitArrayVal(bits)$ changes the same way that the specification HC of the hour clock allows hr to change. Mathematically, this means that we obtain the specification of the binary hour clock by substituting $BitArrayVal(bits)$ for the variable hr in HC. In TLA$^+$, substitution is expressed with the INSTANCE statement. Writing

$$B \triangleq \text{INSTANCE } HourClock \text{ WITH } hr \leftarrow BitArrayVal(bits)$$

defines (among other things) $B!HC$ to be the formula obtained from HC by substituting $BitArrayVal(bits)$ for hr.

Unfortunately, this specification is wrong. The value of $BitArrayVal(b)$ is specified only if b is a function with domain $0 \,..\, n$ for some natural number n. We don't know what $BitArrayVal(\{\text{“abc”}\})$ equals. It might equal 7. If it did, then $B!HC$ would allow a behavior in which the initial value of $bits$ is $\{\text{“abc”}\}$. We must rule out this possibility by substituting for hr not $BitArrayVal(bits)$, but some expression $HourVal(bits)$ whose value is an element of $1 \,..\, 12$ only if b is a function in $[(0 \,..\, 3) \to \{0, 1\}]$. For example, we can write

$$HourVal(b) \triangleq \text{IF } b \in [(0 \,..\, 3) \to \{0, 1\}] \text{ THEN } BitArrayVal(b) \text{ ELSE } 99$$

$$B \triangleq \text{INSTANCE } HourClock \text{ WITH } hr \leftarrow HourVal(bits)$$

This defines $B!HC$ to be the desired specification of the binary hour clock. Because HC is not satisfied by a behavior in which hr ever assumes the value 99, $B!HC$ is not satisfied by any behavior in which $bits$ ever assumes a value not in the set $[(0 \,..\, 3) \to \{0, 1\}]$.

There is another way to use the specification HC of the hour clock to specify the binary hour clock. Instead of substituting for hr in the hour-clock specification, we first specify a system consisting of both an hour clock and a binary hour clock that keep the same time, and we then hide the hour clock. This specification has the form

(10.7) $\exists\, hr : IR \wedge HC$

where IR is a temporal formula that is true iff $bits$ is always the four-bit value representing the value of hr. This formula asserts that $bits$ is the representation of hr as a four-bit array, for some choice of values for hr that satisfies HC. Using the definition of $HourVal$ given above, we can define IR simply to equal $\Box(h = HourVal(b))$.

If HC is defined as in module $HourClock$, then (10.7) can't appear in a TLA$^+$ specification. For HC to be defined in the context of the formula, the variable hr must be declared in that context. If hr is already declared, then it can't be used as the bound variable of the quantifier $\exists$. As usual, this problem is solved with parametrized instantiation. The complete TLA$^+$ specification BHC of the binary hour clock appears in module $BinaryHourClock$ of Figure 10.3 on the next page.

10.8.2 Refining a Channel

As our second example of interface refinement, consider a system that interacts with its environment by sending numbers from 1 through 12 over a channel. We refine it to a lower-level system that is the same, except it sends a number

```
MODULE BinaryHourClock
EXTENDS Naturals
VARIABLE bits
H(hr) ≜ INSTANCE HourClock
BitArrayVal(b) ≜ LET n ≜ CHOOSE i ∈ Nat : DOMAIN b = 0 .. i
                     val[i ∈ 0 .. n] ≜ Defines val[i] to equal b[0] * 2^0 + ... + b[i] * 2^i.
                       IF i = 0 THEN b[0] * 2^0 ELSE (b[i] * 2^i) + val[i − 1]
                 IN  val[n]
HourVal(b) ≜ IF b ∈ [(0 .. 3) → {0, 1}] THEN BitArrayVal(b)
                                        ELSE 99
IR(b, h) ≜ □(h = HourVal(b))
BHC ≜ ∃ hr : IR(bits, hr) ∧ H(hr)!HC
```

Figure 10.3: A specification of a binary hour clock.

as a sequence of four bits. Each bit is sent separately, starting with the leftmost (most significant) one. For example, to send the number 5, the lower-level system sends the sequence of bits 0, 1, 0, 1. We specify both channels with the *Channel* module of Figure 3.2 on page 30, so each value that is sent must be acknowledged before the next one can be sent.

