Specifying Concurrent Systems with TLA$^+$

Leslie Lamport

Compaq

3 March 1999

To appear in *Calculational System Design.* M. Broy and R. Steinbrüggen, editors. IOS Press, Amsterdam, 1999.

Contents

Specifying Concurrent Systems with TLA$^+$

Leslie LAMPORT
Compaq

1 Introduction

Writing a specification for a system helps us understand it. It's a good idea to understand something before building it, so it's a good idea to specify a system before implementing it. Specifications written in an imprecise language like English are usually imprecise. In engineering, imprecision is an invitation to error. Science and engineering have adopted mathematics as a language for writing precise descriptions.

The mathematics written by most mathematicians and scientists is still imprecise. Most mathematics texts are precise in the small, but imprecise in the large. Each equation is a precise assertion, but you have to read the text to understand how the equations relate to one another and what the theorems really mean. Logicians have developed ways of eliminating the words and formalizing mathematics. Most mathematicians and computer scientists think that writing mathematics formally, without words, is tiresome. Once you learn how, I hope you'll find that it's easy to express ordinary mathematics in a precise, completely formal language.

To specify systems with mathematics, we must decide what kind of mathematics to use. We can specify an ordinary sequential program by describing its output as a function of its input. So, sequential programs can be specified in terms of functions. Concurrent systems are usually described in terms of their behaviors—what they do in the course of an execution. In 1977, Amir Pnueli introduced the use of temporal logic for describing such behaviors [12].

Temporal logic is appealing because, in principle, it allows a concurrent system to be described by a single formula. In practice, temporal logic proved to be cumbersome. Pnueli's temporal logic was ideal for describing some properties of systems, but awkward for others. So, it was usually combined with some more traditional way of describing systems.

In the late 1980's, I discovered TLA, the Temporal Logic of Actions. TLA is a simple variant of Pnueli's original logic that makes it practical to write a specification as a single formula. Most of a TLA specification consists of ordinary, nontemporal mathematics. Temporal logic plays a significant role only in describing those properties that it's good at describing. TLA also provides a nice way to formalize the style of reasoning about systems that has proved to be most effective in practice—a style known as *assertional* reasoning. However, the topic of this chapter is specification, not proof, so I will have little to say about proofs.

TLA provides a mathematical foundation for describing concurrent systems. To write specifications, we need a complete language built atop that foundation. Although mathematicians have developed the science of writing formulas, they haven't turned that science into an engineering discipline. They have developed notations for mathematics in the small, but not for mathematics in the large. The specification of a real system can be dozens or even hundreds of pages long. Mathematicians know how to write 20-line formulas, not 20-page formulas. So, I had to introduce notations for writing long formulas. I also took from programming languages some ideas for modularizing large specifications.

The language I came up with is called TLA$^+$. I refined TLA$^+$ in the course of writing specifications of disparate systems. But, it has changed little in the last few years. I have found TLA$^+$ to be quite good for specifying a wide class of systems—from program interfaces (api's) to distributed systems. It can be used to write a precise, formal describe of almost any sort of discrete system. It's especially well suited to describing asynchronous systems—that is, systems with components that do not operate in strict lock-step.

One advantage of having a precise language for writing specifications is that one can build tools to help with the process. Among the tools currently under development are a parser and a model checker. The parser can catch simple errors in any TLA$^+$ specification. The model checker can catch many more errors, but it will work on a restricted class of specifications—a class that should include a large fraction of the specifications of interest to industry today. These tools will be described elsewhere; see the TLA Web page for the latest information [7].

A system specification consists of a lot of ordinary mathematics glued together with a little bit of temporal logic. So, most of the work in writing a precise specification consists of expressing ordinary mathematics precisely. That's why most of the details of TLA$^+$ are concerned with ordinary mathematics.

Unfortunately, the computer science departments in many universities apparently believe that fluency in C++ is more important than a sound education in elementary mathematics. So, some readers may be unfamiliar with the mathematics needed to write specifications. Fortunately, this mathematics is quite simple. If overexposure to C++ hasn't destroyed your ability to think logically, you should have no trouble filling any gaps in your mathematics education. You probably learned arithmetic before being exposed to C++, so I will assume you know about numbers and arithmetic operations on them.[2] I will try to explain all other mathematical concepts that you need, starting in Section 2 with a review of some elementary math. I hope most readers will find this review completely unnecessary. However, you should at least glance at it, since it introduces some notation that is used later.

After the brief review of simple mathematics in the next section, Sections 3 through 6 describe TLA$^+$ with a sequence of examples. Section 7 explains some more about the math used in writing specifications. By the time you finish Section 7, you should be able to specify the class of properties known as *safety* properties. Specifying safety properties requires almost no temporal logic. Temporal logic comes to the fore in Section 8, where it is used to specify the additional class of properties known as liveness properties. In practice, specifications are written to help detect errors—hopefully, before the system is built. Experience indicates that specifying safety properties catches many more errors than specifying liveness properties. So, you may decide not to bother reading Section 8. Finally, Section 9 reviews what we've done and provides some practical advice. At the end of the chapter is some useful reference material: Figures 12 and 13 list all the built-in operators of TLA$^+$, Figure 14 lists the user-definable operator symbols, and Figure 15 shows the ASCII representations of all TLA$^+$ symbols.

2 A Little Simple Math

2.1 Propositional Logic

Elementary algebra is the mathematics of real numbers and the operators $+$, $-$, $*$ (multiplication), and $/$ (division). Propositional logic is the mathematics of the two Boolean values TRUE and FALSE and the five operators whose names (and common pronunciations) are:

$\land$ conjunction (and)	$\neg$ negation (not)
$\lor$ disjunction (or)	$\Rightarrow$ implication (implies)
$\equiv$ equivalence (is equivalent to)	

To learn how to compute with numbers, you had to memorize addition and multiplication tables and algorithms for calculating with multidigit numbers. Propositional logic is much

[2] Some readers may need reminding that numbers are not strings of bits, and $2^{33} * 2^{33}$ equals 2^{66}, not *overflow error*.

simpler, since there are only two values, TRUE and FALSE. To learn how to compute with these values, all you need to know are the following definitions of the five Boolean operators:

- $\wedge$ $F \wedge G$ equals TRUE iff[3] both F and G equal TRUE.
- $\vee$ $F \vee G$ equals TRUE iff F or G equals TRUE (or both do).[4]
- $\neg$ $\neg F$ equals TRUE iff F equals FALSE.
- $\Rightarrow$ $F \Rightarrow G$ equals TRUE iff F equals FALSE or G equals TRUE (or both).
- $\equiv$ $F \equiv G$ equals TRUE iff F and G both equal TRUE or both equal FALSE.

We can also describe these operators by *truth tables*. This truth table for $F \Rightarrow G$ gives its value for all four combinations of truth values of F and G:

F	G	$F \Rightarrow G$
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	TRUE
FALSE	FALSE	TRUE

People are often confused about why $\Rightarrow$ means implication. In particular, they don't understand why FALSE $\Rightarrow$ TRUE and FALSE $\Rightarrow$ FALSE should equal TRUE. The explanation is simple. We expect that if n is greater than 3 then it should be greater than 1, so $n > 3$ should imply $n > 1$. Substituting 4, 2, and 0 for n in the formula $(n > 3) \Rightarrow (n > 0)$ explains why we can read $F \Rightarrow G$ as F *implies* G or, equivalently, as *if* F *then* G.

The equivalence operator $\equiv$ is equality for Booleans. We can replace $\equiv$ by $=$, but not vice versa. (We can write FALSE $= \neg$TRUE, but not $2 + 2 \equiv 4$.) Writing $\equiv$ instead of $=$ makes it clear that the equal expressions are Booleans.[5]

Formulas of propositional logic are made up of values, operators, variables, and parentheses just like those of algebra. In algebraic formulas, $*$ has higher precedence (binds more tightly) than $+$, so $x + y * z$ means $x + (y * z)$. Similarly, $\neg$ has higher precedence than $\wedge$ and $\vee$, which have higher precedence than $\Rightarrow$ and $\equiv$, so $\neg F \wedge G \Rightarrow H$ means $((\neg F) \wedge G) \Rightarrow H$. Other mathematical operators like $+$ and $>$ have higher precedence than the operators of propositional logic, so $n > 0 \Rightarrow n - 1 \geq 0$ means $(n > 0) \Rightarrow (n - 1 \geq 0)$. Redundant parentheses can't hurt and often make a formula easier to read. If you have any doubt about whether parentheses are needed, use them.

The operators $\wedge$ and $\vee$ are associative, just like $+$ and $*$. Associativity of $+$ means that $x + (y + z)$ equals $(x + y) + z$, so we can write $x + y + z$ without parentheses. Similarly, associativity of $\wedge$ and $\vee$ lets us write $F \wedge G \wedge H$ or $F \vee G \vee H$.

A *tautology* of propositional logic is a formula like $(F \Rightarrow G) \equiv (\neg F \vee G)$ that is true for all possible truth values of its variables. One can prove all tautologies from a few simple axioms and rules. However, that would be like computing 437+256 from the axioms of arithmetic. It's much easier to verify that a simple formula is a tautology by writing its truth table—that is, by directly calculating the value of the formula for all possible truth values of its components. The formula is a tautology iff it equals TRUE for all these values. To construct the truth table

[3] *iff* stands for *if and only if*.

[4] Like most mathematicians, I use *or* to mean *and/or*.

[5] Section 7.2 touches on a more subtle reason to write $\equiv$ instead of $=$.

for a formula, we construct the truth table for all its subformulas. For example, the following truth table shows that $(F \Rightarrow G) \equiv (\neg F \vee G)$ is indeed a tautology.

F	G	$F \Rightarrow G$	$\neg F$	$\neg F \vee G$	$(F \Rightarrow G) \equiv \neg F \vee G$
TRUE	TRUE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	TRUE	TRUE	TRUE	TRUE
FALSE	FALSE	TRUE	TRUE	TRUE	TRUE

Propositional logic is the basis of all mathematical reasoning. It should be as familiar to you as simple algebra. Checking that $\neg(F \vee G) \equiv \neg F \wedge \neg G$ is a tautology should be as easy as checking that $2 * (x + 3 * y)$ equals $2 * x + 6 * y$.

Although propositional logic is simple, complex propositional formulas can get confusing. You may find yourself trying to simplify some formula and not being sure if the simplified version means the same thing as the original. There exist a number of programs for verifying propositional logic tautologies; some of them can be found and used on the World Wide Web.

2.2 Sets

Set theory is the foundation of ordinary mathematics. A set is often described as a collection of elements, but saying that a set is a collection doesn't explain very much. The concept of set is so fundamental that we don't try to define it. We take as undefined concepts the notion of a set and the relation $\in$, where $x \in S$ means that x is an element of S. We often say *is in* instead of *is an element of*.

A set can have a finite or infinite number of elements. The set of all natural numbers (0, 1, 2, etc.) is an infinite set. The set of all natural numbers less than 3 is finite, and contains the three elements 0, 1, and 2. We can write this set $\{0, 1, 2\}$.

A set is completely determined by its elements. Two sets are equal iff they have the same elements. Thus, $\{0, 1, 2\}$ and $\{2, 1, 0\}$ and $\{0, 0, 1, 2, 2\}$ are all the same set—the unique set containing the three elements 0, 1, and 2. The empty set, which we write $\{\}$, is the unique set that has no elements.

The most common operations on sets are:

$\cap$ intersection $\quad$ $\cup$ union $\quad$ $\subseteq$ subset $\quad$ $\setminus$ set difference

Here are their definitions and examples of their use:

$S \cap T$ The set of elements in both S and T. $\{1, -1/2, 3\} \cap \{1, 2, 3, 5, 7\} = \{1, 3\}$

$S \cup T$ The set of elements in S or T (or both). $\{1, -1/2\} \cup \{1, 5, 7\} = \{1, -1/2, 5, 7\}$

$S \subseteq T$ True iff every element of S is an element of T. $\{1, 3\} \subseteq \{3, 2, 1\}$

$S \setminus T$ The set of elements in S that are not in T. $\{1, -1/2, 3\} \setminus \{1, 5, 7\} = \{-1/2, 3\}$

This is all you need to know about sets before we start looking at how to specify systems. We'll return to set theory in Section 7.1.

2.3 Predicate Logic

Once we have sets, it's natural to say that some formula is true for all the elements of a set, or for some of the elements of a set. Predicate logic extends propositional logic with the two quantifiers:

$\forall$ universal quantification (for all)
$\exists$ existential quantification (there exists)

The formula $\forall x \in S : F$ asserts that formula F is true for every element x in the set S. For example, $\forall n \in Nat : n + 1 > n$ asserts that the formula $n + 1 > n$ is true for all elements n of the set Nat of natural numbers. This formula happens to be true.

The formula $\exists x \in S : F$ asserts that formula F is true for at least one element x in S. For example, $\exists n \in Nat : n^2 = 2$ asserts that there exists a natural number n whose square equals 2. This formula happens to be false.

Formula F is true for all $x \in S$ iff there is no $x \in S$ for which F is false, which is true iff there is no $x \in S$ for which $\neg F$ is true. Hence, the formula

$$(1) \quad (\exists x \in S : F) \;\equiv\; \neg(\forall x \in S : \neg F)$$

is a tautology of predicate logic.[6]

Since there exists no element in the empty set, the formula $\exists x \in \{\} : F$ is false for every formula F. By (1), this implies that $\forall x \in \{\} : F$ must be true for every F.

