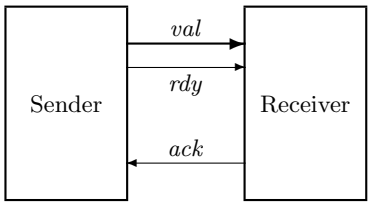
Chapter 3

An Asynchronous Interface

We now specify an interface for transmitting data between asynchronous devices. A *sender* and a *receiver* are connected as shown here.



Data is sent on *val*, and the *rdy* and *ack* lines are used for synchronization. The sender must wait for an acknowledgment (an *Ack*) for one data item before it can send the next. The interface uses the standard two-phase handshake protocol, described by the following sample behavior:

$$\begin{bmatrix} val &=& 26 \\ rdy &=& 0 \\ ack &=& 0 \end{bmatrix} \overset{Send\ 37}{\longrightarrow} \begin{bmatrix} val &=& 37 \\ rdy &=& 1 \\ ack &=& 0 \end{bmatrix} \overset{Ack}{\longrightarrow} \begin{bmatrix} val &=& 37 \\ rdy &=& 1 \\ ack &=& 1 \end{bmatrix} \overset{Send\ 4}{\longrightarrow}$$

$$\begin{bmatrix} val &=& 4 \\ rdy &=& 0 \\ ack &=& 1 \end{bmatrix} \overset{Ack}{\longrightarrow} \begin{bmatrix} val &=& 4 \\ rdy &=& 0 \\ ack &=& 0 \end{bmatrix} \overset{Send\ 19}{\longrightarrow} \begin{bmatrix} val &=& 19 \\ rdy &=& 1 \\ ack &=& 0 \end{bmatrix} \overset{Ack}{\longrightarrow} \cdots$$

(It doesn't matter what value *val* has in the initial state.)

It's easy to see from this sample behavior what the set of all possible behaviors should be—once we decide what the data values are that can be sent. But, before writing the TLA$^+$ specification that describes these behaviors, let's look at what I've just done.

In writing this behavior, I made the decision that *val* and *rdy* should change in a single step. The values of the variables *val* and *rdy* represent voltages

on some set of wires in the physical device. Voltages on different wires don't change at precisely the same instant. I decided to ignore this aspect of the physical system and pretend that the values of *val* and *rdy* represented by those voltages change instantaneously. This simplifies the specification, but at the price of ignoring what may be an important detail of the system. In an actual implementation of the protocol, the voltage on the *rdy* line shouldn't change until the voltages on the *val* lines have stabilized; but you won't learn that from my specification. Had I wanted the specification to convey this requirement, I would have written a behavior in which the value of *val* and the value of *rdy* change in separate steps.

A specification is an abstraction. It describes some aspects of the system and ignores others. We want the specification to be as simple as possible, so we want to ignore as many details as we can. But, whenever we omit some aspect of the system from the specification, we admit a potential source of error. With my specification, we can verify the correctness of a system that uses this interface, and the system could still fail because the implementer didn't know that the *val* line should stabilize before the *rdy* line is changed.

The hardest part of writing a specification is choosing the proper abstraction. I can teach you about TLA$^+$, so expressing an abstract view of a system as a TLA$^+$ specification becomes a straightforward task. But I don't know how to teach you about abstraction. A good engineer knows how to abstract the essence of a system and suppress the unimportant details when specifying and designing it. The art of abstraction is learned only through experience.

When writing a specification, you must first choose the abstraction. In a TLA$^+$ specification, this means choosing the variables that represent the system's state and the granularity of the steps that change those variables' values. Should the *rdy* and *ack* lines be represented as separate variables or as a single variable? Should *val* and *rdy* change in one step, two steps, or an arbitrary number of steps? To help make these choices, I recommend that you start by writing the first few steps of one or two sample behaviors, just as I did at the beginning of this section. Chapter 7 has more to say about these choices.

