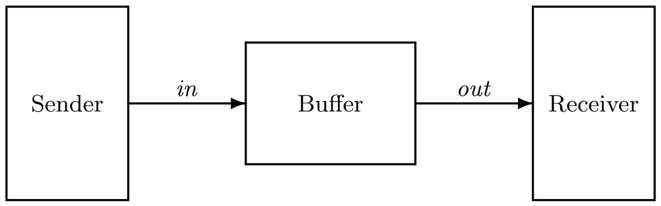
Chapter 4

A FIFO

Our next example is a FIFO buffer, called a FIFO for short—a device with which a sender process transmits a sequence of values to a receiver. The sender and receiver use two channels, *in* and *out*, to communicate with the buffer:



Values are sent over *in* and *out* using the asynchronous protocol specified by the *Channel* module of Figure 3.2 on page 30. The system's specification will allow behaviors with four kinds of nonstuttering steps: *Send* and *Rcv* steps on both the *in* channel and the *out* channel.

4.1 The Inner Specification

The specification of the FIFO first EXTENDS modules *Naturals* and *Sequences*. The *Sequences* module defines operations on finite sequences. We represent a finite sequence as a tuple, so the sequence of three numbers 3, 2, 1 is the triple $\langle 3, 2, 1 \rangle$. The *Sequences* module defines the following operators on sequences:

$Seq(S)$	The set of all sequences of elements of the set S. For example, $\langle 3, 7 \rangle$ is an element of $Seq(Nat)$.
$Head(s)$	The first element of sequence s. For example, $Head(\langle 3, 7 \rangle)$ equals 3.

$Tail(s)$	The tail of sequence s, which consists of s with its head removed. For example, $Tail(\langle 3, 7 \rangle)$ equals $\langle 7 \rangle$.
$Append(s, e)$	The sequence obtained by appending element e to the tail of sequence s. For example, $Append(\langle 3, 7 \rangle, 3)$ equals $\langle 3, 7, 3 \rangle$.
$s \circ t$	The sequence obtained by concatenating the sequences s and t. For example, $\langle 3, 7 \rangle \circ \langle 3 \rangle$ equals $\langle 3, 7, 3 \rangle$. (We type $\circ$ in ASCII as `\o`.)
$Len(s)$	The length of sequence s. For example, $Len(\langle 3, 7 \rangle)$ equals 2.

The FIFO's specification continues by declaring the constant *Message*, which represents the set of all messages that can be sent.[1] It then declares the variables. There are three variables: *in* and *out*, representing the channels, and a third variable q that represents the queue of buffered messages. The value of q is the sequence of messages that have been sent by the sender but not yet received by the receiver. (Section 4.3 has more to say about this additional variable q.)

We want to use the definitions in the *Channel* module to specify operations on the channels *in* and *out*. This requires two instances of that module—one in which the variable *chan* of the *Channel* module is replaced with the variable *in* of our current module, and the other in which *chan* is replaced with *out*. In both instances, the constant *Data* of the *Channel* module is replaced with *Message*. We obtain the first of these instances with the statement

$$InChan \triangleq \textsc{instance}\ Channel\ \textsc{with}\ Data \leftarrow Message,\ chan \leftarrow in$$

For every symbol σ defined in module *Channel*, this defines $InChan!\sigma$ to have the same meaning in the current module as σ had in module *Channel*, except with *Message* substituted for *Data* and *in* substituted for *chan*. For example, this statement defines $InChan!TypeInvariant$ to equal

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

(The statement does *not* define $InChan!Data$ because *Data* is declared, not defined, in module *Channel*.) We introduce our second instance of the *Channel* module with the analogous statement

$$OutChan \triangleq \textsc{instance}\ Channel\ \textsc{with}\ Data \leftarrow Message,\ chan \leftarrow out$$

The initial states of the *in* and *out* channels are specified by $InChan!Init$ and $OutChan!Init$. Initially, no messages have been sent or received, so q should

[1] I like to use a singular noun like *Message* rather than a plural like *Messages* for the name of a set. That way, the $\in$ in the expression $m \in Message$ can be read *is a*. This is the same convention that most programmers use for naming types.

equal the empty sequence. The empty sequence is the 0-tuple (there's only one, and it's written $\langle\,\rangle$), so we define the initial predicate to be

$$\begin{aligned} \mathit{Init} \;\triangleq\; & \land\ \mathit{InChan}!\mathit{Init} \\ & \land\ \mathit{OutChan}!\mathit{Init} \\ & \land\ q = \langle\,\rangle \end{aligned}$$