The quantification in the formulas $\forall x \in S : F$ and $\exists x \in S : F$ is said to be *bounded*, since these formulas make an assertion only about elements in the set S. There is also unbounded quantification. The formula $\forall x : F$ asserts that F is true for all values x, and $\exists x : F$ asserts that F is true for at least one value of x—a value that is not constrained to be in any particular set. Bounded and unbounded quantification are related by the following tautologies:

$$\begin{aligned} (\forall x \in S : F) &\equiv (\forall x : (x \in S) \Rightarrow F) \\ (\exists x \in S : F) &\equiv (\exists x : (x \in S) \land F) \end{aligned}$$

The analog of (1) for unbounded quantifiers is also a tautology:

$$(\exists x : F) \;\equiv\; \neg(\forall x : \neg F)$$

Whenever possible, it is better to use bounded than unbounded quantification in a specification. This makes the specification easier for both people and tools to understand.

Logicians have rules for proving such predicate-logic tautologies, but you shouldn't need them. You should become familiar enough with predicate logic that simple tautologies are obvious. It can help to think of $\forall x \in S : F$ as the conjunction of the formulas obtained by substituting all possible elements of S for x in F. The associativity and commutativity of conjunction then lead to the tautology:

$$(\forall x \in S : F) \land (\forall x \in S : G) \equiv (\forall x \in S : F \land G)$$

Similarly, you can think of $\exists x \in S : F$ as the disjunction of formulas, so associativity and commutativity of disjunction imply:

$$(\exists x \in S : F) \lor (\exists x \in S : G) \equiv (\exists x \in S : F \lor G)$$

for any set S and formulas F and G.

Mathematicians use some obvious abbreviations for nested quantifiers. For example:

$$\begin{aligned} &\forall x \in S,\, y \in T : F \ \text{ means } \ \forall x \in S : (\forall y \in T : F) \\ &\exists w, x, y, z \in S : F \ \text{ means } \ \exists w \in S : (\exists x \in S : (\exists y \in S : (\exists z \in S : F))) \end{aligned}$$

In the expression $\exists x \in S : F$, logicians say that x is a *bound variable* and that occurrences of x in F are *bound*. For example, n is a bound variable in the formula $\exists n \in Nat : n + 1 > n$, and the two occurrences of n in the subexpression $n+1 > n$ are bound. A variable x that's not bound is said to be *free*, and occurrences of x that are not bound are called *free* occurrences.

[6]Strictly speaking, $\in$ isn't an operator of predicate logic, so this isn't really a predicate logic tautology.

This terminology is rather misleading. A bound variable doesn't really occur in a formula because replacing it by some new variable doesn't change the formula. The two formulas

$$\exists\, n \in \mathit{Nat} : n + 1 > n \qquad \exists\, x \in \mathit{Nat} : x + 1 > x$$

are completely equivalent. Calling n a variable of the first formula is a bit like calling a a variable of that formula because it appears in the name Nat. Although misleading, this terminology is common and often convenient.

3 Specifying a Simple Clock

3.1 Behaviors

Before we try to specify a system, let's look at how scientists do it. For centuries, they have described a system with equations that determine how its state evolves with time, where the state consists of the values of variables. For example, the state of the system comprising the earth and the moon might be described by the values of the four variables e_pos, m_pos, e_vel, and m_vel, representing the positions and velocities of the two bodies. These values are elements in a 3-dimensional space. The earth-moon system is described by equations expressing the variables' values as functions of time and of certain constants—namely, their masses and initial positions and velocities.

A behavior of the earth-moon system consists of a function F from time to states, $F(t)$ representing the state of the system at time t. A computer system differs from the systems traditionally studied by scientists because we can pretend that its state changes in discrete steps. So, we represent the execution of a system as a sequence of states. Formally, we define a *behavior* to be a sequence of states, where a state is an assignment of values to variables. We specify a system by specifying a set of possible behaviors—the ones representing a correct execution of the system.

3.2 An Hour Clock

Let's start with a very trivial system—a digital clock that displays only the hour. To make the system completely trivial, we ignore the relation between the display and the actual time. The hour clock is then just a device whose display cycles through the values 1 through 12. Let the variable hr represent the clock's display. A typical behavior of the clock is the sequence

$$(2) \quad [hr = 11] \;\rightarrow\; [hr = 12] \;\rightarrow\; [hr = 1] \;\rightarrow\; [hr = 2] \;\rightarrow\; \cdots$$

of states, where $[hr = 11]$ is a state in which the variable hr has the value 11. A pair of successive states, such as $[hr = 1] \rightarrow [hr = 2]$, is called a *step*.

To specify the hour clock, we describe all its possible behaviors. We write an *initial predicate* that specifies the possible initial values of hr, and a *next-state relation* that specifies how the value of hr can change in any step.

We don't want to specify exactly what the display reads initially; any hour will do. So, we want the initial predicate to assert that hr can have any value from 1 through 12. Let's call the initial predicate $HCini$. We might informally define $HCini$ by:

$$HCini \;\triangleq\; hr \in \{1, \ldots, 12\}$$

Later, we'll see how to write this definition formally, without the "..." that stands for the informal *and so on*.

The next-state relation $HCnxt$ is a formula expressing the relation between the values of hr in the old (first) state and new (second) state of a step. We let hr represent the value of hr in the old state and hr' represent its value in the new state. (The $'$ in hr' is read *prime*.) We want the next-state relation to assert that hr' equals $hr + 1$ except if hr equals 12, in which case hr' should equal 1. Using an IF/THEN/ELSE construct with the obvious meaning, we can define *HCnxt* to be the next-state relation by writing:

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

HCnxt is an ordinary mathematical formula, except that it contains primed as well as unprimed variables. Such a formula is called an *action*. An action is true or false of a step. A step that satisfies the action *HCnxt* is called an *HCnxt step*.

When an *HCnxt* step occurs, we sometimes say that *HCnxt* is *executed*. However, it would be a mistake to take this terminology seriously. An action is a formula, and formulas aren't executed.

We want our specification to be a single formula, not the pair of formulas *HCini* and *HCnxt*. We write this formula with the temporal-logic operator $\Box$ (pronounced *box*). The temporal formula $\Box F$ asserts that formula F is always true. In particular, $\Box HCnxt$ is the assertion that *HCnxt* is true for every step in the behavior. A formula without a $\Box$ is an assertion about the beginning of the behavior. So, $HCini \wedge \Box HCnxt$ is true of a behavior iff the initial state satisfies *HCini* and every step satisfies *HCnxt*. This formula describes all behaviors like the one in (2); it seems to be the specification we're looking for.

If we considered the clock only in isolation and never tried to relate it to another system, then this would be a fine specification. However, suppose the clock is part of a larger system—for example, the hour display of a weather station that displays the current hour and temperature. The state of the station is described by two variables: *hr*, representing the hour display, and *tmp*, representing the temperature display. Consider this behavior of the weather station:

$$\begin{bmatrix} hr & = 11 \\ tmp & = 23.5 \end{bmatrix} \rightarrow \begin{bmatrix} hr & = 12 \\ tmp & = 23.5 \end{bmatrix} \rightarrow \begin{bmatrix} hr & = 12 \\ tmp & = 23.4 \end{bmatrix} \rightarrow \begin{bmatrix} hr & = 12 \\ tmp & = 23.3 \end{bmatrix} \rightarrow$$
$$\begin{bmatrix} hr & = 1 \\ tmp & = 23.3 \end{bmatrix} \rightarrow \cdots$$

In the second and third steps, *tmp* changes but *hr* remains the same. These steps are not allowed by $\Box HCnxt$, which asserts that every step must increment *hr*. The formula $HCini \wedge \Box HCnxt$ does not describe the hour clock in the weather station.

A formula that describes *any* hour clock must allow steps that leave *hr* unchanged—in other words, $hr' = hr$ steps. These are called *stuttering steps* of the clock. A specification of the hour clock should allow both *HCnxt* steps and stuttering steps. So, a step should be allowed iff it is either an *HCnxt* step or a stuttering step—that is, iff it is a step satisfying $HCnxt \vee (hr' = hr)$. This suggests that we adopt $HCini \wedge \Box(HCnxt \vee (hr' = hr))$ as our specification. In TLA, we let $[HCnxt]_{hr}$ stand for $HCnxt \vee (hr' = hr)$, so we can write the formula more compactly as $HCini \wedge \Box[HCnxt]_{hr}$.

The formula $HCini \wedge \Box[HCnxt]_{hr}$ does allow stuttering steps. In fact, it allows the behavior

$$[hr = 11] \;\rightarrow\; [hr = 12] \;\rightarrow\; [hr = 12] \;\rightarrow\; [hr = 12] \;\rightarrow\; [hr = 12] \;\rightarrow\; \cdots$$

that ends with an infinite sequence of stuttering steps. This behavior describes a clock whose display attains the value 12 and then keeps that value forever—in other words, a clock that stops at 12. In a like manner, we can represent a terminating execution of any system by an infinite behavior that ends with a sequence of nothing but stuttering steps. We have no need of finite behaviors (finite sequences of states), so we consider only infinite ones.

It's natural to require that a clock does not stop, so our specification should assert that there are infinitely many nonstuttering steps. Section 8 explains how to express this requirement. For now, we content ourselves with clocks that may stop, and we take as our specification of an hour clock the formula *HC* defined by

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

3.3 A Closer Look at the Hour-Clock Specification

A state is an assignment of values to variables, but what variables? The answer is simple: all variables. In the behavior (2), $[hr = 1]$ represents some particular state that assigns the value 1 to hr. It might assign the value 23 to the variable tmp and the value $\sqrt{-17}$ to the variable m_pos. We can think of a state as representing a potential state of the entire universe. A state that assigns 1 to hr and a particular point in 3-space to m_pos describes a state of the universe in which the hour clock reads 1 and the moon is in a particular place. A state that assigns $\sqrt{-2}$ to hr doesn't correspond to any state of the universe that we recognize, because the hour-clock can't display the value $\sqrt{-2}$. It might represent the state of the universe after a bomb fell on the clock, making its display purely imaginary.

A behavior is an infinite sequence of states—for example:

$$(3)\quad [hr = 11] \;\rightarrow\; [hr = 77.2] \;\rightarrow\; [hr = 78.2] \;\rightarrow\; [hr = \sqrt{-2}] \;\rightarrow\; \cdots$$

A behavior describes a potential history of the universe. The behavior (3) doesn't correspond to a history that we understand, because we don't know how the clock's display can change from 11 to 77.2. Whatever kind of history it represents is not one in which the clock is doing what it's supposed to.

Formula HC is a temporal formula. A temporal formula is an assertion about behaviors. We say that a behavior *satisfies* HC iff HC is a true assertion about the behavior. Behavior (2) satisfies formula HC. Behavior (3) does not, because HC asserts that every step satisfies $HCnxt$, and the first and third steps of (3) don't. (The second step, $[hr = 77.2] \rightarrow [hr = 78.2]$, does satisfy $HCnxt$.) We regard formula HC to be the specification of an hour clock because it is satisfied by exactly those behaviors that represent histories of the universe in which the clock functions properly.

If the clock is behaving properly, then its display should be an integer from 1 through 12. So, hr should be an integer from 1 through 12 in every state of any behavior satisfying the clock's specification, HC. Formula $HCini$ asserts that hr is an integer from 1 through 12, and $\Box HCini$ asserts that $HCini$ is always true. So, $\Box HCini$ should be true for any behavior satisfying HC. Another way of saying this is that HC implies $\Box HCini$, for any behavior. Thus, the formula $HC \Rightarrow \Box HCini$ should be satisfied by *every* behavior. A temporal formula satisfied by every behavior is called a *theorem*, so $HC \Rightarrow \Box HCini$ should be a theorem. It's easy to see that it is: HC implies that $HCini$ is true initially (in the first state of the behavior), and $\Box[HCnxt]_{hr}$ implies that each step either advances hr to its proper next value or else leaves hr unchanged. We can formalize this reasoning using the proof rules of TLA, but I'm not going to delve into proofs and proof rules.

3.4 The Hour-Clock Specification in TLA$^+$

Figure 1 shows how the hour clock specification can be written in TLA$^+$. There are two versions: the ASCII version on the bottom is the actual TLA$^+$ specification, the way you type it; the top version is typeset the way a "pretty-printer" might display it. Before trying to understand the specification, observe the relation between the two syntaxes:

- Reserved words that appear in small upper-case letters (like EXTENDS) are written in ASCII with ordinary upper-case letters.

- When possible, symbols are represented pictorially in ASCII—for example, $\Box$ is typed as `[]` and $\neq$ as `#`. (You can also type $\neq$ as `/=`.)

- When there is no good ASCII representation, TeX notation [5] is used—for example, $\in$ is typed as `\in`.

———————— MODULE *HourClock* ————————

EXTENDS *Naturals*

VARIABLE *hr*

$HCini \triangleq hr \in (1 \,..\, 12)$

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

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

————————————————————————

THEOREM $HC \Rightarrow \Box HCini$

————————————————————————

```
---------------------- MODULE HourClock ----------------------
EXTENDS Naturals
VARIABLE hr
HCini  ==  hr \in (1 .. 12)
HCnxt  ==  hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC  ==  HCini /\ [][HCnxt]_hr
--------------------------------------------------------------
THEOREM  HC => []HCini
==============================================================
```

Figure 1: The hour clock specification—typeset and ASCII versions.

A complete list of symbols and their ASCII equivalents appears at the end of this chapter in Figure 15. I will usually show the typeset version of a specification.