3.1 The First Specification

Let's specify the asynchronous interface with a module *AsynchInterface*. The specification uses subtraction of natural numbers, so our module EXTENDS the *Naturals* module to incorporate the definition of the subtraction operator "$-$". We next decide what the possible values of *val* should be—that is, what data values may be sent. We could write a specification that places no restriction on the data values. The specification could allow the sender first to send 37, then to send $\sqrt{-15}$, and then to send *Nat* (the entire set of natural numbers). However, any real device can send only a restricted set of values. We could pick

some specific set—for example, 32-bit numbers. However, the protocol is the same regardless of whether it's used to send 32-bit numbers or 128-bit numbers. So, we compromise between the two extremes of allowing anything to be sent and allowing only 32-bit numbers to be sent by assuming only that there is some set *Data* of data values that may be sent. The constant *Data* is a parameter of the specification. It's declared by the statement

CONSTANT *Data*

Our three variables are declared by

VARIABLES val, rdy, ack

The keywords VARIABLE and VARIABLES are synonymous, as are CONSTANT and CONSTANTS.

The variable *rdy* can assume any value—for example, $-1/2$. That is, there exist states that assign the value $-1/2$ to *rdy*. When discussing the specification, we usually say that *rdy* can assume only the values 0 and 1. What we really mean is that the value of *rdy* equals 0 or 1 in every state of any behavior satisfying the specification. But a reader of the specification shouldn't have to understand the complete specification to figure this out. We can make the specification easier to understand by telling the reader what values the variables can assume in a behavior that satisfies the specification. We could do this with comments, but I prefer to use a definition like this one:

$$\mathit{TypeInvariant} \triangleq (\mathit{val} \in \mathit{Data}) \land (\mathit{rdy} \in \{0,1\}) \land (\mathit{ack} \in \{0,1\})$$

I call the set $\{0,1\}$ the *type* of *rdy*, and I call *TypeInvariant* a *type invariant*. Let's define *type* and some other terms more precisely.

- A *state function* is an ordinary expression (one with no prime or $\Box$) that can contain variables and constants.
- A *state predicate* is a Boolean-valued state function.
- An *invariant Inv* of a specification *Spec* is a state predicate such that $\mathit{Spec} \Rightarrow \Box \mathit{Inv}$ is a theorem.
- A variable v has *type* T in a specification *Spec* iff $v \in T$ is an invariant of *Spec*.

We can make the definition of *TypeInvariant* easier to read by writing it as follows.

$$\begin{array}{lll} \mathit{TypeInvariant} & \triangleq & \land\ \mathit{val} \in \mathit{Data} \\ & & \land\ \mathit{rdy} \in \{0,1\} \\ & & \land\ \mathit{ack} \in \{0,1\} \end{array}$$

Each conjunct begins with a $\land$ and must lie completely to the right of that $\land$. (The conjunct may occupy multiple lines). We use a similar notation for disjunctions. When using this bulleted-list notation, the $\land$'s or $\lor$'s must line up precisely (even in the ASCII input). Because the indentation is significant, we can eliminate parentheses, making this notation especially useful when conjunctions and disjunctions are nested.

The formula *TypeInvariant* will not appear as part of the specification. We do not assume that *TypeInvariant* is an invariant; the specification should imply that it is. In fact, its invariance will be asserted as a theorem.

The initial predicate is straightforward. Initially, *val* can equal any element of *Data*. We can start with *rdy* and *ack* either both 0 or both 1.

$$\begin{array}{lll} Init & \triangleq & \land\ val \in Data \\ & & \land\ rdy \in \{0, 1\} \\ & & \land\ ack = rdy \end{array}$$

Now for the next-state action *Next*. A step of the protocol either sends a value or receives a value. We define separately the two actions *Send* and *Rcv* that describe the sending and receiving of a value. A *Next* step (one satisfying action *Next*) is either a *Send* step or a *Rcv* step, so it is a $Send \lor Rcv$ step. Therefore, *Next* is defined to equal $Send \lor Rcv$. Let's now define *Send* and *Rcv*.