We next define the type invariant. The type invariants for *in* and *out* come from the *Channel* module, and the type of q is the set of finite sequences of messages. The type invariant for the FIFO specification is therefore

$$\begin{aligned} \mathit{TypeInvariant} \;\triangleq\; & \land\ \mathit{InChan}!\mathit{TypeInvariant} \\ & \land\ \mathit{OutChan}!\mathit{TypeInvariant} \\ & \land\ q \in \mathit{Seq}(\mathit{Message}) \end{aligned}$$

The four kinds of nonstuttering steps allowed by the next-state action are described by four actions:

$\mathit{SSend}(\mathit{msg})$	The sender sends message *msg* on the *in* channel.
BufRcv	The buffer receives the message from the *in* channel and appends it to the tail of q.
BufSend	The buffer removes the message from the head of q and sends it on channel *out*.
RRcv	The receiver receives the message from the *out* channel.

The definitions of these actions, along with the rest of the specification, are in module *InnerFIFO* of Figure 4.1 on the next page. The reason for the adjective *Inner* is explained in Section 4.3 below.

4.2 Instantiation Examined

The INSTANCE statement is seldom used except in one idiom for hiding variables, which is described in Section 4.3. So, most readers can skip this section and go directly to page 41.

4.2.1 Instantiation Is Substitution

Consider the definition of *Next* in module *Channel* (page 30). We can remove every defined symbol that appears in that definition by using the symbol's definition. For example, we can eliminate the expression $\mathit{Send}(d)$ by expanding the definition of *Send*. We can repeat this process. For example, the "$-$" that appears in the expression $1 - @$ (obtained by expanding the definition of *Send*)

```
─────────────────────── MODULE InnerFIFO ───────────────────────
EXTENDS Naturals, Sequences
CONSTANT Message
VARIABLES in, out, q
InChan  ≜ INSTANCE Channel WITH Data ← Message, chan ← in
OutChan ≜ INSTANCE Channel WITH Data ← Message, chan ← out
────────────────────────────────────────────────────────────────
Init ≜ ∧ InChan!Init
       ∧ OutChan!Init
       ∧ q = ⟨ ⟩

TypeInvariant ≜ ∧ InChan!TypeInvariant
                ∧ OutChan!TypeInvariant
                ∧ q ∈ Seq(Message)

SSend(msg) ≜ ∧ InChan!Send(msg)        Send msg on channel in.
             ∧ UNCHANGED ⟨out, q⟩

BufRcv ≜ ∧ InChan!Rcv                  Receive message from channel in
         ∧ q′ = Append(q, in.val)        and append it to tail of q.
         ∧ UNCHANGED out

BufSend ≜ ∧ q ≠ ⟨ ⟩                    Enabled only if q is nonempty.
          ∧ OutChan!Send(Head(q))      Send Head(q) on channel out
          ∧ q′ = Tail(q)                 and remove it from q.
          ∧ UNCHANGED in

RRcv ≜ ∧ OutChan!Rcv                   Receive message from channel out.
       ∧ UNCHANGED ⟨in, q⟩

Next ≜ ∨ ∃ msg ∈ Message : SSend(msg)
       ∨ BufRcv
       ∨ BufSend
       ∨ RRcv

Spec ≜ Init ∧ □[Next]_⟨in, out, q⟩
────────────────────────────────────────────────────────────────
THEOREM Spec ⇒ □TypeInvariant
────────────────────────────────────────────────────────────────
```

Figure 4.1: The specification of a FIFO, with the internal variable q visible.

can be eliminated by using the definition of "$-$" from the *Naturals* module. Continuing in this way, we eventually obtain a definition for *Next* in terms of only the built-in operators of TLA$^+$ and the parameters *Data* and *chan* of the *Channel* module. We consider this to be the "real" definition of *Next* in module *Channel*. The statement

$$\mathit{InChan} \;\triangleq\; \textsc{instance}\ \mathit{Channel}\ \textsc{with}\ \mathit{Data} \leftarrow \mathit{Message},\ \mathit{chan} \leftarrow \mathit{in}$$

in module *InnerFIFO* defines *InChan*!*Next* to be the formula obtained from this real definition of *Next* by substituting *Message* for *Data* and *in* for *chan*. This defines *InChan*!*Next* in terms of only the built-in operators of TLA$^+$ and the parameters *Message* and *in* of module *InnerFIFO*.