Now let's look at what the specification says. It starts with

———————— MODULE *HourClock* ————————

which begins a module named *HourClock*. TLA$^+$ specifications are partitioned into modules; the hour clock's specification consists of this single module.

Arithmetic operators like $+$ are not built into TLA$^+$, but are themselves defined in modules. (You might want to write a specification in which $+$ means addition of matrices rather than numbers.) The usual operators on natural numbers are defined in the *Naturals* module. Their definitions are incorporated into module *HourClock* by the statement

EXTENDS *Naturals*

Every symbol that appears in a formula must either be a built-in operator of TLA$^+$, or else it must be declared or defined. The statement

VARIABLE *hr*

declares *hr* to be a variable.

To define *HCini*, we need to express the set $\{1, \ldots, 12\}$ formally, without the ellipsis "$\ldots$". We can write this set out completely as

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

but that's tiresome. Instead, we use the operator $..$ defined in the *Naturals* modules to write this set as $1 \,..\, 12$. In general $i \,..\, j$ is the set of integers from i through j, for any integers i and j. (It equals the empty set if $j < i$.) It's now obvious how to write the definition of *HCini*. The definitions of *HCnxt* and *HC* are written just as before. (The ordinary mathematical operators of logic and set theory, like $\land$ and $\in$, and the IF construct are built into TLA$^+$.)

The line

can appear anywhere between statements; it's purely cosmetic and has no meaning. Following it is the statement

$$\text{THEOREM } HC \Rightarrow \Box HCini$$

of the theorem that was discussed above. This statement asserts that the formula $HC \Rightarrow \Box HCini$ is true in the context of the statement. More precisely, it asserts that the formula follows logically from the definitions in this module, the definitions in the *Naturals* module, and the rules of TLA$^+$. If the formula were not true, then the module would be incorrect.

The module is terminated by the symbol

The specification of the hour clock is the definition of HC, including the definitions of the formulas $HCnxt$ and $HCini$ and of the operators $..$ and $+$ that appear in the definition of HC. Formally, nothing in the module tells us that HC rather than $HCini$ is the clock's specification. TLA$^+$ is a language for writing mathematics—in particular, for writing mathematical definitions and theorems. What those definitions represent, and what significance we attach to those theorems, lies outside the scope of mathematics and therefore outside the scope of TLA$^+$. Engineering requires not just the ability to use mathematics, but the ability to understand what, if anything, the mathematics tells us about an actual system.

3.5 Another Way to Specify the Hour Clock

The *Naturals* module also defines the modulus operator, which we write %. The formula $i \,\%\, n$, which mathematicians write $i \bmod n$, is the remainder when i is divided by n. More formally, $i \,\%\, n$ is the natural number less than n satisfying $i = q * n + (i \,\%\, n)$ for some natural number q. Let's express this condition mathematically. The *Naturals* module defines Nat to be the set of natural numbers, and the assertion that there exists a q in the set Nat satisfying a formula F is written $\exists\, q \in Nat : F$. Thus, if i and n are elements of Nat and $n > 0$, then $i \,\%\, n$ is the unique number satisfying

$$(i \,\%\, n \in 0 \,..\, (n-1)) \land (\exists\, q \in Nat : i = q * n + (i \,\%\, n))$$

We can use % to simplify our hour-clock specification a bit. Observing that $(11 \,\%\, 12) + 1$ equals 12 and $(12 \,\%\, 12) + 1$ equals 1, we can define a different next-state action $HCnxt2$ and a different formula $HC2$ to be the clock specification:

$$HCnxt2 \triangleq hr' = (hr \,\%\, 12) + 1 \qquad HC2 \triangleq HCini \land \Box[HCnxt2]_{hr}$$

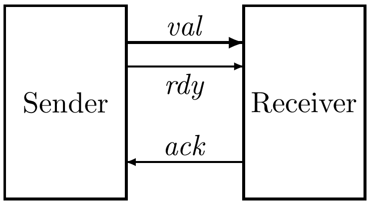
Actions $HCnxt$ and $HCnxt2$ are not equivalent. The step $[hr = 24] \rightarrow [hr = 25]$ satisfies $HCnxt$ but not $HCnxt2$, while the step $[hr = 24] \rightarrow [hr = 1]$ satisfies $HCnxt2$ but not $HCnxt$. However, any step starting in a state with hr in $1 \,..\, 12$ satisfies $HCnxt$ iff it satisfies $HCnxt2$. It's therefore not hard to deduce that any behavior starting in a state satisfying $HCini$ satisfies $\Box[HCnxt]_{hr}$ iff it satisfies $\Box[HCnxt2]_{hr}$. Hence, formulas HC and $HC2$ *are* equivalent. It doesn't matter which of them we take to be the specification of an hour clock.

Mathematics provides infinitely many ways of expressing the same thing. The expressions $6 + 6$, $3 * 4$, and $141 - 129$ all have the same meaning; they are just different ways of writing the number 12. We could replace either instance of the number 12 in module *HourClock* by any of these expressions without changing the meaning of any of the module's formulas.

When writing a specification, you will often be faced with a choice of how to express something. When that happens, you should first make sure that the choices yield equivalent specifications. If they do, then you can choose the one that you feel makes the specification easiest to understand. If they don't, then you must decide which one you mean.

4 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 acknowledgement (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.

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

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

(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 implementor 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 (i) the variables that represent the system's state and (ii) 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. Section 9 has more to say about these choices.

4.1 The First Specification

Now let's specify the interface with a module *AsynchInterface*. The variables *rdy* and *ack* can assume the values 0 and 1, which are natural numbers, so our module EXTENDS the *Naturals* module. 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 to first send 37, then send $\sqrt{-15}$, and then 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

$$\text{CONSTANT } \mathit{Data}$$

Our three variables are declared by

$$\text{VARIABLES } \mathit{val}, \mathit{rdy}, \mathit{ack}$$

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:

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

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.

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

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 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.

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

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 $\mathit{Send} \lor \mathit{Rcv}$ step. Therefore, *Next* is defined to equal $\mathit{Send} \lor \mathit{Rcv}$. 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 *act*. 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 $\mathit{rdy} = \mathit{ack}$. 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 $\mathit{val}' \in \mathit{Data}$. The value of *rdy* changes from 0 to 1 or from 1 to 0, so rdy' equals $1 - \mathit{rdy}$ (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 variables. So, we define *Send* by:

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

(I could have written $\mathit{ack}' = \mathit{ack}$ instead of UNCHANGED *ack*, but I prefer the UNCHANGED construct.)

A *Rcv* step is enabled iff *rdy* is different from *ack*; it complements the value of *ack* and leaves *rdy* and *ack* unchanged. Both *rdy* and *ack* are left unchanged iff the pair of values *rdy*, *ack* is left unchanged. TLA$^+$ uses angle brackets $\langle$ and $\rangle$ to enclose ordered tuples, so *Rcv* asserts that $\langle \mathit{rdy}, \mathit{ack} \rangle$ 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 2.

MODULE *AsynchInterface*

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

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

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

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

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

$$Next \triangleq Send \lor Rcv$$

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

THEOREM $Spec \Rightarrow \Box TypeInvariant$

Figure 2: The First Specification of an Asynchronous Interface

4.2 Another Specification

Module *AsynchInterface* is a fine description of the interface and its handshake protocol. However, it's not easy to use it to help specify a system that uses 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 components 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 `|->`.) The fields of records are not ordered, so 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:

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

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* component equals d and its *rdy* component equals $1 - chan.rdy$. In TLA$^+$, we can write this value as

$$[chan \;\textsc{except}\; !.val = d,\; !.rdy = 1 - chan.rdy]$$

Think of the ! as standing for *chan*, the record being modified by the EXCEPT expression. In the value replacing $!.rdy$, the symbol @ stands for $chan.rdy$, so we can write this expression as:

$$[chan \;\textsc{except}\; !.val = d,\; !.rdy = 1 - @]$$

In general, for any record r, the expression

$$[r \;\textsc{except}\; !.c_1 = e_1,\; \ldots\,,\; !.c_n = e_n]$$

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:

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

The definition of *Rcv* is straightforward. A value can be received when $chan.rdy \neq chan.ack$, and receiving the value complements $chan.ack$:

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

$$
\begin{array}{l}
\text{———————————— MODULE } \mathit{Channel} \text{ ————————————}\\
\text{EXTENDS } \mathit{Naturals}\\
\text{CONSTANT } \mathit{Data}\\
\text{VARIABLE } \mathit{chan}\\
\mathit{TypeInvariant} \triangleq \mathit{chan} \in [\mathit{val} : \mathit{Data},\ \mathit{rdy} : \{0,1\},\ \mathit{ack} : \{0,1\}]\\
\hline
\mathit{Init} \triangleq \land\ \mathit{TypeInvariant}\\
\qquad\quad \land\ \mathit{chan.ack} = \mathit{chan.rdy}\\
\mathit{Send}(d) \triangleq \land\ \mathit{chan.rdy} = \mathit{chan.ack}\\
\qquad\qquad\ \ \land\ \mathit{chan}' = [\mathit{chan} \text{ EXCEPT } !.\mathit{val} = d,\ !.\mathit{rdy} = 1 - @]\\
\mathit{Rcv} \qquad\ \triangleq \land\ \mathit{chan.rdy} \neq \mathit{chan.ack}\\
\qquad\qquad\ \ \land\ \mathit{chan}' = [\mathit{chan} \text{ EXCEPT } !.\mathit{ack} = 1 - @]\\
\mathit{Next} \triangleq (\exists\, d \in \mathit{Data} : \mathit{Send}(d)) \lor \mathit{Rcv}\\
\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{chan}}\\
\hline
\text{THEOREM } \mathit{Spec} \Rightarrow \Box\, \mathit{TypeInvariant}\\
\hline
\end{array}
$$

Figure 3: Our second specification of an asynchronous interface.

The complete specification appears in Figure 3.

We have now written two different specifications of the asynchronous interface. They are two different mathematical representations of the same physical system. In module *AsynchInterface*, we represented the system with the three variables *val*, *rdy*, and *ack*. In module *Channel*, we used a single variable *chan*. Since these two representations are at the same level of abstraction, they should, in some sense, be equivalent. Section 6.8 explains one sense in which they're equivalent.

4.3 Types: A Reminder

As defined in Section 4.1, a variable v has type T in specification *Spec* iff $v \in T$ is an invariant of *Spec*. Thus, *hr* has type $1 \mathrel{..} 12$ in the specification *HC* of the hour clock. This assertion does *not* mean that the variable *hr* can assume only values in the set $1 \mathrel{..} 12$. 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 $1 \mathrel{..} 12$.

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 7.2. 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.

4.4 Definitions

Let's examine what a definition means. If *Id* is a simple identifier like *Init* or *Spec*, then the definition $\mathit{Id} \triangleq \mathit{exp}$ defines *Id* to be synonymous with the expression *exp*. Replacing *Id* by *exp*, or vice-versa, in any expression e does not change the meaning of e. 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 $\mathit{Id}(p) \triangleq \mathit{exp}$, where *Id* and p are identifiers. For any expression e, this defines $\mathit{Id}(e)$ to be the expression obtained by substituting e for p in *exp*. For example, the definition of *Send* in the *Channel* module defines $\mathit{Send}(-5)$ to equal

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

$\mathit{Send}(e)$ is an expression, for any expression e. Thus, $\mathit{Send}(-5) \land (\mathit{chan.ack} = 1)$ is a formula. The identifier *Send* by itself is not an expression, and $\mathit{Send} \land (\mathit{chan.ack} = 1)$ 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. In the obvious way, we can define operators that take more than one argument, the general form being:

$$(4) \qquad \mathit{Id}(p_1, \ldots, p_n) \ \triangleq\ \mathit{exp}$$

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 rest of the module. 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 (4) implicitly includes a declarations of the identifiers $p_1, \ldots, p_n$ whose scope is the expression *exp*. An expression of the form

$$\exists\, v \in S : \mathit{exp}$$

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

$$(\exists\, v \in S : \mathit{exp}_1) \land (\exists\, v \in T : \mathit{exp}_2)$$

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

$$(\exists\, v \in S : (\mathit{exp}_1 \land \exists\, v \in T : \mathit{exp}_2))$$

is illegal because the declaration of v in the second $\exists\, v$ lies inside the scope of the 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.

4.5 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.

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 4: The hour clock specification with comments.

Figure 4 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 way of writing comments in the ASCII version. A comments may appear anywhere enclosed between `(*` and `*)`. The text of the comment itself may not contain `*)`, so these comments can't be nested. An end-of-line comment is preceded by `\*`.

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 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*.

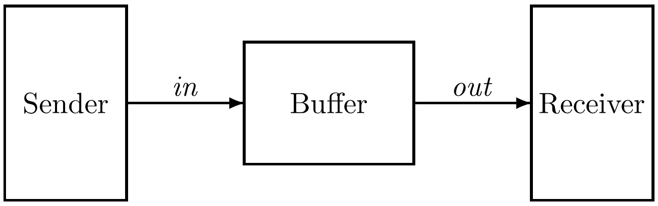
HCSpec A module that EXTENDS module *HCActions*, defines formula *HC*, and asserts the type-correctness theorem.

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 Section 5 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.

5 A FIFO

Our next example is a FIFO buffer, 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. The system's specification will allow behaviors with four kinds of non-stuttering steps: *Send* and *Rcv* actions on both the *in* channel and the *out* channel.

5.1 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 (all but the head of s). 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.[7] 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 5.3 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

[7] 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.