Suppose *HSpec* is the higher-level system's specification, and its channel is represented by the variable h. Let l be the variable representing the lower-level channel. We write the lower-level system's specification as

(10.8) $\exists\, h : IR \wedge HSpec$

where IR specifies the sequence of values sent over h as a function of the values sent over l. The sending of the fourth bit on l is interpreted as the sending of the complete number on h; the next acknowledgment on l is interpreted as the sending of the acknowledgment on h; and any other step is interpreted as a step that doesn't change h.

To define IR, we instantiate module *Channel* for each of the channels:

Data is the set of values that can be sent over the channel.

$$H \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow h,\ Data \leftarrow 1 \,..\, 12$$
$$L \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow l,\ Data \leftarrow \{0, 1\}$$

Sending a value d over channel l is thus represented by an $L!Send(d)$ step, and acknowledging receipt of a value on channel h is represented by an $H!Rcv$ step. The following behavior represents sending and acknowledging a 5, where I have

omitted all steps that don't change l:

$$s_0 \xrightarrow{L!Send(0)} s_1 \xrightarrow{L!Rcv} s_2 \xrightarrow{L!Send(1)} s_3 \xrightarrow{L!Rcv} s_4 \xrightarrow{L!Send(0)} s_5 \xrightarrow{L!Rcv} s_6 \xrightarrow{L!Send(1)} s_7 \xrightarrow{L!Rcv} s_8 \longrightarrow \cdots$$

This behavior will satisfy *IR* iff $s_6 \to s_7$ is an $H!Send(5)$ step, $s_7 \to s_8$ is an $H!Rcv$ step, and all the other steps leave h unchanged.

We want to make sure that (10.8) is not satisfied unless l represents a correct lower-level channel—for example, (10.8) should be false if l is set to some bizarre value. We will therefore define *IR* so that, if the sequence of values assumed by l does not represent a channel over which bits are sent and acknowledged, then the sequence of values of h does not represent a correct behavior of a channel over which numbers from 1 to 12 are sent. Formula *HSpec*, and hence (10.8), will then be false for such a behavior.

Formula *IR* will have the standard form for a TLA specification, with an initial condition and a next-state action. However, it specifies h as a function of l; it does not constrain l. Therefore, the initial condition does not specify the initial value of l, and the next-state action does not specify the value of l'. (The value of l is constrained implicitly by *IR*, which asserts a relation between the values of h and l, together with the conjunct *HSpec* in (10.8), which constrains the value of h.) For the next-state action to specify the value sent on h, we need an internal variable that remembers what has been sent on l since the last complete number. We let the variable *bitsSent* contain the sequence of bits sent so far for the current number. For convenience, *bitsSent* contains the sequence of bits in reverse order, with the most recently-sent bit at the head. This means that the high-order bit of the number, which is sent first, is at the tail of *bitsSent*.

The definition of *IR* appears in module *ChannelRefinement* of Figure 10.4 on the next page. The module first defines *ErrorVal* to be an arbitrary value that is not a legal value of h. Next comes the definition of the function *BitSeqToNat*. If s is a sequence of bits, then $BitSeqToNat[s]$ is its numeric value interpreted as a binary number whose low-order bit is at the head of s. For example $BitSeqToNat[\langle 0, 1, 1\rangle]$ equals 6. Then come the two instantiations of module *Channel*.

There follows a submodule that defines the internal specification—the one with the internal variable *bitsSent* visible. The internal specification's initial predicate *Init* asserts that if l has a legal initial value, then h can have any legal initial value; otherwise h has an illegal value. Initially *bitsSent* is the empty sequence $\langle\,\rangle$. The internal specification's next-state action is the disjunction of three actions:

The use of a submodule to define an internal specification was introduced in the real-time hour-clock specification of Section 9.1.