Let's now consider an arbitrary INSTANCE statement

$$\mathit{IM} \;\triangleq\; \textsc{instance}\ M\ \textsc{with}\ p_1 \leftarrow e_1,\ \ldots,\ p_n \leftarrow e_n$$

Let Σ be a symbol defined in module M and let d be its "real" definition. The INSTANCE statement defines $\mathit{IM}!\Sigma$ to have as its real definition the expression obtained from d by replacing all instances of p_i by the expression e_i, for each i. The definition of $\mathit{IM}!\Sigma$ must contain only the parameters (declared constants and variables) of the current module, not the ones of module M. Hence, the p_i must consist of all the parameters of module M. The e_i must be expressions that are meaningful in the current module.

4.2.2 Parametrized Instantiation

The FIFO specification uses two instances of module *Channel*—one with *in* substituted for *chan* and the other with *out* substituted for *chan*. We could instead use a single parametrized instance by putting the following statement in module *InnerFIFO*:

$$\mathit{Chan}(\mathit{ch}) \;\triangleq\; \textsc{instance}\ \mathit{Channel}\ \textsc{with}\ \mathit{Data} \leftarrow \mathit{Message}, \mathit{chan} \leftarrow \mathit{ch}$$

For any symbol Σ defined in module *Channel* and any expression *exp*, this defines $\mathit{Chan}(\mathit{exp})!\Sigma$ to equal formula Σ with *Message* substituted for *Data* and *exp* substituted for *chan*. The *Rcv* action on channel *in* could then be written $\mathit{Chan}(\mathit{in})!\mathit{Rcv}$, and the $\mathit{Send}(\mathit{msg})$ action on channel *out* could be written $\mathit{Chan}(\mathit{out})!\mathit{Send}(\mathit{msg})$.

The instantiation above defines $\mathit{Chan}!\mathit{Send}$ to be an operator with two arguments. Writing $\mathit{Chan}(\mathit{out})!\mathit{Send}(\mathit{msg})$ instead of $\mathit{Chan}!\mathit{Send}(\mathit{out}, \mathit{msg})$ is just an idiosyncrasy of the syntax. It is no stranger than the syntax for infix operators, which has us write $a + b$ instead of $+(a, b)$.

Parametrized instantiation is used almost exclusively in the TLA$^+$ idiom for variable hiding, described in Section 4.3. You can use that idiom without understanding it, so you probably don't need to know anything about parametrized instantiation.

4.2.3 Implicit Substitutions

The use of *Message* as the name for the set of transmitted values in the FIFO specification is a bit strange, since we had just used the name *Data* for the analogous set in the asynchronous channel specifications. Suppose we had used *Data* in place of *Message* as the constant parameter of module *InnerFIFO*. The first instantiation statement would then have been

$$InChan \triangleq \textsc{instance}\ Channel\ \textsc{with}\ Data \leftarrow Data, chan \leftarrow in$$

The substitution $Data \leftarrow Data$ indicates that the constant parameter *Data* of the instantiated module *Channel* is replaced with the expression *Data* of the current module. TLA$^+$ allows us to drop any substitution of the form $\Sigma \leftarrow \Sigma$, for a symbol Σ. So, the statement above can be written as

$$InChan \triangleq \textsc{instance}\ Channel\ \textsc{with}\ chan \leftarrow in$$

We know there is an implied $Data \leftarrow Data$ substitution because an INSTANCE statement must have a substitution for every parameter of the instantiated module. If some parameter p has no explicit substitution, then there is an implicit substitution $p \leftarrow p$. This means that the INSTANCE statement must lie within the scope of a declaration or definition of the symbol p.

It is quite common to instantiate a module with this kind of implicit substitution. Often, every parameter has an implicit substitution, in which case the list of explicit substitutions is empty. The WITH is then omitted.

4.2.4 Instantiation Without Renaming

So far, all the instantiations we've used have been with renaming. For example, the first instantiation of module *Channel* renames the defined symbol *Send* as *InChan*!*Send*. This kind of renaming is necessary if we are using multiple instances of the module, or a single parametrized instance. The two instances *InChan*!*Init* and *OutChan*!*Init* of *Init* in module *InnerFIFO* are different formulas, so they need different names.