We say that action *Send* is *enabled* in a state from which it is possible to take a *Send* step. From the sample behavior above, we see that *Send* is enabled iff *rdy* equals *ack*. Usually, the first question we ask about an action is, when is it enabled? So, the definition of an action usually begins with its enabling condition. The first conjunct in the definition of *Send* is therefore $rdy = ack$. The next conjuncts tell us what the new values of the variables *val*, *rdy*, and *ack* are. The new value val' of *val* can be any element of *Data*—that is, any value satisfying $val' \in Data$. The value of *rdy* changes from 0 to 1 or from 1 to 0, so rdy' equals $1 - rdy$ (because $1 = 1 - 0$ and $0 = 1 - 1$). The value of *ack* is left unchanged.

TLA$^+$ defines UNCHANGED v to mean that the expression v has the same value in the old and new states. More precisely, UNCHANGED v equals $v' = v$, where v' is the expression obtained from v by priming all its variables. So, we define *Send* by

$$\begin{array}{lll} Send & \triangleq & \land\ rdy = ack \\ & & \land\ val' \in Data \\ & & \land\ rdy' = 1 - rdy \\ & & \land\ \text{UNCHANGED}\ ack \end{array}$$

(I could have written $ack' = ack$ instead of UNCHANGED *ack*, but I prefer to use the UNCHANGED construct in specifications.)

A *Rcv* step is enabled iff *rdy* is different from *ack*; it complements the value of *ack* and leaves *val* and *rdy* unchanged. Both *val* and *rdy* are left unchanged iff

MODULE *AsynchInterface*

EXTENDS *Naturals*
CONSTANT *Data*
VARIABLES *val*, *rdy*, *ack*

$$\begin{aligned}
\mathit{TypeInvariant} \triangleq\ & \land\ \mathit{val} \in \mathit{Data} \\
& \land\ \mathit{rdy} \in \{0, 1\} \\
& \land\ \mathit{ack} \in \{0, 1\}
\end{aligned}$$

$$\begin{aligned}
\mathit{Init} \triangleq\ & \land\ \mathit{val} \in \mathit{Data} \\
& \land\ \mathit{rdy} \in \{0, 1\} \\
& \land\ \mathit{ack} = \mathit{rdy}
\end{aligned}$$

$$\begin{aligned}
\mathit{Send} \triangleq\ & \land\ \mathit{rdy} = \mathit{ack} \\
& \land\ \mathit{val}' \in \mathit{Data} \\
& \land\ \mathit{rdy}' = 1 - \mathit{rdy} \\
& \land\ \text{UNCHANGED}\ \mathit{ack}
\end{aligned}$$

$$\begin{aligned}
\mathit{Rcv} \triangleq\ & \land\ \mathit{rdy} \neq \mathit{ack} \\
& \land\ \mathit{ack}' = 1 - \mathit{ack} \\
& \land\ \text{UNCHANGED}\ \langle \mathit{val}, \mathit{rdy} \rangle
\end{aligned}$$

$$\mathit{Next} \triangleq \mathit{Send} \lor \mathit{Rcv}$$

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{val}, \mathit{rdy}, \mathit{ack} \rangle}$$

THEOREM $\mathit{Spec} \Rightarrow \Box\, \mathit{TypeInvariant}$

Figure 3.1: Our first specification of an asynchronous interface.

the pair of values *val*, *rdy* is left unchanged. TLA$^+$ uses angle brackets $\langle$ and $\rangle$ to enclose ordered tuples, so *Rcv* asserts that $\langle \mathit{val}, \mathit{rdy} \rangle$ is left unchanged. (Angle brackets are typed in ASCII as << and >>.) The definition of *Rcv* is therefore

$$\begin{aligned}
\mathit{Rcv} \triangleq\ & \land\ \mathit{rdy} \neq \mathit{ack} \\
& \land\ \mathit{ack}' = 1 - \mathit{ack} \\
& \land\ \text{UNCHANGED}\ \langle \mathit{val}, \mathit{rdy} \rangle
\end{aligned}$$

As in our clock example, the complete specification *Spec* should allow stuttering steps—in this case, ones that leave all three variables unchanged. So, *Spec* allows steps that leave $\langle \mathit{val}, \mathit{rdy}, \mathit{ack} \rangle$ unchanged. Its definition is

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{val}, \mathit{rdy}, \mathit{ack} \rangle}$$

Module *AsynchInterface* also asserts the invariance of *TypeInvariant*. It appears in full in Figure 3.1 on this page.