SendBit A *SendBit* step is one in which a bit is sent on l. If *bitsSent* has fewer than three elements, so fewer than three bits have already been sent, then the bit is prepended to the head of *bitsSent* and h

MODULE *ChannelRefinement*

This module defines an interface refinement from a higher-level channel h, over which numbers in 1 .. 12 are sent, to a lower-level channel l in which a number is sent as a sequence of four bits, each separately acknowledged. (See the *Channel* module in Figure 3.2 on page 30.) Formula *IR* is true iff the sequence of values assumed by h represents the higher-level view of the sequence of values sent on l. If the sequence of values assumed by l doesn't represent the sending and acknowledging of bits, then h assumes an illegal value.

EXTENDS *Naturals*, *Sequences*

VARIABLES h, l

$ErrorVal \triangleq$ CHOOSE $v : v \notin [val : 1 \,..\, 12,\ rdy : \{0,1\},\ ack : \{0,1\}]$

$BitSeqToNat[s \in Seq(\{0,1\})] \triangleq$ IF $s = \langle\,\rangle$ THEN 0 ELSE $Head(s) + 2 * BitSeqToNat[Tail(s)]$

BitSeqToNat$[\langle b_0,\ b_1,\ b_2,\ b_3\rangle] = b_0 + 2 * (b_1 + 2 * (b_2 + 2 * b_3))$

$H \triangleq$ INSTANCE *Channel* WITH *chan* $\leftarrow h$, *Data* $\leftarrow 1 \,..\, 12$

$L \triangleq$ INSTANCE *Channel* WITH *chan* $\leftarrow l$, *Data* $\leftarrow \{0,1\}$

H is a channel for sending numbers in 1 .. 12; L is a channel for sending bits.

MODULE *Inner*

VARIABLE *bitsSent* The sequence of the bits sent so far for the current number.

$Init \triangleq \land\ bitsSent = \langle\,\rangle$
$\land$ IF $L!Init$ THEN $H!Init$ ELSE $h = ErrorVal$

Defines the initial value of h as a function of l.

$SendBit \triangleq \exists\, b \in \{0,1\} :$
$\land\ L!Send(b)$
$\land$ IF $Len(bitsSent) < 3$
THEN $\land\ bitsSent' = \langle b\rangle \circ bitsSent$
$\land$ UNCHANGED h
ELSE $\land\ bitsSent' = \langle\,\rangle$
$\land\ H!Send(BitSeqToNat[\langle b\rangle \circ bitsSent])$

Sending one of the first three bits on l prepends it to the front of *bitsSent* and leaves h unchanged; sending the fourth bit resets *bitsSent* and sends the complete number on h.

$RcvBit \triangleq \land\ L!Rcv$
$\land$ IF $bitsSent = \langle\,\rangle$ THEN $H!Rcv$ ELSE UNCHANGED h
$\land$ UNCHANGED *bitsSent*

A *Rcv* action on l causes a *Rcv* action on h iff it follows the sending of the fourth bit.

$Error \triangleq \land\ l' \neq l$
$\land\ \neg((\exists\, b \in \{0,1\} : L!Send(b)) \lor L!Rcv)$
$\land\ h' = ErrorVal$

An illegal action on l sets h to *ErrorVal*.

$Next \triangleq SendBit \lor RcvBit \lor Error$

$InnerIR \triangleq Init \land \Box[Next]_{\langle l,\, h,\, bitsSent\rangle}$

$I(bitsSent) \triangleq$ INSTANCE *Inner*

$IR \triangleq \exists\, bitsSent : I(bitsSent)!InnerIR$

Figure 10.4: Refining a channel.

is left unchanged. Otherwise, the value represented by the four bits sent so far, including the current bit, is sent on h and *bitsSent* is reset to $\langle\,\rangle$.

RcvBit A *RcvBit* step is one in which an acknowledgment is sent on l. It represents the sending of an acknowledgment on h iff this is an acknowledgment of the fourth bit, which is true iff *bitsSent* is the empty sequence.

Error An *Error* step is one in which an illegal change to l occurs. It sets h to an illegal value.

The inner specification *InnerIR* has the usual form. (There is no liveness requirement.) The outer module then instantiates the inner submodule with *bitsSent* as a parameter, and it defines *IR* to equal *InnerIR* with *bitsSent* hidden.