$Data$ and in substituted for $chan$. For example, this statement defines $InChan!TypeInvariant$ to equal

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

We introduce our second instance of the $Channel$ module with the analogous statement:

$$OutChan \ \triangleq\ \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Message,\ chan \leftarrow out$$

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

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

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:

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

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

$SSend(msg)$	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 $FIFO$ of Figure 5.

5.2 Instantiation Examined

5.2.1 Parameterized 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 $InnerFIFO$:

$$Chan(ch) \ \triangleq\ \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Message, chan \leftarrow ch$$

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

MODULE *InnerFIFO*

EXTENDS $\mathit{Naturals}, \mathit{Sequences}$
CONSTANT $\mathit{Message}$
VARIABLES $\mathit{in}, \mathit{out}, q$

$\mathit{InChan} \triangleq$ INSTANCE $\mathit{Channel}$ WITH $\mathit{Data} \leftarrow \mathit{Message}, \mathit{chan} \leftarrow \mathit{in}$
$\mathit{OutChan} \triangleq$ INSTANCE $\mathit{Channel}$ WITH $\mathit{Data} \leftarrow \mathit{Message}, \mathit{chan} \leftarrow \mathit{out}$

$\mathit{Init} \triangleq \land\ \mathit{InChan}!\mathit{Init}$
$\qquad \land\ \mathit{OutChan}!\mathit{Init}$
$\qquad \land\ q = \langle\,\rangle$

$\mathit{TypeInvariant} \triangleq \land\ \mathit{InChan}!\mathit{TypeInvariant}$
$\qquad \land\ \mathit{OutChan}!\mathit{TypeInvariant}$
$\qquad \land\ q \in \mathit{Seq}(\mathit{Message})$

$\mathit{SSend}(\mathit{msg}) \triangleq \land\ \mathit{InChan}!\mathit{Send}(\mathit{msg})$ — Send msg on channel in.
$\qquad \land$ UNCHANGED $\langle \mathit{out}, q\rangle$

$\mathit{BufRcv} \triangleq \land\ \mathit{InChan}!\mathit{Rcv}$ — Receive message from channel in
$\qquad \land\ q' = \mathit{Append}(q, \mathit{in}.\mathit{val})$ — and append it to tail of q.
$\qquad \land$ UNCHANGED out

$\mathit{BufSend} \triangleq \land\ q \neq \langle\,\rangle$ — Enabled only if q is nonempty.
$\qquad \land\ \mathit{OutChan}!\mathit{Send}(\mathit{Head}(q))$ — Send $\mathit{Head}(q)$ on channel out
$\qquad \land\ q' = \mathit{Tail}(q)$ — and remove it from q.
$\qquad \land$ UNCHANGED in

$\mathit{RRcv} \triangleq \land\ \mathit{OutChan}!\mathit{Rcv}$ — Receive message from channel out.
$\qquad \land$ UNCHANGED $\langle \mathit{in}, q\rangle$

$\mathit{Next} \triangleq \lor\ \exists\, \mathit{msg} \in \mathit{Message} : \mathit{SSend}(\mathit{msg})$
$\qquad \lor\ \mathit{BufRcv}$
$\qquad \lor\ \mathit{BufSend}$
$\qquad \lor\ \mathit{RRcv}$

$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{in}, \mathit{out}, q\rangle}$

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

Figure 5: The specification of a FIFO, with the internal variable q visible.

5.2.2 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

$$InChan \triangleq \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Data, chan \leftarrow in$$

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

$$InChan \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow in$$

We know there is an implied $Data \leftarrow Data$ substitution because an INSTANCE statement must have a substitution for every parameter (CONSTANT or VARIABLE) 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.

5.2.3 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 parameterized 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

$$\text{INSTANCE } Channel \text{ WITH } Data \leftarrow D, chan \leftarrow x$$

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*. Of course, 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.

5.3 Hiding the Queue

Module *InnerFIFO* of Figure 5 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*. My picture of the FIFO system 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. In the final specification, it should be hidden.

In TLA, we hide a variable with the existential quantifier $\exists\!\!\!\exists$ of temporal logic. The formula $\exists\!\!\!\exists\, x : F$ 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 $\exists\!\!\!\exists$ is defined more precisely in Section 8.6.)