3.2 Another Specification

Module *AsynchInterface* is a fine description of the interface and its handshake protocol. However, it's not well suited for helping to specify systems that use the interface. Let's rewrite the interface specification in a form that makes it more convenient to use as part of a larger specification.

The first problem with the original specification is that it uses three variables to describe a single interface. A system might use several different instances of the interface. To avoid a proliferation of variables, we replace the three variables *val*, *rdy*, *ack* with a single variable *chan* (short for *channel*). A mathematician would do this by letting the value of *chan* be an ordered triple—for example, a state $[chan = \langle -1/2,\ 0,\ 1\rangle]$ might replace the state with $val = -1/2$, $rdy = 0$, and $ack = 1$. But programmers have learned that using tuples like this leads to mistakes; it's easy to forget if the *ack* line is represented by the second or third component. TLA$^+$ therefore provides records in addition to more conventional mathematical notation.

Let's represent the state of the channel as a record with *val*, *rdy*, and *ack* fields. If r is such a record, then $r.val$ is its *val* field. The type invariant asserts that the value of *chan* is an element of the set of all such records r in which $r.val$ is an element of the set *Data* and $r.rdy$ and $r.ack$ are elements of the set $\{0, 1\}$. This set of records is written

$$[val : Data,\ rdy : \{0, 1\},\ ack : \{0, 1\}]$$

The fields of a record are not ordered, so it doesn't matter in what order we write them. This same set of records can also be written as

$$[ack : \{0, 1\},\ val : Data,\ rdy : \{0, 1\}]$$

Initially, *chan* can equal any element of this set whose *ack* and *rdy* fields are equal, so the initial predicate is the conjunction of the type invariant and the condition $chan.ack = chan.rdy$.

A system that uses the interface may perform an operation that sends some data value d and performs some other changes that depend on the value d. We'd like to represent such an operation as an action that is the conjunction of two separate actions: one that describes the sending of d and the other that describes the other changes. Thus, instead of defining an action *Send* that sends some unspecified data value, we define the action $Send(d)$ that sends data value d. The next-state action is satisfied by a $Send(d)$ step, for some d in *Data*, or a *Rcv* step. (The value received by a *Rcv* step equals $chan.val$.) Saying that a step is a $Send(d)$ step for some d in *Data* means that there exists a d in *Data* such that the step satisfies $Send(d)$—in other words, that the step is an $\exists\, d \in Data : Send(d)$ step. So we define

$$Next \triangleq (\exists\, d \in Data : Send(d)) \lor Rcv$$

The $Send(d)$ action asserts that $chan'$ equals the record r such that

$$r.val = d \qquad r.rdy = 1 - chan.rdy \qquad r.ack = chan.ack$$

This record is written in TLA$^+$ as

$$[val \mapsto d,\ \ rdy \mapsto 1 - chan.rdy,\ \ ack \mapsto chan.ack]$$

(The symbol $\mapsto$ is typed in ASCII as `|->`.) Since the fields of records are not ordered, this record can just as well be written

$$[ack \mapsto chan.ack,\ \ val \mapsto d,\ \ rdy \mapsto 1 - chan.rdy]$$

The enabling condition of $Send(d)$ is that the rdy and ack lines are equal, so we can define

$$\begin{array}{l} Send(d) \ \triangleq \\ \quad \land\ chan.rdy = chan.ack \\ \quad \land\ chan' = [val \mapsto d,\ rdy \mapsto 1 - chan.rdy,\ ack \mapsto chan.ack] \end{array}$$