Sometimes we need only a single instance of a module. For example, suppose we are specifying a system with only a single asynchronous channel. We then need only one instance of *Channel*, so we don't have to rename the instantiated symbols. In that case, we can write something like

$$\textsc{instance}\ Channel\ \textsc{with}\ Data \leftarrow D, chan \leftarrow x$$

This instantiates *Channel* with no renaming, but with substitution. Thus, it defines *Rcv* to be the formula of the same name from the *Channel* module, except with D substituted for *Data* and x substituted for *chan*. The expressions substituted for an instantiated module's parameters must be defined. So, this INSTANCE statement must be within the scope of the definitions or declarations of D and x.

4.3 Hiding the Queue

Module *InnerFIFO* of Figure 4.1 defines *Spec* to be $Init \land \Box[Next]_{\ldots}$, the sort of formula we've become accustomed to as a system specification. However, formula *Spec* describes the value of variable q, as well as of the variables *in* and *out*. The picture of the FIFO system I drew on page 35 shows only channels *in* and *out*; it doesn't show anything inside the boxes. A specification of the FIFO should describe only the values sent and received on the channels. The variable q, which represents what's going on inside the box labeled *Buffer*, is used to specify what values are sent and received. It is an *internal* variable and, in the final specification, it should be hidden.

In TLA, we hide a variable with the existential quantifier $\boldsymbol{\exists}$ of temporal logic. The formula $\boldsymbol{\exists}\, x : F$ is true of a behavior iff there exists some sequence of values—one in each state of the behavior—that can be assigned to the variable x that will make formula F true. (The meaning of $\boldsymbol{\exists}$ is defined more precisely in Section 8.8.)

The obvious way to write a FIFO specification in which q is hidden is with the formula $\boldsymbol{\exists}\, q : Spec$. However, we can't put this definition in module *InnerFIFO* because q is already declared there, and a formula $\boldsymbol{\exists}\, q : \ldots$ would redeclare it. Instead, we use a new module with a parametrized instantiation of the *InnerFIFO* module (see Section 4.2.2 on page 39):

```
──────────────── MODULE FIFO ────────────────
CONSTANT Message
VARIABLES in, out

Inner(q) ≜ INSTANCE InnerFIFO

Spec ≜ ∃ q : Inner(q)!Spec
─────────────────────────────────────────────
```

Observe that the INSTANCE statement is an abbreviation for

$$\begin{array}{l} Inner(q) \;\triangleq\; \text{INSTANCE } InnerFIFO \\ \qquad\qquad\qquad \text{WITH } q \leftarrow q,\; in \leftarrow in,\; out \leftarrow out,\; Message \leftarrow Message \end{array}$$

The variable parameter q of module *InnerFIFO* is instantiated with the parameter q of the definition of *Inner*. The other parameters of the *InnerFIFO* module are instantiated with the parameters of module *FIFO*.

If this seems confusing, don't worry about it. Just learn the TLA$^+$ idiom for hiding variables used here and be content with its intuitive meaning. In fact, for most applications, there's no need to hide variables in the specification. You can just write the inner specification and note in the comments which variables should be regarded as visible and which as internal (hidden).

4.4 A Bounded FIFO

We have specified an unbounded FIFO—a buffer that can hold an unbounded number of messages. Any real system has a finite amount of resources, so it can contain only a bounded number of in-transit messages. In many situations, we wish to abstract away the bound on resources and describe a system in terms of unbounded FIFOs. In other situations, we may care about that bound. We then want to strengthen our specification by placing a bound N on the number of outstanding messages.

A specification of a bounded FIFO differs from our specification of the unbounded FIFO only in that action *BufRcv* should not be enabled unless there are fewer than N messages in the buffer—that is, unless $Len(q)$ is less than N. It would be easy to write a complete new specification of a bounded FIFO by copying module *InnerFIFO* and just adding the conjunct $Len(q) < N$ to the definition of *BufRcv*. But let's use module *InnerFIFO* as it is, rather than copying it.

The next-state action *BNext* for the bounded FIFO is the same as the FIFO's next-state action *Next* except that it allows a *BufRcv* step only if $Len(q)$ is less than N. In other words, *BNext* should allow a step only if (i) it's a *Next* step and (ii) if it's a *BufRcv* step, then $Len(q) < N$ is true in the first state. In other words, *BNext* should equal

$$Next \;\land\; (BufRcv \Rightarrow (Len(q) < N))$$

Module *BoundedFIFO* in Figure 4.2 on the next page contains the specification. It introduces the new constant parameter N. It also contains the statement

$$\textsc{assume}\ (N \in Nat) \land (N > 0)$$

which asserts that, in this module, we are assuming that N is a positive natural number. Such an assumption has no effect on any definitions made in the module. However, it may be taken as a hypothesis when proving any theorems asserted in the module. In other words, a module asserts that its assumptions imply its theorems. It's a good idea to state this kind of simple assumption about constants.