Now suppose we have a module *HigherSpec* that defines a specification *HSpec* of a system that communicates by sending numbers from 1 through 12 over a channel *hchan*. We obtain, as follows, a lower-level specification *LSpec* in which the numbers are sent as sequences of bits on a channel *lchan*. We first declare *lchan* and all the variables and constants of the *HigherSpec* module except *hchan*. We then write

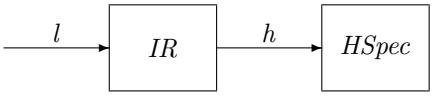
$$\begin{aligned} &HS(hchan) \triangleq \text{INSTANCE } HigherSpec \\ &CR(h) \triangleq \text{INSTANCE } ChannelRefinement \text{ WITH } l \leftarrow lchan \\ &LSpec \triangleq \exists\, h : CR(h)!IR \wedge HS(h)!HSpec \end{aligned}$$

10.8.3 Interface Refinement in General

In the examples of the binary clock and of channel refinement, we defined a lower-level specification *LSpec* in terms of a higher-level one *HSpec* as

$$LSpec \triangleq \exists\, h : IR \wedge HSpec \tag{10.9}$$

where h is a free variable of *HSpec* and *IR* is a relation between h and the lower-level variable l of *LSpec*. We can view the internal specification $IR \wedge HSpec$ as the composition of two components, as shown here:



We can regard *IR* as the specification of a component that transforms the lower-level behavior of l into the higher-level behavior of h. Formula *IR* is called an *interface refinement*.

In both examples, the interface refinement is independent of the system specification. It depends only on the representation of the interface—that is, on how the interaction between the system and its environment is represented. In general, for an interface refinement IR to be independent of the system using the interface, it should ascribe a behavior of the higher-level interface variable h to any behavior of the lower-level variable l. In other words, for any sequence of values for l, there should be some sequence of values for h that satisfy IR. This is expressed mathematically by the requirement that the formula $\exists\, h : IR$ should be valid—that is, true for all behaviors.

So far, I have discussed refinement of a single interface variable h by a single variable l. This generalizes in the obvious way to the refinement of a collection of higher-level variables $h_1, \ldots, h_n$ by the variables $l_1, \ldots, l_m$. The interface refinement IR specifies the values of the h_i in terms of the values of the l_j and perhaps of other variables as well. Formula (10.9) is replaced by

$$LSpec \triangleq \exists\, h_1, \ldots, h_n : IR \wedge HSpec$$

A particularly simple type of interface refinement is a *data refinement*, in which IR has the form $\Box P$, where P is a state predicate that expresses the values of the higher-level variables $h_1, \ldots, h_n$ as functions of the values of the lower-level variables $l_1, \ldots, l_m$. The interface refinement in our binary clock specification is a data refinement, where P is the predicate $hr = HourVal(bits)$. As another example, the two specifications of an asynchronous channel interface in Chapter 3 can each be obtained from the other by an interface refinement. The specification *Spec* of the *Channel* module (page 30) is equivalent to the specification obtained as a data refinement of the specification *Spec* of the *AsynchInterface* module (page 27) by letting P equal

$$(10.10)\quad chan = [val \mapsto val,\ rdy \mapsto rdy,\ ack \mapsto ack]$$

This formula asserts that *chan* is a record whose *val* field is the value of the variable *val*, whose *rdy* field is the value of the variable *rdy*, and whose *ack* field is the value of the variable *ack*. Conversely, specification *Spec* of the *AsynchInterface* module is equivalent to a data refinement of the specification *Spec* of the *Channel* module. In this case, defining the state predicate P is a little tricky. The obvious choice is to let P be the formula *GoodVals* defined by

$$\begin{aligned} GoodVals \triangleq\ & \wedge\ val = chan.val \\ & \wedge\ rdy = chan.rdy \\ & \wedge\ ack = chan.ack \end{aligned}$$

However, this can assert that *val*, *rdy*, and *ack* have good values even if *chan* has an illegal value—for example, if it is a record with more than three fields. Instead, we let P equal

$$\text{IF}\ \ chan \in [val : Data,\ rdy : \{0,1\},\ ack : \{0,1\}]\ \ \text{THEN}\ \ GoodVals \quad \text{ELSE}\ \ BadVals$$

where *BadVals* asserts that *val*, *rdy*, and *ack* have some illegal values—that is, values that are impossible in a behavior satisfying formula *Spec* of module *AsynchInterface*. (We don't need such a trick when defining *chan* as a function of *val*, *rdy*, and *ack* because (10.10) implies that the value of *chan* is legal iff the values of all three variables *val*, *rdy*, and *ack* are legal.)