The obvious thing to do now is to define the FIFO specification to be the formula $\exists\!\!\!\exists\, q : Spec$. However, we can't put this definition in module *InnerFIFO* because q is already declared there, and a formula $\exists\!\!\!\exists\, q : \ldots$ would redeclare it. Instead, we use a new module with a parametrized instantiation of the *InnerFIFO* module (see Section 5.2.1 above):

```
------------------------------ 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

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

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*.

5.4 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 be enabled only when there are fewer than N messages in the buffer—that is, only when $Len(q)$ 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 6 contains the specification. It introduces the new constant parameter N. It also contains the statement

$$\text{ASSUME } (N \in Nat) \land (N > 0)$$

which asserts the assumption that N is a positive natural number. 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

MODULE *BoundedFIFO*

VARIABLES in, out
CONSTANT $Message, N$
ASSUME $(N \in Nat) \land (N > 0)$

$$Inner(q) \triangleq \text{INSTANCE } InnerFIFO$$

$$BNext(q) \triangleq \land\ Inner(q)!Next \\ \qquad\qquad\quad \land\ Inner(q)!BufRcv \Rightarrow (Len(q) < N)$$

$$Spec \triangleq \exists\, q : Inner(q)!Init \land \Box[BNext(q)]_{\langle in, out, q\rangle}$$

Figure 6: Specification of a FIFO buffer of length N.

its assumptions imply its theorems. It's a good idea to assert this kind of simple assumption about constants.

An ASSUME statement should only be used to assert 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 4.3, 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 contradict the laws of TLA.

You may wonder why module *BoundedFIFO* asserts that N is a positive natural, but doesn't assert that *Message* is a set. Similarly, why didn't we have to specify that the constant parameter *Data* in our asynchronous interface specifications is a set? The answer is that, in TLA$^+$, every value is a set.[8] 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 assert 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. (Writing a formula that asserts that a set is finite is a nice exercise in TLA$^+$.) However, most specifications make perfect sense for infinite sets of messages or processors, so there is no reason to require these sets to be finite.

5.5 What We're Specifying

I wrote above, at the beginning of this section, 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 behave correctly. It's often helpful in understanding a specification to indicate explicitly which steps are system steps and which

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

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

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

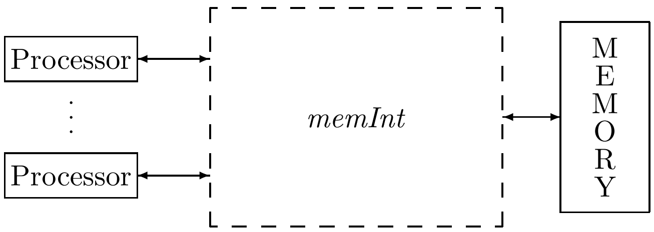
While suggestive, this way of defining the next-state action has no formal significance. The specification *Spec* equals $Init \land \Box[Next]_{\ldots}$; changing the way we structure the definition of *Next* doesn't change its meaning. If a behavior fails to satisfy *Spec*, nothing tells us if the 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. How to write open-system specifications is described elsewhere [1].

Open-system specifications are philosophically more satisfying. However, closed-system specifications are a little bit 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 little reason to do so.

6 A Caching Memory

A memory system consists of a set of processors connected to a memory by some abstract interface, which we label *memInt*.



In this section we specify what the memory is supposed to do, then we specify a particular implementation of the memory using caches. We begin by specifying the memory interface, which is common to both specifications.

6.1 The Memory Interface

The asynchronous interface described in Section 4 uses a handshake protocol. Receipt of a data value must be acknowledged before the next data value can be sent. In the memory interface, we abstract away this kind of detail and represent both the sending of a data value and its receipt as a single step. We call it a *Send* step if a processor is sending the value to the memory; it's a *Reply* step if the memory is sending to a processor. Processors do not send values to one another, and the memory sends to only one processor at a time.

We represent the state of the memory interface by the value of the variable *memInt*. A *Send* step changes *memInt* in some way, but we don't want to specify exactly how. The way to leave something unspecified in a specification is to make it a parameter. For example, in the bounded FIFO of Section 5.4, we left the size of the buffer unspecified by making it a parameter

N. We'd therefore like to declare a parameter *Send* so that $Send(p, d)$ describes how *memInt* is changed by a step that represents processor p sending data value d to the memory. However, TLA$^+$ provides only CONSTANT and VARIABLE parameters, not action parameters.[9] So, we declare *Send* to be a constant operator and write $Send(p, d, memInt, memInt')$ instead of $Send(p, d)$.

In TLA$^+$, we declare *Send* to be a constant operator that takes four arguments by writing

$$\text{CONSTANT } Send(\_, \_, \_, \_)$$

This means that $Send(p, d, miOld, miNew)$ is an expression, for any expressions p, d, *miOld*, and *miNew*, but it says nothing about what the value of that expression is. We want it to be a Boolean value that is true iff a step in which *memInt* equals *miOld* in the first state and *miNew* in the second state represents the sending by p of value d to the memory.[10] We can assert that the value is a Boolean by the assumption:

$$\text{ASSUME } \forall\, p,\ d,\ miOld,\ miNew\ :\ Send(p, d, miOld, miNew) \in \text{BOOLEAN}$$

This asserts that the formula

$$Send(p, d, miOld, miNew) \in \text{BOOLEAN}$$

is true for all values of p, d, *miOld*, and *miNew*. The built-in symbol BOOLEAN denotes the set {TRUE, FALSE}, whose elements are the two Boolean values TRUE and FALSE.

This ASSUME statement asserts formally that the value of

$$Send(p, d, miOld, miNew)$$

is a Boolean. But the only way to assert formally what that value signifies would be to say what it actually equals—that is, to define *Send* rather than making it a parameter. We don't want to do that, so we just state informally what the value means. This statement is part of the intrinsically informal description of the relation between our mathematical abstraction and a physical memory system.

To allow the reader to understand the specification, we have to describe informally what *Send* means. The ASSUME statement asserting that $Send(\ldots)$ is a Boolean is then superfluous as an explanation. However, it could help tools understand the specification, so it's a good idea to include it anyway.

A specification that uses the memory interface can use the operators *Send* and *Reply* to specify how the variable *memInt* changes. The specification must also describe *memInt*'s initial value. We therefore declare a constant parameter *InitMemInt* that is the set of possible initial values of *memInt*.

We also introduce three constant parameters that are needed to describe the interface:

Proc The set of processor identifiers. (We usually shorten *processor identifier* to *processor* when referring to an element of *Proc*.)

Adr The set of memory addresses.

Val The set of possible memory values that can be assigned to an address.

---

[9] Even if TLA$^+$ allowed us to declare an action parameter, we would have no way to specify that a $Send(p, d)$ action constrains only *memInt* and not other variables.

[10] We expect $Send(p, d, miOld, miNew)$ to have this meaning only when p is a processor and d a value that p is allowed to send, but we simplify the specification a bit by requiring it to be a Boolean for all values of p and d.

MODULE *MemoryInterface*

VARIABLE $memInt$

CONSTANTS $Send(\_,\_,\_,\_)$, A $Send(p, d, memInt, memInt')$ step represents processor p sending value d to the memory.

$Reply(\_,\_,\_,\_)$, A $Reply(p, d, memInt, memInt')$ step represents the memory sending value d to processor p.

$InitMemInt$, The set of possible initial values of $memInt$.

$Proc$, The set of processor identifiers.

Adr, The set memory addresses.

Val The set memory values.

$$\text{ASSUME } \forall\, p, d, miOld, miNew \;:\; \land\; Send(p, d, miOld, miNew) \in \text{BOOLEAN}$$
$$\land\; Reply(p, d, miOld, miNew) \in \text{BOOLEAN}$$

$MReq \;\triangleq\; [op : \{\text{“Rd”}\},\ adr : Adr] \;\cup\; [op : \{\text{“Wr”}\},\ adr : Adr,\ val : Val]$

The set of all requests; a read specifies an address, a write specifies an address and a value.

$NoVal \;\triangleq\; \text{CHOOSE } v : v \notin Val$ An arbitrary value not in Val.

Figure 7: The Specification of a Memory Interface

Finally, we define the values that the processors and memory send to one another over the interface. A processor sends a request to the memory. We represent a request as a record with an *op* field that specifies the type of request and additional fields that specify its arguments. Our simple memory allows just read and write requests. A read request has *op* field “Rd” and an *adr* field specifying the address to be read. The set of all read requests is therefore the set

$$[op : \{\text{“Rd”}\},\ adr : Adr]$$

of all records whose *op* field equals “Rd” (is an element of the set {“Rd”} whose only element is the string “Rd”) and whose *adr* field is an element of Adr. A write request must specify the address to be written and the value to write. It is represented by a record with *op* field equal to “Wr”, and with *adr* and *val* fields specifying the address and value. We define $MReq$, the set of all requests, to equal the union of these two sets. (Set operations, including union, are described in Section 2.2.)

The memory responds to a read request with the memory value it read. It must also respond to a write request; and it seems nice to let the response be different from the response to any read request. We therefore require the memory to respond to a write request by returning a value $NoVal$ that is different from any memory value. We could declare $NoVal$ to be a constant parameter and add the assumption $NoVal \notin Val$. (The symbol $\notin$ is typed in ASCII as `\notin`.) But it's best, when possible, to avoid introducing parameters. Instead, we define $NoVal$ by:

$$NoVal \;\triangleq\; \text{CHOOSE } v : v \notin Val$$

The expression CHOOSE $x : F$ equals an arbitrarily chosen value x that satisfies the formula F. (If no such x exists, the expression has a completely arbitrary value.) This statement defines $NoVal$ to be some value that is not an element of Val. We have no idea what the value of $NoVal$ is; we just know what it isn't—namely, that it isn't an element of Val. The CHOOSE operator is discussed in Section 7.6.

The complete memory interface specification is module *MemoryInterface* in Figure 7.

6.2 Functions

A memory assigns values to addresses. The state of the memory is therefore an assignment of elements of *Val* (memory values) to elements of *Adr* (memory addresses). In a programming language, such an assignment is called an array of type *Val* indexed by *Adr*. In mathematics, it's called a function from *Adr* to *Val*. Before writing the memory specification, let's look at the mathematics of functions, and how it is described in TLA$^+$.

A function f has a domain, written DOMAIN f, and it assigns to each element x of its domain the value $f[x]$. (Mathematicians write this as $f(x)$, but TLA$^+$ uses the array notation of programming languages, with square brackets.) Two functions f and g are equal iff they have the same domain and $f[x] = g[x]$ for all x in their domain.

The *range* of a function f is the set of all elements of the form $f[x]$ with x in DOMAIN f. For any sets S and T, the set of all functions whose domain equals S and whose range is any subset of T is written $[S \rightarrow T]$.

Ordinary mathematics does not have a convenient notation for writing an expression whose value is a function. TLA$^+$ defines $[x \in S \mapsto e]$ to be the function f with domain S such that $f[x] = e$ for every $x \in S$.[11] For example,

$$succ \;\triangleq\; [n \in Nat \mapsto n + 1]$$

defines *succ* to be the successor function on the natural numbers—the function with domain *Nat* such that $succ[n] = n + 1$ for all $n \in Nat$.

A record is a function whose domain is a finite set of strings. For example, a record with *val*, *ack*, and *rdy* fields is a function whose domain is the set {"val", "ack", "rdy"} consisting of the three strings "val", "ack", and "rdy". The expression $r.ack$, the *ack* field of a record r, is an abbreviation for $r[\text{"ack"}]$. The record

$$[val \mapsto 42,\; ack \mapsto 1,\; rdy \mapsto 0]$$

can be written

$$\begin{array}{l}[i \in \{\text{"val"}, \text{"ack"}, \text{"rdy"}\} \mapsto \\ \quad \text{IF } i = \text{"val" THEN } 42 \text{ ELSE IF } i = \text{"ack" THEN } 1 \text{ ELSE } 0]\end{array}$$

The EXCEPT construct for records, explained in Section 4.2, is a special case of a general EXCEPT construct for functions, where $!.c$ is an abbreviation for $![\text{"c"}]$. For any function f, the expression $[f \text{ EXCEPT } ![c] = e]$ is the function $\hat{f}$ that is the same as f except with $\hat{f}[c] = e$. This function can also be written:

$$[x \in \text{DOMAIN } f \;\mapsto\; \text{IF } x = c \text{ THEN } e \text{ ELSE } f[x]]$$

assuming that the symbol x does not occur in any of the expressions f, c, and e. For example, $[succ \text{ EXCEPT } ![42] = 86]$ is the function g that is the same as *succ* except that $g[42]$ equals 86 instead of 43.

As in the EXCEPT construct for records, the expression e in

$$[f \text{ EXCEPT } ![c] = e]$$

can contain the symbol @, where it means $f[c]$. For example,

$$[succ \text{ EXCEPT } ![42] = 2 * @] \;=\; [succ \text{ EXCEPT } ![42] = 2 * succ[42]]$$

[11] Computer scientists write $\lambda x : S.e$ to mean something very much like $[x \in S \mapsto e]$. Such λ expressions aren't quite the same as the functions of ordinary mathematics, so TLA$^+$ doesn't use that notation for writing functions.

In general,

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

is the function $\hat{f}$ that is the same as f except with $\hat{f}[c_i] = e_i$ for each i. More precisely, this expression equals

$$[\ldots[[f \text{ EXCEPT } ![c_1] = e_1] \text{ EXCEPT } ![c_2] = e_2] \ \ldots \ \text{EXCEPT } ![c_n] = e_n]$$

Functions correspond to the arrays of programming languages. The domain of a function corresponds to the index set of an array. The function $[f \text{ EXCEPT } ![c] = e]$ corresponds to the array obtained from f by assigning e to $f[c]$. A function whose range is a set of functions corresponds to an array of arrays. TLA$^+$ defines $[f \text{ EXCEPT } ![c][d] = e]$ to be the function corresponding to the array obtained by assigning e to $f[c][d]$. It can be written as

$$[f \text{ EXCEPT } ![c] = [@ \text{ EXCEPT } ![d] = e]]$$

The generalization to $[f \text{ EXCEPT } ![c_1]\ldots[c_n] = e]$ for any n should be obvious. Since a record is a function, this notation can be used for records as well. TLA$^+$ uniformly maintains the notation that $\sigma.c$ is an abbreviation for $\sigma[\text{“c”}]$. For example, this implies:

$$[f \text{ EXCEPT } ![c].d = e] \quad = \quad [f \text{ EXCEPT } ![c] = [@ \text{ EXCEPT } !.d = e]]$$

The TLA$^+$ definition of records as functions makes it possible to manipulate them in ways that have no counterparts in programming languages. For example, we can define an operator R such that $R(r, s)$ is the record obtained from r by replacing the value of each field c that is also a field of the record s with $s.c$. In other words, for every field c of r, if c is a field of s then $R(r, s).c = s.c$; otherwise $R(r, s).c = r.c$. The definition is:

$$R(r, s) \ \triangleq \ [c \in \text{DOMAIN } r \mapsto \text{IF } \ c \in \text{DOMAIN } s \ \text{ THEN } \ s[c] \ \text{ ELSE } \ r[c]]$$

All the TLA$^+$ function constructs have generalizations for functions with multiple arguments. We describe only functions of a single argument here.

6.3 A Linearizable Memory Specification

We specify a very simple memory system in which a processor p issues a memory request and then waits for a response before issuing the next request. In our specification, the request is executed by accessing (reading or modifying) a variable *mem*, which represents the current state of the memory. Because the memory can receive requests from other processors before responding to processor p, it matters when *mem* is accessed. We let the access of *mem* occur any time between the request and the response. This specifies what is called a *linearizable* memory [4]. (The more commonly specified *sequentially consistent* memory [9] allows the access of *mem* to occur at any time.)

In addition to *mem*, the specification has the internal variables *ctl* and *buf*, where $ctl[p]$ describes the status of processor p's request and $buf[p]$ contains either the request or the response. Consider the request *req* that equals

$$[op \mapsto \text{“Wr”}, \ adr \mapsto a, \ val \mapsto v]$$

It is a request to write v to memory address a, and it generates the response *NoVal*. The processing of this request is represented by the following three steps:

$$\begin{bmatrix} ctl[p] & = & \text{“rdy”} \\ buf[p] & = & \cdots \\ mem[a] & = & \cdots \end{bmatrix} \xrightarrow{Req(p)} \begin{bmatrix} ctl[p] & = & \text{“busy”} \\ buf[p] & = & req \\ mem[a] & = & \cdots \end{bmatrix} \xrightarrow{Do(p)} \begin{bmatrix} ctl[p] & = & \text{“done”} \\ buf[p] & = & NoVal \\ mem[a] & = & v \end{bmatrix}$$

$$\xrightarrow{Rsp(p)} \begin{bmatrix} ctl[p] & = & \text{“rdy”} \\ buf[p] & = & NoVal \\ mem[a] & = & v \end{bmatrix}$$

A $Req(p)$ step represents the issuing of a request by processor p. It is enabled when $ctl[p] =$ "rdy"; it sets $ctl[p]$ to "busy" and sets $buf[p]$ to the request. A $Do(p)$ step represents the memory access; it is enabled when $ctl[p] =$ "busy" and it sets $ctl[p]$ to "done" and $buf[p]$ to the response. A $Rsp(p)$ step represents the memory's response to p; it is enabled when $ctl[p] =$ "done" and it sets $ctl[p]$ to "rdy".

Writing the specification is a straightforward exercise in representing these changes to the variables in TLA$^+$ notation. The internal specification, with *mem*, *ctl*, and *buf* visible (free variables), appears in module *InternalMemory* in Figure 8. The memory specification, which hides the three internal variables, is module *Memory* in Figure 9.

6.4 Tuples as Functions

Before writing our caching memory specification, let's take a closer look at tuples. Recall that $\langle a, b, c \rangle$ is the 3-tuple with components a, b, and c. In TLA$^+$, this 3-tuple is actually the function with domain $\{1, 2, 3\}$ that maps 1 to a, 2 to b, and 3 to c. Thus, $\langle a, b, c \rangle[2]$ equals b.

TLA$^+$ provides the Cartesian product operator $\times$ of ordinary mathematics, where $A \times B \times C$ is the set of all 3-tuples $\langle a, b, c \rangle$ such that $a \in A$, $b \in B$, and $c \in C$. Note that $A \times B \times C$ is different from $A \times (B \times C)$, which is the set of pairs $\langle a, p \rangle$ with a in A and p in the set of pairs $B \times C$.

The *Sequences* module defines finite sequences to be tuples. Hence, a sequence of length n is a function with domain $1 \,..\, n$. In fact, s is a sequence iff it equals $[i \in 1 \,..\, Len(s) \mapsto s[i]]$. Below are a few operator definitions from the *Sequences* module. (The meanings of the operators are described in Section 5.1.)

$$
\begin{array}{lcl}
Head(s) & \triangleq & s[1] \\
Tail(s) & \triangleq & [i \in 1 \,..\, (Len(s) - 1) \mapsto s[i+1]] \\
s \circ t & \triangleq & [i \in 1 \,..\, (Len(s) + Len(t)) \mapsto \\
 & & \quad \text{IF } i \leq Len(s) \text{ THEN } s[i] \text{ ELSE } t[i - Len(s)]]
\end{array}
$$

6.5 Recursive Function Definitions

We need one more tool to write the caching memory specification: recursive function definitions. Recursively defined functions are familiar to programmers. The classic example is the factorial function, which I'll call *fact*. It's usually defined by writing:

$$fact[n] \;=\; \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1]$$

for all $n \in Nat$. The TLA$^+$ notation for writing functions suggests trying to define *fact* by

$$fact \;\triangleq\; [n \in Nat \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1]]$$

This definition is illegal because the occurrence of *fact* to the right of the $\triangleq$ is undefined—*fact* is defined only after its definition.

TLA$^+$ does allow the apparent circularity of recursive function definitions. We can define the factorial function *fact* by:

$$fact[n \in Nat] \;\triangleq\; \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1]$$

In general, a definition of the form $f[x \in S] \triangleq e$ can be used to define recursively a function f with domain S. Section 7.3 explains exactly what such a definition means. For now, we will just write recursive definitions without worrying about their meaning.

MODULE *InternalMemory*

EXTENDS *MemoryInterface*
VARIABLES mem, ctl, buf

$IInit \triangleq$ The initial predicate
$\land\ mem \in [Adr \rightarrow Val]$ Initially, memory locations have any legal values,
$\land\ ctl = [p \in Proc \mapsto \text{“rdy”}]$ each processor is ready to issue requests,
$\land\ buf = [p \in Proc \mapsto NoVal]$ each $buf[p]$ is arbitrarily initialized to $NoVal$,
$\land\ memInt \in InitMemInt$ and $memInt$ is any element of $InitMemInt$.

$TypeInvariant \triangleq$ The type-correctness invariant.
$\land\ mem \in [Adr \rightarrow Val]$ mem is a function from Adr to Val.
$\land\ ctl \in [Proc \rightarrow \{\text{“rdy”}, \text{“busy”}, \text{“done”}\}]$ $ctl[p]$ equals “rdy”, “busy”, or “done”.
$\land\ buf \in [Proc \rightarrow MReq \cup Val \cup \{NoVal\}]$ $buf[p]$ is a request or a response.

$Req(p) \triangleq$ Processor p issues a request.
$\land\ ctl[p] = \text{“rdy”}$ Enabled iff p is ready to issue a request.
$\land\ \exists\, req \in MReq :$ For some request req:
  $\land\ Send(p, req, memInt, memInt')$ Send req on the interface.
  $\land\ buf' = [buf \text{ EXCEPT } ![p] = req]$ Set $buf[p]$ to the request.
  $\land\ ctl' = [ctl \text{ EXCEPT } ![p] = \text{“busy”}]$ Set $ctl[p]$ to “busy”.
$\land$ UNCHANGED mem

$Do(p) \triangleq$ Perform p's request to memory.
$\land\ ctl[p] = \text{“busy”}$ Enabled iff p's request is pending.
$\land\ mem' = \text{IF } buf[p].op = \text{“Wr”}$
  $\text{THEN } [mem \text{ EXCEPT}$ Write to memory on a “Wr” request.
    $![buf[p].adr] = buf[p].val]$
  $\text{ELSE } mem$ Leave mem unchanged on a “Rd” request.
$\land\ buf' = [buf \text{ EXCEPT } ![p] = \text{IF } buf[p].op = \text{“Wr”}$ Set $buf[p]$ to the response:
  $\text{THEN } NoVal$ $NoVal$ for a write;
  $\text{ELSE } mem[buf[p].adr]\,]$ the memory value for a read.
$\land\ ctl' = [ctl \text{ EXCEPT } ![p] = \text{“done”}]$ Set $ctl[p]$ to “done”.
$\land$ UNCHANGED $memInt$

$Rsp(p) \triangleq$ Return the response to p's request.
$\land\ ctl[p] = \text{“done”}$ Enabled iff request is done but response not sent.
$\land\ Reply(p, buf[p], memInt, memInt')$ Send the response on the interface.
$\land\ ctl' = [ctl \text{ EXCEPT } ![p] = \text{“rdy”}]$ Set $ctl[p]$ to “rdy”.
$\land$ UNCHANGED $\langle mem, buf \rangle$

$INext \triangleq \exists\, p \in Proc : Req(p) \lor Do(p) \lor Rsp(p)$ The next-state action.

$ISpec \triangleq IInit \land \Box[INext]_{\langle memInt, mem, ctl, buf \rangle}$ The specification.

THEOREM $ISpec \Rightarrow \Box TypeInvariant$

Figure 8: The internal memory specification.

——— MODULE *Memory* ———

EXTENDS *MemoryInterface*

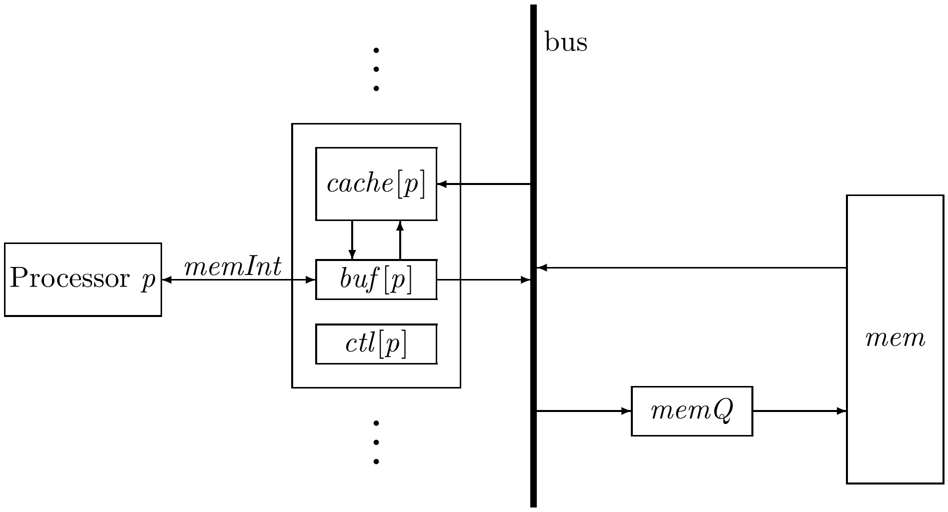
$Inner(mem, ctl, buf) \;\triangleq\;$ INSTANCE *InternalMemory*

$Spec \;\triangleq\; \exists\, mem, ctl, buf \,:\, Inner(mem, ctl, buf)!ISpec$

Figure 9: The memory specification.

6.6 A Write-Through Cache

We now specify a simple write-through cache that implements the memory specification. The system is described by the following picture:



Each processor p communicates with a local controller, which maintains three state components: $buf[p]$, $ctl[p]$, and $cache[p]$. The value of $cache[p]$ represents the processor's cache; $buf[p]$ and $ctl[p]$ play the same role as in the internal memory specification of module *InternalMemory*. (However, as we will see below, $ctl[p]$ can assume an additional value "waiting".) These local controllers communicate with the main memory *mem*,[12] and with one another, over a bus. Requests from the processors to the main memory are in the queue *memQ* of maximum length *QLen*.

A write request by processor p is performed by the action $DoWr(p)$. This is a write-through cache, meaning that every write request updates main memory. So, the $DoWr(p)$ action writes the value into $cache[p]$ and adds the write request to the tail of *memQ*. It also updates $cache[q]$ for any other processor q that has a copy of the address in its cache. When the request reaches the head of *memQ*, the action *MemQWr* stores the value in *mem*.

A read request by processor p is performed by the action $DoRd(p)$, which obtains the value from the cache. If the value is not in the cache, the action $RdMiss(p)$ adds the request to the tail of *memQ* and sets $ctl[p]$ to "waiting". When the enqueued request reaches the head of *memQ*, the action *MemQRd* reads the value and puts it in $cache[p]$, enabling the $DoRd(p)$ action.

We might expect the *MemQRd* action to read the value from *mem*. However, this could cause an error if there is a write to that address enqueued in *memQ* behind the read request. In that case, reading the value from memory could lead to two processors having different values for the address in their caches: the one that issued the read request, and the one that issued the write request that followed the read in *memQ*. So, the *MemQRd* action must read

[12] This main memory does not directly correspond to the memory represented by variable *mem* in module *InternalMemory*.

the value from the last write to that address in *memQ*, if there is such a write; otherwise, it reads the value from *mem*.

Eviction of an address from processor p's cache is represented by a separate $Evict(p)$ action. Since all cached values have been written to memory, eviction does nothing but remove the address from the cache. There is no reason to evict an address until the space is needed, so this action would be executed only when a request for an uncached address is received from p and p's cache is full. But that's a performance optimization; it doesn't affect the correctness of the algorithm. We allow a cached address to be evicted from p's cache at any time—except if the address was just put there by a *MemQRd* action for the current request. This is the case when $ctl[p]$ equals "waiting" and $buf[p].adr$ equals the cached address.

The actions $WReq(p)$ and $WRsp(p)$, which represent processor p issuing a request and the memory issuing a reply to p, are the same as the corresponding actions of the memory specification, except that they also leave the new variables *cache* and *memQ* unchanged.

To specify all these actions, we must decide how the processor caches and the queue of requests to memory are represented by the variables *memQ* and *cache*. We let *memQ* be a sequence of pairs of the form $\langle p, req \rangle$, where *req* is a request and p is the processor that issued it. For any memory address a, we let $cache[p][a]$ be the value in p's cache for address a (the "copy" of a in p's cache). If p's cache does not have a copy of a, we let $cache[p][a]$ equal *NoVal*.

The specification appears in module *WriteThroughCache* in Figures 10 and 11. I'll now go through this specification, explaining some of the finer points and some notation that we haven't encountered before.

The EXTENDS, declaration statements, and ASSUME are familiar. We can re-use some of the definitions from the *InternalMemory* module, so an INSTANCE statement instantiates a copy of that module. (The parameters of module *InternalMemory* are instantiated by the parameters of the same name in module *WriteThroughCache*.)

The initial predicate *Init* contains the conjunct $M!IInit$, which asserts that *mem*, *ctl*, and *buf* have the same initial values as in the internal memory specification. The write-through cache allows $ctl[p]$ to have the value "waiting" that it didn't in the internal memory specification, so we can't re-use the internal memory's type invariant $M!TypeInvariant$. Formula *TypeInvariant* therefore explicitly describes the types of *mem*, *ctl*, and *buf*. The type of *memQ* is the set of sequences of $\langle processor, request \rangle$ pairs.

The module next defines the predicate *Coherence*, which asserts the basic cache coherence property of the write-through cache: for any processors p and q and any address a, if p and q each has a copy of address a in its cache, then those copies are equal. Note the trick of writing $x \notin \{y, z\}$ instead of the equivalent but longer formula $(x \neq y) \wedge (x \neq z)$.

The state function *vmem* is used in defining action *MemQRd* below. It is defined to equal the value that the main memory *mem* will have after all the write operations currently in *memQ* have been performed. Recall that the value read by *MemQRd* must be the most recent one written to that address—a value that may still be in *memQ*. That value is the one in *vmem*. The definition of *vmem* introduces the TLA$^+$ LET/IN construct. A LET clause consists of a sequence of definitions, whose scope extends until the end of the IN clause. In the definition of *vmem*, the LET clause defines the function f so that $f[i]$ is the value *mem* will have after the first i operations in *memQ* have been performed. Note that $memQ[i][2]$ is the second component (the request) of $memQ[i]$, the ith element in the sequence *memQ*.

The actions $Req(p)$ and $Rsp(p)$, which represent a processor sending a request and receiving a reply, are essentially the same as the corresponding actions in module *InternalMemory*. The only difference is that they must specify that the variables *cache* and *memQ*, not present in module *InternalMemory*, are left unchanged.

In the definition of *RdMiss*, the expression $Append(memQ, \langle p, buf[p] \rangle)$ is the sequence obtained by appending the element $\langle p, buf[p] \rangle$ to the end of *memQ*.

MODULE *WriteThroughCache*

EXTENDS *Naturals*, *Sequences*, *MemoryInterface*
VARIABLES *mem*, *ctl*, *buf*, *cache*, *memQ*
CONSTANT *QLen*
ASSUME $(QLen \in Nat) \land (QLen > 0)$
$M \triangleq$ INSTANCE *InternalMemory*

$Init \triangleq$ The initial predicate