An ASSUME statement should be used only to make assumptions about constants. The formula being assumed should not contain any variables. It might be tempting to assert type declarations as assumptions—for example, to add to module *InnerFIFO* the assumption $q \in Seq(Message)$. However, that would be wrong because it asserts that, in any state, q is a sequence of messages. As we observed in Section 3.3, a state is a completely arbitrary assignment of values to variables, so there are states in which q has the value $\sqrt{-17}$. Assuming that such a state doesn't exist would lead to a logical contradiction.

You may wonder why module *BoundedFIFO* assumes that N is a positive natural, but doesn't assume that *Message* is a set. Similarly, why didn't we

```
──────────────── MODULE BoundedFIFO ────────────────
EXTENDS Naturals, Sequences
VARIABLES in, out
CONSTANT Message, N
ASSUME (N ∈ Nat) ∧ (N > 0)
Inner(q) ≜ INSTANCE InnerFIFO
BNext(q) ≜ ∧ Inner(q)!Next
           ∧ Inner(q)!BufRcv ⇒ (Len(q) < N)
Spec ≜ ∃ q : Inner(q)!Init ∧ □[BNext(q)]_⟨in, out, q⟩
────────────────────────────────────────────────────
```

Figure 4.2: A specification of a FIFO buffer of length *N*.

assume that the constant parameter *Data* in our asynchronous interface specifications is a set? The answer is that, in TLA$^+$, every value is a set.[2] A value like the number 3, which we don't think of as a set, is formally a set. We just don't know what its elements are. The formula $2 \in 3$ is a perfectly reasonable one, but TLA$^+$ does not specify whether it's true or false. So, we don't have to assume that *Message* is a set because we know that it is one.

Although *Message* is automatically a set, it isn't necessarily a finite set. For example, *Message* could be instantiated with the set *Nat* of natural numbers. If you want to assume that a constant parameter is a finite set, then you need to state this as an assumption. (You can do this with the *IsFiniteSet* operator from the *FiniteSets* module, described in Section 6.1.) However, most specifications make perfect sense for infinite sets of messages or processors, so there is no reason to assume these sets to be finite.

4.5 What We're Specifying

I wrote at the beginning of this chapter that we were going to specify a FIFO buffer. Formula *Spec* of the *FIFO* module actually specifies a set of behaviors, each representing a sequence of sending and receiving operations on the channels *in* and *out*. The sending operations on *in* are performed by the sender, and the receiving operations on *out* are performed by the receiver. The sender and receiver are not part of the FIFO buffer; they form its *environment*.

Our specification describes a system consisting of the FIFO buffer and its environment. The behaviors satisfying formula *Spec* of module *FIFO* represent those histories of the universe in which both the system and its environment

[2] TLA$^+$ is based on the mathematical formalism known as Zermelo-Fränkel set theory, also called ZF.

behave correctly. It's often helpful in understanding a specification to indicate explicitly which steps are system steps and which are environment steps. We can do this by defining the next-state action to be

$$Next \triangleq SysNext \lor EnvNext$$

where *SysNext* describes system steps and *EnvNext* describes environment steps. For the FIFO, we have

$$\begin{aligned} SysNext &\triangleq BufRcv \lor BufSend \\ EnvNext &\triangleq (\exists\, msg \in Message : SSend(msg)) \lor RRcv \end{aligned}$$

While suggestive, this way of defining the next-state action has no formal significance. The specification *Spec* equals $Init \land \Box[Next]_{\ldots}$; changing the way we structure the definition of *Next* doesn't change its meaning. If a behavior fails to satisfy *Spec*, nothing tells us if the system or its environment is to blame.

A formula like *Spec*, which describes the correct behavior of both the system and its environment, is called a *closed-system* or *complete-system* specification. An *open-system* specification is one that describes only the correct behavior of the system. A behavior satisfies an open-system specification if it represents a history in which either the system operates correctly, or it failed to operate correctly only because its environment did something wrong. Section 10.7 explains how to write open-system specifications.

Open-system specifications are philosophically more satisfying. However, closed-system specifications are a little easier to write, and the mathematics underlying them is simpler. So, we almost always write closed-system specifications. It's usually quite easy to turn a closed-system specification into an open-system specification. But in practice, there's seldom any reason to do so.