Data refinement is the simplest form of interface refinement. In a more complicated interface refinement, the value of the higher-level variables cannot be expressed as a function of the current values of the lower-level variables. In the channel refinement example of Section 10.8.2, the number being sent on the higher-level channel depends on the values of bits that were previously sent on the lower-level channel, not just on the lower-level channel's current state.

We often refine both a system and its interface at the same time. For example, we may implement a specification H of a system that communicates by sending numbers over a channel with a lower-level specification *LImpl* of a system that sends individual bits. In this case, *LImpl* is not itself obtained from *HSpec* by an interface refinement. Rather, *LImpl* implements some specification *LSpec* that is obtained from *HSpec* by an interface refinement *IR*. In that case, we say that *LImpl* implements *HSpec under the interface refinement IR*.

10.8.4 Open-System Specifications

So far, we have considered interface refinement for complete-system specifications. Let's now consider what happens if the higher-level specification *HSpec* is the kind of open-system specification discussed in Section 10.7 above. For simplicity, we consider the refinement of a single higher-level interface variable h by a single lower-level variable l. The generalization to more variables will be obvious.

Let's suppose first that *HSpec* is a safety property, with no liveness condition. As explained in Section 10.7, the specification attributes each change to h either to the system or to the environment. Any change to a lower-level interface variable l that produces a change to h is therefore attributed to the system or the environment. A bad change to h that is attributed to the environment makes *HSpec* true; a bad change that is attributed to the system makes *HSpec* false. Thus, (10.9) defines *LSpec* to be an open-system specification. For this to be a sensible specification, the interface refinement *IR* must ensure that the way changes to l are attributed to the system or environment is sensible.

If *HSpec* contains liveness conditions, then interface refinement can be more subtle. Suppose *IR* is the interface refinement defined in the *ChannelRefinement* module of Figure 10.4 on page 162, and suppose that *HSpec* requires that the system eventually send some number on h. Consider a behavior in which the system sends the first bit of a number on l, but the environment never acknowledges it. Under the interface refinement *IR*, this behavior is interpreted as one in which h never changes. Such a behavior fails to satisfy the liveness condition

of *HSpec*. Thus, if *LSpec* is defined by (10.9), then failure of the environment to do something can cause *LSpec* to be violated, through no fault of the system.

In this example, we want the environment to be at fault if it causes the system to halt by failing to acknowledge any of the first three bits of a number sent by the system. (The acknowledgment of the fourth bit is interpreted by *IR* as the acknowledgment of a value sent on h, so blame for its absence is properly assigned to the environment.) Putting the environment at fault means making *LSpec* true. We can achieve this by modifying (10.9) to define *LSpec* as follows:

$$(10.11)\quad LSpec \triangleq Liveness \Rightarrow \exists\, h : IR \wedge HSpec$$

where *Liveness* is a formula requiring that any bit sent on l, other than the last bit of a number, must eventually be acknowledged. However, if l is set to an illegal value, then we want the safety part of the specification to determine who is responsible. So, we want *Liveness* to be true in this case.