This is a perfectly good definition of $Send(d)$. However, I prefer a slightly different one. We can describe the value of $chan'$ by saying that it is the same as the value of $chan$ except that its val field equals d and its rdy field equals $1 - chan.rdy$. In TLA$^+$, we can write this value as

$$[chan \ \text{EXCEPT}\ !.val = d,\ !.rdy = 1 - chan.rdy]$$

Think of the ! as standing for the new record that the EXCEPT expression forms by modifying $chan$. So, the expression can be read as the record ! that is the same as $chan$ except $!.val$ equals d and $!.rdy$ equals $1 - chan.rdy$. In the expression that $!.rdy$ equals, the symbol @ stands for $chan.rdy$, so we can write this EXCEPT expression as

$$[chan \ \text{EXCEPT}\ !.val = d,\ !.rdy = 1 - @]$$

In general, for any record r, the expression

$$[r \ \text{EXCEPT}\ !.c_1 = e_1,\ \ldots,\ !.c_n = e_n]$$

is the record obtained from r by replacing $r.c_i$ with e_i, for each i in $1 \,..\, n$. An @ in the expression e_i stands for $r.c_i$. Using this notation, we define

$$\begin{array}{lll} Send(d) & \triangleq & \land\ chan.rdy = chan.ack \\ & & \land\ chan' = [chan \ \text{EXCEPT}\ !.val = d,\ !.rdy = 1 - @] \end{array}$$

The definition of Rcv is straightforward. A value can be received when $chan.rdy$ does not equal $chan.ack$, and receiving the value complements $chan.ack$:

$$\begin{array}{lll} Rcv & \triangleq & \land\ chan.rdy \neq chan.ack \\ & & \land\ chan' = [chan \ \text{EXCEPT}\ !.ack = 1 - @] \end{array}$$

The complete specification appears in Figure 3.2 on the next page.

MODULE *Channel*

EXTENDS *Naturals*
CONSTANT *Data*
VARIABLE *chan*
$TypeInvariant \triangleq chan \in [val : Data,\ rdy : \{0,1\},\ ack : \{0,1\}]$

$Init \triangleq \land\ TypeInvariant$
$\qquad\qquad \land\ chan.ack = chan.rdy$
$Send(d) \triangleq \land\ chan.rdy = chan.ack$
$\qquad\qquad\quad \land\ chan' = [chan \text{ EXCEPT } !.val = d,\ !.rdy = 1 - @]$
$Rcv \triangleq \land\ chan.rdy \neq chan.ack$
$\qquad\qquad \land\ chan' = [chan \text{ EXCEPT } !.ack = 1 - @]$
$Next \triangleq (\exists\, d \in Data : Send(d)) \lor Rcv$
$Spec \triangleq Init \land \Box[Next]_{chan}$

THEOREM $Spec \Rightarrow \Box\, TypeInvariant$

Figure 3.2: Our second specification of an asynchronous interface.

3.3 Types: A Reminder

As defined in Section 3.1, a variable v has type T in specification *Spec* iff $v \in T$ is an invariant of *Spec*. Thus, *hr* has type $1 \,..\, 12$ in the specification *HC* of the hour clock. This assertion does *not* mean that the variable *hr* can assume only values in the set $1 \,..\, 12$. A state is an arbitrary assignment of values to variables, so there exist states in which the value of *hr* is $\sqrt{-2}$. The assertion does mean that, in every behavior satisfying formula *HC*, the value of *hr* is an element of $1 \,..\, 12$.

If you are used to types in programming languages, it may seem strange that TLA$^+$ allows a variable to assume any value. Why not restrict our states to ones in which variables have the values of the right type? In other words, why not add a formal type system to TLA$^+$? A complete answer would take us too far afield. The question is addressed further in Section 6.2. For now, remember that TLA$^+$ is an untyped language. Type correctness is just a name for a certain invariance property. Assigning the name *TypeInvariant* to a formula gives it no special status.