$\quad \land\ M!IInit$ — *mem*, *buf*, and *ctl* are initialized as in the internal memory spec.
$\quad \land\ cache = [p \in Proc \mapsto [a \in Adr \mapsto NoVal]]$ — All caches are initially empty ($cache[p][a] = NoVal$ for all p, a).
$\quad \land\ memQ = \langle\,\rangle$ — The queue *memQ* is initially empty.

$TypeInvariant \triangleq$ The type invariant.
$\quad \land\ mem \in [Adr \to Val]$
$\quad \land\ ctl \in [Proc \to \{\text{“rdy”}, \text{“busy”}, \text{“waiting”}, \text{“done”}\}]$
$\quad \land\ buf \in [Proc \to MReq \cup Val \cup \{NoVal\}]$
$\quad \land\ cache \in [Proc \to [Adr \to Val \cup \{NoVal\}]]$
$\quad \land\ memQ \in Seq(Proc \times MReq)$ — *memQ* is a sequence of ⟨proc., request⟩ pairs.

$Coherence \triangleq$ Asserts that if two processors' caches both have copies of an address, then those copies have equal values.
$\quad \forall p, q \in Proc,\ a \in Adr :$
$\qquad (NoVal \notin \{cache[p][a],\ cache[q][a]\}) \Rightarrow (cache[p][a] = cache[q][a])$

$vmem \triangleq$ The value *mem* will have after all the writes in *memQ* are performed.
$\quad$ LET $f[i \in 0 \,..\, Len(memQ)] \triangleq$ — The value *mem* will have after the first i writes in *memQ* are performed.
$\qquad$ IF $i = 0$ THEN mem
$\qquad\qquad$ ELSE IF $memQ[i][2].op = \text{“Rd”}$
$\qquad\qquad\qquad$ THEN $f[i-1]$
$\qquad\qquad\qquad$ ELSE $[f[i-1]$ EXCEPT $![memQ[i][2].adr] = memQ[i][2].val]$
$\quad$ IN $f[Len(memQ)]$

$Req(p) \triangleq$ Processor p issues a request.
$\quad M!Req(p) \land$ UNCHANGED $\langle cache, memQ\rangle$

$Rsp(p) \triangleq$ The system issues a response to processor p.
$\quad M!Rsp(p) \land$ UNCHANGED $\langle cache, memQ\rangle$

$RdMiss(p) \triangleq$ Enqueue a request to write value from memory to p's cache.
$\quad \land\ (ctl[p] = \text{“busy”}) \land (buf[p].op = \text{“Rd”})$ — Enabled on a read request when
$\quad \land\ cache[p][buf[p].adr] = NoVal$ — the address is not in p's cache
$\quad \land\ Len(memQ) < QLen$ — and *memQ* is not full.
$\quad \land\ memQ' = Append(memQ, \langle p, buf[p]\rangle)$ — Append ⟨p, request⟩ to *memQ*.
$\quad \land\ ctl' = [ctl$ EXCEPT $![p] = \text{“waiting”}]$ — Set $ctl[p]$ to “waiting”.
$\quad \land$ UNCHANGED $\langle memInt, mem, buf, cache\rangle$

$DoRd(p) \triangleq$ Perform a read by p of a value in its cache.
$\quad \land\ (ctl[p] \in \{\text{“busy”}, \text{“waiting”}\}) \land (buf[p].op = \text{“Rd”})$ — Enabled if a read
$\quad \land\ cache[p][buf[p].adr] \neq NoVal$ — request is pending and
$\quad \land\ buf' = [buf$ EXCEPT $![p] = cache[p][buf[p].adr]]$ — address is in cache.
$\quad \land\ ctl' = [ctl$ EXCEPT $![p] = \text{“done”}]$ — Get result from cache.
$\quad \land$ UNCHANGED $\langle memInt, mem, cache, memQ\rangle$ — Set $ctl[p]$ to “done”.

Figure 10: The write-through cache specification (beginning).

$DoWr(p) \triangleq$ Write to p's cache, update other caches, and enqueue memory update.

$$
\begin{array}{ll}
\text{LET } r \triangleq buf[p] & \text{Processor } p\text{'s request.} \\
\text{IN } \land (ctl[p] = \text{"busy"}) \land (r.op = \text{"Wr"}) & \text{Enabled if write request pending} \\
\quad \land Len(memQ) < QLen & \text{and } memQ \text{ is not full.} \\
\quad \land cache' = & \text{Update } p\text{'s cache and any other cache that has a copy.} \\
\qquad [q \in Proc \mapsto \text{IF } (p = q) \lor (cache[q][r.adr] \neq NoVal) & \\
\qquad\qquad \text{THEN } [cache[q] \text{ EXCEPT } ![r.adr] = r.val] & \\
\qquad\qquad \text{ELSE } cache[q]] & \\
\quad \land memQ' = Append(memQ, \langle p, r \rangle) & \text{Enqueue write at tail of } memQ. \\
\quad \land buf' = [buf \text{ EXCEPT } ![p] = NoVal] & \text{Generate response.} \\
\quad \land ctl' = [ctl \text{ EXCEPT } ![p] = \text{"done"}] & \text{Set } ctl \text{ to indicate request is done.} \\
\quad \land \text{UNCHANGED } \langle memInt, mem \rangle &
\end{array}
$$

$MemQWr \triangleq$ Perform write at head of $memQ$ to memory.

$$
\begin{array}{ll}
\text{LET } r \triangleq Head(memQ)[2] & \text{The request at the head of } memQ. \\
\text{IN } \land (memQ \neq \langle \rangle) \land (r.op = \text{"Wr"}) & \text{Enabled if } Head(memQ) \text{ a write.} \\
\quad \land mem' = [mem \text{ EXCEPT } ![r.adr] = r.val] & \text{Perform the write to memory.} \\
\quad \land memQ' = Tail(memQ) & \text{Remove the write from } memQ. \\
\quad \land \text{UNCHANGED } \langle memInt, mem, buf, ctl \rangle &
\end{array}
$$

$MemQRd \triangleq$ Perform an enqueued read to memory.

$$
\begin{array}{ll}
\text{LET } p \triangleq Head(memQ)[1] & \text{The requesting processor.} \\
\quad\;\; r \triangleq Head(memQ)[2] & \text{The request at the head of } memQ. \\
\text{IN } \land (memQ \neq \langle \rangle) \land (r.op = \text{"Rd"}) & \text{Enabled if } Head(memQ) \text{ is a read.} \\
\quad \land memQ' = Tail(memQ) & \text{Remove the head of } memQ. \\
\quad \land cache' = & \text{Put value from memory or } memQ \\
\qquad [cache \text{ EXCEPT } ![p][r.adr] = vmem[r.adr]] & \text{in } p\text{'s cache.} \\
\quad \land \text{UNCHANGED } \langle memInt, mem, buf, ctl \rangle &
\end{array}
$$

$Evict(p, a) \triangleq$ Remove address a from p's cache.

$$
\begin{array}{ll}
\land (ctl[p] = \text{"waiting"}) \Rightarrow (buf[p].adr \neq a) & \text{Can't evict } a \text{ if it was just read} \\
\land cache' = [cache \text{ EXCEPT } ![p][a] = NoVal] & \text{into cache from memory.} \\
\land \text{UNCHANGED } \langle memInt, mem, buf, ctl, memQ \rangle &
\end{array}
$$

$$
\begin{array}{ll}
Next \triangleq & \lor \exists p \in Proc : \lor Req(p) \lor Rsp(p) \\
& \qquad\qquad\qquad \lor RdMiss(p) \lor DoRd(p) \lor DoWr(p) \\
& \qquad\qquad\qquad \lor \exists a \in Adr : Evict(p, a) \\
& \lor MemQWr \lor MemQRd
\end{array}
$$

$$Spec \triangleq Init \land \Box[Next]_{\langle memInt, mem, buf, ctl, cache, memQ \rangle}$$

THEOREM $Spec \Rightarrow \Box(TypeInvariant \land Coherence)$

$LM \triangleq$ INSTANCE $Memory$ — The memory spec. with internal variables hidden.

THEOREM $Spec \Rightarrow LM!Spec$ — Formula $Spec$ implements the memory spec.

Figure 11: The write-through cache specification (end).

The $DoRd(p)$ action represents the performing of the read from p's cache. If $ctl[p] =$ "busy", then the address was originally in the cache. If $ctl[p] =$ "waiting", then the address was just read into the cache from memory.

The $DoWr(p)$ action writes the value to p's cache and updates the value in any other caches that have copies. It also enqueues a write request in $memQ$. In an implementation, the request is put on the bus, which transmits it to the other caches and to the $memQ$ queue. In our high-level view of the system, we represent all this as a single step.

In the definition of $DoWr$, the LET clause defines r to equal $buf[p]$ within the IN clause. Observe that the definition of r contains the parameter p of the definition of $DoWr$. Hence, we could not move the definition of r outside the definition of $DoWr$.

A definition in a LET is just like an ordinary definition in a module; in particular, it can have parameters. These local definitions can be used to shorten an expression by replacing common subexpressions with an operator. In the definition of $DoWr$, I replaced five instances of $buf[p]$ by the single symbol r. This was a silly thing to do, because it makes almost no difference in the length of the definition and it requires the reader to remember the definition of the new symbol r. But using a LET to eliminate common subexpressions can often greatly shorten and simplify an expression.

A LET can also be used to make an expression easier to read, even if the operators it defines appear only once in the IN expression. We write a specification with a sequence of definitions, instead of just defining a single monolithic formula, because a formula is easier to understand when presented in smaller chunks. The LET construct allows the process of splitting a formula into smaller parts to be done hierarchically. A LET can appear as a subexpression of an IN expression. Nested LETs are common in large, complicated specifications.

Returning to module *WriteThroughCache*, we next come to the two actions $MemQWr$ and $MemQRd$. They represent the processing of the request at the head of the $memQ$ queue—$MemQWr$ for a write request, and $MemQRd$ for a read request. These actions also use a LET to make local definitions. Here, the definitions of p and r could be moved before the definition of $MemQWr$. In fact, we could save space by replacing the two local definitions of r with one global (within the module) definition. However, making the definition of r global in this way would be somewhat distracting, since r is used only in the definitions of $MemQWr$ and $MemQRd$. It might be better instead to combine these two actions into one. Whether you put a definition into a LET or make it more global should depend on what makes the specification easier to read. Writing specifications is a craft whose mastery requires talent and hard work.

The $Evict(p, a)$ action represents the operation of removing address a from processor p's cache. As explained above, we allow an address to be evicted at any time—unless the address was just written to satisfy a pending read request, which is the case iff $ctl[p] =$ "waiting" and $buf[p].adr = a$. Note the use of the "double subscript" in the EXCEPT expression of the action's second conjunct. This conjunct "assigns $NoVal$ to $cache[p][a]$". If address a is not in p's cache, then $cache[p][a]$ already equals $NoVal$ and an $Evict(p, a)$ step is a stuttering step.

The definitions of the next-state action $Next$ and of the complete specification $Spec$ are straightforward. The module closes with two theorems that are discussed below.

6.7 Invariance

Module *WriteThroughCache* contains the theorem

$$\text{THEOREM } Spec \Rightarrow \Box(TypeInvariant \land Coherence)$$

which asserts that $TypeInvariant \land Coherence$ is an invariant of $Spec$. A state predicate $P \land Q$ is always true iff both P and Q are always true, so $\Box(P \land Q)$ is equivalent to $\Box P \land \Box Q$. This implies that the theorem above is equivalent to the two theorems:

$$\text{THEOREM } Spec \Rightarrow \Box TypeInvariant$$
$$\text{THEOREM } Spec \Rightarrow \Box Coherence$$

The first theorem is the usual type-invariance assertion. The second, which asserts that *Coherence* is an invariant of *Spec*, expresses an important property of the algorithm.

Although *TypeInvariant* and *Coherence* are both invariants of the temporal formula *Spec*, they differ in a fundamental way. If s is any state satisfying *TypeInvariant*, then any state t such that $s \rightarrow t$ is a *Next* step also satisfies *TypeInvariant*. This property is expressed by:

$$\text{THEOREM } \mathit{TypeInvariant} \land \mathit{Next} \Rightarrow \mathit{TypeInvariant}'$$

(Recall that *TypeInvariant*$'$ is the formula obtained by priming all the variables in formula *TypeInvariant*.) In general, when $P \land N \Rightarrow P'$ holds, we say that predicate P is an invariant of action N.[13] Predicate *TypeInvariant* is an invariant of *Spec* because it is an invariant of *Next* and it is implied by the initial predicate *Init*.

Predicate *Coherence* is not an invariant of the next-state action *Next*. Suppose s is a state in which

- $\mathit{cache}[p1][a] = 1$
- $\mathit{cache}[q][b] = \mathit{NoVal}$, for all $\langle q, b \rangle$ different from $\langle p1, a \rangle$
- $\mathit{mem}[a] = 2$
- *memQ* contains the single element $\langle p2, [op \mapsto \text{“Rd”}, adr \mapsto a] \rangle$

for two different processors $p1$ and $p2$ and some address a. Then *Coherence* is true in state s. Let t be the state obtained from s by taking a *MemQRd* step. In state t, we have $\mathit{cache}[p2][a] = 2$ and $\mathit{cache}[p1][a] = 1$, so *Coherence* is false. Hence *Coherence* is not an invariant of the next-state action.

Coherence is an invariant of formula *Spec* because states like s cannot occur in a behavior satisfying *Spec*. Proving its invariance is not so easy. We must find a predicate *Inv* that is an invariant of *Next* such that *Inv* implies *Coherence* and is implied by the initial predicate *Init*.

Important properties of a specification can often be expressed as invariants. Proving that a state predicate P is an invariant of a specification means proving a formula of the form

$$\mathit{Init} \land \Box[\mathit{Next}]_v \Rightarrow \Box P$$

This is done by finding an appropriate state predicate *Inv* and proving

$$\mathit{Init} \Rightarrow \mathit{Inv}, \qquad \mathit{Inv} \land [\mathit{Next}]_v \Rightarrow \mathit{Inv}', \qquad \mathit{Inv} \Rightarrow P$$

Since our subject is specification, not proof, I won't discuss how to find *Inv*.

6.8 Proving Implementation

Module *WriteThroughCache* ends with the theorem

$$\text{THEOREM } \mathit{Spec} \Rightarrow \mathit{LM}!\mathit{Spec}$$

where *LM*!*Spec* is formula *Spec* of module *Memory*. By definition of this formula, we can restate the theorem as

$$\text{THEOREM } \mathit{Spec} \;\Rightarrow\; \boldsymbol{\exists}\, \mathit{mem}, \mathit{ctl}, \mathit{buf} \;:\; \mathit{LM}!\mathit{Inner}(\mathit{mem}, \mathit{ctl}, \mathit{buf})!\mathit{ISpec}$$

where *LM*!*Inner*(*mem*, *ctl*, *buf*)!*ISpec* is formula *ISpec* of the *InnerMemory* module. The rules of logic tell us that to prove such a theorem, we must find “witnesses” for the quantified

[13] An invariant of a specification S that is also an invariant of its next-state action is sometimes called an *inductive* invariant of S.

variables *mem*, *ctl*, and *buf*. These witness are state functions (ordinary expressions with no primes), which I'll call *omem*, *octl*, and *obuf*, that satisfy:

(5) $\quad Spec \;\Rightarrow\; LM!Inner(omem, octl, obuf)!ISpec$

The tuple $\langle omem, octl, obuf \rangle$ of witness functions is called a *refinement mapping*, and we describe (5) as the assertion that *Spec* implements formula *ISpec* (of module *InnerMemory*) under this refinement mapping. Intuitively, this means *Spec* implies that the value of the tuple of state functions $\langle memInt, omem, octl, obuf \rangle$ changes the way *ISpec* asserts that the tuple of variables $\langle memInt, mem, ctl, buf \rangle$ should change.

I will now briefly describe how we prove (5); for details, see [8]. Let me first introduce a bit of non-TLA$^+$ notation. For any formula F of module *InnerMemory*, let $\overline{F}$ equal $LM!Inner(omem, octl, obuf)!F$, which is formula F with *omem*, *octl*, and *obuf* substituted for *mem*, *ctl*, and *buf*. In particular, $\overline{mem}$, $\overline{ctl}$, and $\overline{buf}$ equal *omem*, *octl*, and *obuf*, respectively.

Replacing *Spec* and *ISpec* by their definitions transforms (5) to

$$\begin{aligned} &Init \wedge \Box[Next]_{\langle memInt,\, mem,\, buf,\, ctl,\, cache,\, memQ \rangle} \\ &\quad \Rightarrow \;\; \overline{IInit} \wedge \Box[\overline{INext}]_{\langle memInt,\, \overline{mem},\, \overline{ctl},\, \overline{buf} \rangle} \end{aligned}$$

This is proved by finding an invariant *Inv* of *Spec* such that

$$\begin{aligned} &\wedge\; Init \Rightarrow \overline{IInit} \\ &\wedge\; Inv \wedge Next \;\Rightarrow\; \vee\; \overline{INext} \\ &\qquad\qquad\qquad\qquad\;\; \vee\; \textsc{unchanged}\; \langle memInt, \overline{mem}, \overline{ctl}, \overline{buf} \rangle \end{aligned}$$