We define *Liveness* in terms of the inner variables h and *bitsSent*, which are related to l by formula *InnerIR* from the *Inner* submodule of module *ChannelRefinement*. (Remember that l should be the only free variable of *LSpec*.) The action that acknowledges receipt of one of the first three bits of the number is $RcvBit \wedge (bitsSent \neq \langle\,\rangle)$. Weak fairness of this action asserts that the required acknowledgments must eventually be sent. For the case of illegal values, recall that sending a bad value on l causes h to equal *ErrorVal*. We want *Liveness* to be true if this ever happens, which means if it eventually happens. We therefore add the following definition to the submodule *Inner* of the *ChannelRefinement* module:

$$\begin{aligned} InnerLiveness \triangleq\ & \wedge\ InnerIR \\ & \wedge\ \vee\ \mathrm{WF}_{\langle l,\, h,\, bitsSent\rangle}(RcvBit \wedge (bitsSent \neq \langle\,\rangle)) \\ & \quad\ \vee\ \Diamond(h = ErrorVal) \end{aligned}$$

To define *Liveness*, we have to hide h and *bitsSent* in *InnerLiveness*. We can do this, in a context in which l is declared, as follows:

$$\begin{aligned} ICR(h) &\triangleq \text{INSTANCE } ChannelRefinement \\ Liveness &\triangleq \exists\, h,\, bitsSent : ICR(h)!I(bitsSent)!InnerLiveness \end{aligned}$$

Now, suppose it is the environment that sends numbers over h and the system is supposed to acknowledge their receipt and then process them in some way. In this case, we want failure to acknowledge a bit to be a system error. So, *LSpec* should be false if *Liveness* is. The specification should then be

$$LSpec \triangleq Liveness \wedge (\exists\, h : IR \wedge HSpec)$$

Since h does not occur free in *Liveness*, this definition is equivalent to

$$LSpec \triangleq \exists\, h : Liveness \wedge IR \wedge HSpec$$

which has the form (10.9) if the interface refinement *IR* of (10.9) is taken to be *Liveness* $\wedge$ *IR*. In other words, we can we make the liveness condition part of the interface refinement. (In this case, we can simplify the definition by adding liveness directly to *InnerIR*.)

In general, if *HSpec* is an open-system specification that describes liveness as well as safety, then an interface refinement may have to take the form of formula (10.11). Both *Liveness* and the liveness condition of *IR* may depend on which changes to the lower-level interface variable l are attributed to the system and which to the environmet. For the channel refinement, this means that they will depend on whether the system or the environment is sending values on the channel.

10.9 Should You Compose?

When specifying a system, should we write a monolithic specification with a single next-state action, a closed-system composition that is the conjunction of specifications of individual components, or an open-system specification? The answer is: it usually makes little difference. For a real system, the definitions of the components' actions will take hundreds or thousands of lines. The different forms of specification differ only in the few lines where we assemble the initial predicates and next-state actions into the final formula.

If you are writing a specification from scratch, it's probably better to write a monolithic specification. It is usually easier to understand. Of course, there are exceptions. We write a real-time specification as the conjunction of an untimed specification and timing constraints; describing the changes to the system variables and the timers with a single next-state action usually makes the specification harder to understand.

Writing a composite specification may be sensible when you are starting from an existing specification. If you already have a specification of one component, you may want to write a separate specification of the other component and compose the two specifications. If you have a higher-level specification, you may want to write a lower-level version as an interface refinement. However, these are rather rare situations. Moreover, it's likely to be just as easy to modify the original specification or reuse it in another way. For example, instead of conjoining a new component to the specification of an existing one, you can simply include the definition of the existing component's next-state action, with an EXTENDS or INSTANCE statement, as part of the new specification.

Composition provides a new way of writing a complete-system specification; it doesn't change the specification. The choice between a composite specification and a monolithic one is therefore ultimately a matter of taste. Disjoint-state compositions are generally straightforward and present no problems. Shared-state compositions can be tricky and require care.

Open-system specifications introduce a mathematically different kind of specification. A closed-system specification $E \wedge M$ and its open-system counterpart $E \stackrel{+}{\triangleright} M$ are not equivalent. If we really want a specification to serve as a legal contract between a user and an implementer, then we have to write an open-system specification. We also need open-system specifications if we want to specify and reason about systems built by composing off-the-shelf components with pre-existing specifications. All we can assume about such a component is that it satisfies a contract between the system builder and the supplier, and such a contract can be formalized only as an open-system specification. However, you are unlikely to encounter off-the-shelf component specifications during the early part of the twenty-first century. In the near future, open-system specifications are likely to be of theoretical interest only.