3.4 Definitions

Let's examine what a definition means. If *Id* is a simple identifier like *Init* or *Spec*, then the definition $Id \triangleq exp$ defines *Id* to be synonymous with the expression *exp*. Replacing *Id* by *exp*, or vice-versa, in any expression does not change the meaning of that expression. This replacement must be done after the expression is parsed, not in the "raw input". For example, the definition $x \triangleq a + b$ makes $x * c$ equal to $(a+b)*c$, not to $a+b*c$, which equals $a+(b*c)$.

The definition of *Send* has the form $Id(p) \triangleq exp$, where *Id* and *p* are identifiers. For any expression *e*, this defines $Id(e)$ to be the expression obtained by substituting *e* for *p* in *exp*. For example, the definition of *Send* in the *Channel* module defines $Send(-5)$ to equal

$$\begin{array}{l} \land\ chan.rdy = chan.ack \\ \land\ chan' = [chan \text{ EXCEPT } !.val = -5,\ !.rdy = 1 - @] \end{array}$$

$Send(e)$ is an expression, for any expression *e*. Thus, we can write the formula $Send(-5) \land (chan.ack = 1)$. The identifier *Send* by itself is not an expression, and $Send \land (chan.ack = 1)$ is not a grammatically well-formed string. It's nonsyntactic nonsense, like $a + * b +$.

We say that *Send* is an *operator* that takes a single argument. We define operators that take more than one argument in the obvious way, the general form being

$$Id(p_1, \ldots, p_n) \triangleq exp \tag{3.1}$$

where the p_i are distinct identifiers and *exp* is an expression. We can consider defined identifiers like *Init* and *Spec* to be operators that take no argument, but we generally use *operator* to mean an operator that takes one or more arguments.

I will use the term *symbol* to mean an identifier like *Send* or an operator symbol like +. Every symbol that is used in a specification must either be a built-in operator of TLA$^+$ (like $\in$) or it must be declared or defined. Every symbol declaration or definition has a *scope* within which the symbol may be used. The scope of a VARIABLE or CONSTANT declaration, and of a definition, is the part of the module that follows it. Thus, we can use *Init* in any expression that follows its definition in module *Channel*. The statement EXTENDS *Naturals* extends the scope of symbols like + defined in the *Naturals* module to the *Channel* module.

The operator definition (3.1) implicitly includes a declaration of the identifiers $p_1, \ldots, p_n$ whose scope is the expression *exp*. An expression of the form

$$\exists\, v \in S : exp$$

has a declaration of *v* whose scope is the expression *exp*. Thus the identifier *v* has a meaning within the expression *exp* (but not within the expression *S*).

A symbol cannot be declared or defined if it already has a meaning. The expression

$$(\exists\, v \in S \,:\, exp1) \,\land\, (\exists\, v \in T \,:\, exp2)$$

is all right, because neither declaration of v lies within the scope of the other. Similarly, the two declarations of the symbol d in the *Channel* module (in the definition of *Send* and in the expression $\exists\, d$ in the definition of *Next*) have disjoint scopes. However, the expression

$$(\exists\, v \in S \,:\, (exp1 \,\land\, \exists\, v \in T \,:\, exp2))$$

is illegal because the declaration of v in the second $\exists\, v$ lies inside the scope of its declaration in the first $\exists\, v$. Although conventional mathematics and programming languages allow such redeclarations, TLA$^+$ forbids them because they can lead to confusion and errors.

3.5 Comments

Even simple specifications like the ones in modules *AsynchInterface* and *Channel* can be hard to understand from the mathematics alone. That's why I began with an intuitive explanation of the interface. That explanation made it easier for you to understand formula *Spec* in the module, which is the actual specification. Every specification should be accompanied by an informal prose explanation. The explanation may be in an accompanying document, or it may be included as comments in the specification.

Figure 3.3 on the next page shows how the hour clock's specification in module *HourClock* might be explained by comments. In the typeset version, comments are distinguished from the specification itself by the use of a different font. As shown in the figure, TLA$^+$ provides two ways of writing comments in the ASCII version. A comment may appear anywhere enclosed between `(*` and `*)`. An end-of-line comment is preceded by `\*`. Comments may be nested, so you can comment out a section of a specification by enclosing it between `(*` and `*)`, even if the section contains comments.

A comment almost always appears on a line by itself or at the end of a line. I put a comment between *HCnxt* and $\triangleq$ just to show that it can be done.

To save space, I will write few comments in the example specifications. But specifications should have lots of comments. Even if there is an accompanying document describing the system, comments are needed to help the reader understand how the specification formalizes that description.

Comments can help solve a problem posed by the logical structure of a specification. A symbol has to be declared or defined before it can be used. In module *Channel*, the definition of *Spec* has to follow the definition of *Next*, which has to follow the definitions of *Send* and *Rcv*. But it's usually easiest to

MODULE *HourClock*

This module specifies a digital clock that displays the current hour. It ignores real time, not specifying when the display can change.

EXTENDS *Naturals*

VARIABLE hr Variable hr represents the display.

$HCini \triangleq hr \in (1 \,..\, 12)$ Initially, hr can have any value from 1 through 12.

$HCnxt$ This is a weird place for a comment. $\triangleq$

The value of hr cycles from 1 through 12.

$hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$

$HC \triangleq HCini \wedge \Box[HCnxt]_{hr}$

The complete spec. It permits the clock to stop.

THEOREM $HC \Rightarrow \Box HCini$ Type-correctness of the spec.

```
---------------------- MODULE HourClock ----------------------
  (*************************************************************)
  (* This module specifies a digital clock that displays  *)
  (* the current hour.  It ignores real time, not         *)
  (* specifying when the display can change.              *)
  (*************************************************************)
EXTENDS Naturals
VARIABLE hr     \* Variable hr represents the display.
HCini  == hr \in (1 .. 12)   \* Initially, hr can have any
                             \* value from 1 through 12.
HCnxt  (* This is a weird place for a comment. *)  ==
  (***********************************************)
  (* The value of hr cycles from 1 through 12.     *)
  (***********************************************)
  hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC  ==  HCini /\ [][HCnxt]_hr
  (* The complete spec.  It permits the clock to stop. *)
--------------------------------------------------------------
THEOREM  HC => []HCini  \* Type-correctness of the spec.
==============================================================
```

Figure 3.3: The hour-clock specification with comments.

understand a top-down description of a system. We would probably first want to read the declarations of *Data* and *chan*, then the definition of *Spec*, then the definitions of *Init* and *Next*, and then the definitions of *Send* and *Rcv*. In other words, we want to read the specification more or less from bottom to top. This is easy enough to do for a module as short as *Channel*; it's inconvenient for longer specifications. We can use comments to guide the reader through a longer specification. For example, we could precede the definition of *Send* in the *Channel* module with the comment

> Actions *Send* and *Rcv* below are the disjuncts of the next-state action *Next*.

The module structure also allows us to choose the order in which a specification is read. For example, we can rewrite the hour-clock specification by splitting the *HourClock* module into three separate modules:

HCVar A module that declares the variable hr.

HCActions A module that EXTENDS modules *Naturals* and *HCVar* and defines *HCini* and *HCnxt*.

HCSpec A module that EXTENDS module *HCActions*, defines formula HC, and asserts the type-correctness theorem.

The EXTENDS relation implies a logical ordering of the modules: *HCVar* precedes *HCActions*, which precedes *HCSpec*. But the modules don't have to be read in that order. The reader can be told to read *HCVar* first, then *HCSpec*, and finally *HCActions*. The INSTANCE construct introduced below in Chapter 4 provides another tool for modularizing specifications.

Splitting a tiny specification like *HourClock* in this way would be ludicrous. But the proper splitting of modules can help make a large specification easier to read. When writing a specification, you should decide in what order it should be read. You can then design the module structure to permit reading it in that order, when each individual module is read from beginning to end. Finally, you should ensure that the comments within each module make sense when the different modules are read in the appropriate order.