The second conjunct is called *step simulation.* It asserts that a *Next* step starting in a state satisfying the invariant *Inv* is either an $\overline{INext}$ step—a step that changes the 4-tuple $\langle memInt, omem, octl, obuf \rangle$ the way an *INext* step changes $\langle memInt, mem, ctl, buf \rangle$—or else it leaves that 4-tuple unchanged.

The mathematics of an implementation proof is simple, so the proof is straightforward—in theory. For specifications of real systems, such proofs can be quite difficult. Going from the theory to practice requires turning the mathematics of proofs into an engineering discipline—a subject that deserves a book to itself. However, when writing specifications, it helps to understand refinement mappings and step simulation.

We now return to the question posed in Section 4.2: what is the relation between the specifications of the asynchronous interface in modules *AsynchInterface* and *Channel*? Recall that module *AsynchInterface* describes the interface in terms of the three variables *val*, *rdy*, and *ack*, while module *Channel* describes it with a single variable *chan* whose value is a record with *val*, *rdy*, and *ack* components. In what sense are those two specifications of the interface equivalent?

One answer that now suggests itself is that each of the specifications should implement the other under a refinement mapping. We expect formula *Spec* of module *Channel* to imply the formula obtained from *Spec* of module *AsynchInterface* by substituting for its variables *val*, *rdy*, and *ack* the *val*, *rdy*, and *ack* components of the variable *chan* of module *Channel*. This assertion is expressed precisely by the theorem in the following module.

———————— MODULE *ChannelImplAsynch* ————————

EXTENDS *Channel*

$AInt(val, rdy, ack) \;\triangleq\;$ INSTANCE *AsynchInterface*

THEOREM $Spec \Rightarrow AInt(chan.val, chan.rdy, chan.ack)!Spec$

————————————————————————

In this example, the refinement mapping substitutes $\langle chan.val,\ chan.rdy,\ chan.ack \rangle$ for the tuple $\langle val,\ rdy,\ ack \rangle$ of variables in the formula *Spec* of module *AsynchInterface*.

Similarly, formula *Spec* of module *AsynchInterface* implies formula *Spec* of module *Channel* with *chan* replaced by the record-valued expression:

$$[val \mapsto val,\ rdy \mapsto rdy,\ ack \mapsto ack]$$

(The first *val* in $val \mapsto val$ is the field name in the record constructor, while the second *val* is the variable of module *AsynchInterface*.)

7 Some More Math

Our mathematics is built on a small, simple collection of concepts. You've already seen most of what's needed to describe almost any kind of mathematics. All you lack are a handful of operators on sets that are described below in Section 7.1. After learning about them, you will be able to define all the data structures and operations that occur in specifications.

While our mathematics is simple, its foundations are nonobvious—for example, the meanings of recursive function definitions and the CHOOSE operator are subtle. This section discusses some of those foundations. Understanding them will help you use mathematics more effectively.

7.1 Sets

The simple operations on sets described in Section 2.2 are all you'll need for writing most system specifications. However, you may occasionally have to use more sophisticated operators—especially if you need to define data structures beyond tuples, records, and simple functions.

Two powerful operators of set theory are the unary operators UNION and SUBSET, defined as follows.[14]

UNION S The union of the elements of S. In other words, a value e is an element of UNION S iff it is an element of an element of S. For example:

$$\text{UNION}\ \{\{1,2\},\{2,3\},\{3,4\}\} \ = \ \{1,2,3,4\}$$

SUBSET S The set of all subsets of S. In other words, $T \in \text{SUBSET}\ S$ iff $T \subseteq S$. For example:

$$\text{SUBSET}\ \{1,2\} \ = \ \{\{\},\{1\},\{2\},\{1,2\}\}$$

Mathematicians often describe a set as "the set of all ... such that ...". TLA$^+$ has two constructs that formalize such a description:

$\{x \in S : P\}$ The subset of S consisting of all elements x satisfying property P. For example, the set of odd natural numbers can be written $\{n \in Nat : n \,\%\, 2 = 1\}$. (The modulus operator % is described in Section 3.5.) The identifier x is bound in P; it may not occur in S.

$\{e : x \in S\}$ The set of elements of the form e, for all x in the set S. For example, $\{2 * n + 1 : n \in Nat\}$ is the set of all odd natural numbers. The identifier x is bound in e; it may not occur in S.

[14] Mathematicians usually write $\bigcup S$ instead of UNION S. They often call SUBSET S the *powerset* of S and write it $\mathcal{P}(S)$ or 2^S.

The construct $\{e : x \in S\}$ has the same generalizations as $\exists\, x \in S : F$. For example,

$$\{e \;:\; x \in S,\; y \in T\}$$

is the set of all elements of the form e, for x in S and y in T. In the construct $\{x \in S : P\}$, we can let x be a tuple. For example, $\{\langle y, z \rangle \in S : P\}$ is the set of all pairs $\langle y, z \rangle$ in the set S that satisfy P.

All the set operators we've seen so far are built-in operators of TLA$^+$. There is also a *FiniteSets* module that defines two operators:

$\mathit{Cardinality}(S)$ The number of elements in set S, if S is a finite set.

$\mathit{IsFiniteSet}(S)$ True iff S is a finite set.

Careless reasoning about sets can lead to problems. The classic example of this is Russell's paradox:

> Let $\mathcal{R}$ be the set of all sets S such that $S \notin S$. The definition of $\mathcal{R}$ implies that $\mathcal{R}$ is an element of $\mathcal{R}$ iff $\mathcal{R}$ is not an element of $\mathcal{R}$ is true.

Obviously, $\mathcal{R}$ can't both be and not be an element of $\mathcal{R}$. The source of the paradox is that $\mathcal{R}$ isn't a set. There's no way to write it in TLA$^+$. Intuitively, $\mathcal{R}$ is too big to be a set. A collection $\mathcal{C}$ is too big to be a set if it is as big as the collection of all sets—meaning that we can assign to every set a different element of $\mathcal{C}$. More precisely, $\mathcal{C}$ is too big to be a set if we can define an operator *SMap* such that:

- $\mathit{SMap}(S)$ is in $\mathcal{C}$, for any set S.
- If S and T are two different sets, then $\mathit{SMap}(S) \neq \mathit{SMap}(T)$.

For example, the collection of all sequences of length 2 is too big to be a set; we can define the operator *SMap* by

$$\mathit{SMap}(S) \;\triangleq\; \langle 1, S \rangle$$

7.2 Silliness

Most modern programming languages introduce some form of type checking to prevent you from writing silly expressions like 3/"abc". TLA$^+$ is based on the usual formalization of mathematics, which doesn't have types. In an untyped formalism, every syntactically well-formed expression has a meaning—even a silly expression like 3/"abc". Mathematically, the expression 3/"abc" is no sillier than the expression $3/0$, and mathematicians implicitly write that silly expression all the time. For example, consider the valid formula

$$\forall\, x \in \mathit{Real} \;:\; (x \neq 0) \Rightarrow (x * (3/x) = 3)$$

where *Real* is the set of all real numbers. This asserts that $(x \neq 0) \Rightarrow (x * (3/x) = 3)$ is true for all real numbers x. Substituting 0 for x yields the valid formula $(0 \neq 0) \Rightarrow (0 * (3/0) = 3)$ that contains the silly expression $3/0$. It's valid because $0 \neq 0$ equals FALSE, and FALSE $\Rightarrow P$ is true for any formula P.

A correct formula can contain silly expressions. However, the validity of a correct formula cannot depend on the meaning of a silly expression. If an expression is silly, then its meaning is probably unspecified. The definitions of 0, 3, /, and $*$ (which are in the standard module *Reals*) don't specify the value of $0 * (3/0)$, so there's no way of knowing whether that value equals 3.

No sensible syntactic rules can prevent you from writing 3/0 without also preventing you from writing perfectly reasonable expressions. The typing rules of programming languages introduce complexity and limitations on what you can write that don't exist in ordinary mathematics. In a well-designed programming language, the costs of types are balanced by benefits: types allow a compiler to produce more efficient code, and type checking catches errors. For programming languages, the benefits seem to outweigh the costs. For writing specifications, I have found that the costs outweigh the benefits.

If you're used to the constraints of programming languages, it may be a while before you start taking advantage of the freedom afforded by mathematics. At first, you won't think of defining anything like the operator R defined in Section 6.2, which couldn't be written in a typed programming language.

Mathematically, what a specification means is equivalent to what the true assertions about it are. Assertions are expressed with Boolean operators, so we need to pay specific attention to silly expressions written with those operators—expressions like $5 \land$ "abc". What does such an expression mean? There are several ways of answering this question, but there are two principle ones that I call the *conservative* and *liberal* views.

In the conservative view, the value of an expression like $2 \land$ "abc" is completely unspecified. It could equal $\sqrt{2}$. It need not equal "abc" $\land 2$. Hence, the ordinary laws of logic, such as the commutativity of $\land$, are valid only for Boolean values.

In the liberal interpretation, the value of $2 \land$ "abc" is a Boolean. It equals either TRUE or FALSE, but we don't know which. However, all the ordinary laws of logic, such as the commutativity of $\land$, are valid. Hence, $2 \land$ "abc" equals "abc" $\land 2$. More precisely, any tautology of propositional or predicate logic, such as

$$(\forall x : p) \;\equiv\; \neg(\exists x : \neg p)$$

is valid, even for nonBoolean values.[15]

The semantics of TLA$^+$ assert that the rules of the conservative view are valid. The liberal view is neither required nor forbidden. You should write specifications that make sense under the conservative view. However, you (and the implementer of a tool) are free to use the liberal interpretation if you wish.

7.3 Recursive Function Definitions Revisited

Section 6.5 introduced recursive function definitions. Let's now examine what such definitions mean mathematically. Mathematicians usually define the factorial function *fact* by writing:

$$fact[n] \;=\; \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1], \;\text{ for all } n \in Nat$$

This definition can be justified by proving that it defines a unique function *fact* with domain *Nat*. In other words, *fact* is the unique value satisfying:

(6) $\quad fact = [n \in Nat \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1]]$

The CHOOSE operator, introduced in Section 6.1, allows us to express "the value satisfying property P" as CHOOSE $x : P$. We can therefore define *fact* as follows to be the value satisfying (6):

(7) $\quad fact \;\triangleq\; \text{CHOOSE } fact : fact = [n \in Nat \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1]]$

(Since the symbol *fact* is not yet defined in the expression to the right of the $\triangleq$, we can use it as the bound identifier in the CHOOSE expression.) The TLA$^+$ definition

$$fact[n \in Nat] \;\triangleq\; \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1]$$

[15] Equality ($=$) is not an operator of propositional or predicate logic; this tautology need not be valid for nonBoolean values if $\equiv$ is replaced by $=$.

is simply an abbreviation for (7). In general, $f[x \in S] \triangleq e$ is an abbreviation for:

(8) $f \;\triangleq\; \text{CHOOSE } f : f = [x \in S \mapsto e]$

TLA$^+$ allows you to write silly definitions. For example, you can write

(9) $circ[n \in Nat] \;\triangleq\; \text{CHOOSE } y : y \neq circ[n]$

This appears to define *circ* to be a function such that $circ[n] \neq circ[n]$ for any natural number n. There obviously is no such function, so *circ* can't be defined to equal it. A recursive function definition doesn't necessarily define a function. If there is no f that equals $[x \in S \mapsto e]$, then (8) defines f to be some unspecified value. Thus, the nonsensical definition (9) defines *circ* to be some unknown value.

If we want to reason about a function f defined by $f[x \in S] \triangleq e$, we need to prove that there exists an f that equals $[x \in S \mapsto e]$. The existence of f is obvious if f does not occur in e. If it does, so this is a recursive definition, then there is something to prove. Since I'm not discussing proofs, I won't describe how to prove it. Intuitively, you have to check that, as in the case of the factorial function, the definition uniquely determines the value of $f[x]$ for every x in S.

Recursion is a common programming technique because programs must compute values using a small repertoire of simple elementary operations. It's not used so often in mathematical definitions, where we needn't worry about how to compute the value and can use the powerful operators of logic and set theory. For example, the operators *Head*, *Tail*, and $\circ$ are defined in Section 6.4 without recursion, even though computer scientists usually define them recursively. Still, there are some things that are best defined inductively, using a recursive function definition.

7.4 Functions versus Operators

Consider these definitions, which we've seen before

$$Tail(s) \;\triangleq\; [i \in 1 \,..\, (Len(s) - 1) \mapsto s[i+1]]$$
$$fact[n \in Nat] \;\triangleq\; \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1]$$

They define two very different kinds of objects: *fact* is a function, and *Tail* is an operator. Functions and operators differ in a few basic ways.

Their most obvious difference is that a function like *fact* by itself is a complete expression that denotes a value, but an operator like *Tail* is not. Both $fact[n] \in S$ and $fact \in S$ are syntactically correct expressions. But, while $Tail(n) \in S$ is syntactically correct, $Tail \in S$ is not. It is gibberish—a meaningless string of symbols, like $x+ = 0$.

Their second difference is more profound. The definition of *Tail* defines $Tail(s)$ for all values of s. For example, it defines $Tail(1/2)$ to equal

(10) $[i \in 1 \,..\, (Len(1/2) - 1) \mapsto (1/2)[i+1]]$

We have no idea what this expression means, because we don't know what $Len(1/2)$ or $(1/2)[i+1]$ mean. But, whatever (10) means, it equals $Tail(1/2)$.

The definition of *fact* defines $fact[n]$ only for $n \in Nat$. It tells us nothing about the value of $fact[1/2]$. The expression $fact[1/2]$ is syntactically well-formed, so it denotes some value. But the definition of *fact* tells us nothing about what that value is.

Unlike an operator, a function must have a domain, which is a set. We cannot define a function *Tail* so that $Tail[s]$ is the tail of any nonempty sequence s; the domain of such a function would have to include all nonempty sequences, and the collection of all such sequences is too big to be a set. (The operator *SMap* defined by $SMap(S) \triangleq \langle S \rangle$ maps every set to a different nonempty sequence.) Hence, we can't define *Tail* to be a function.

Unlike a function, an operator cannot be defined recursively. However, we can usually transform an illegal recursive operator definition into a nonrecursive one using a recursive function definition. For example, let's try to define the *Cardinality* operator on the set of finite sets. (Recall that the cardinality of a finite set S is the number of elements in S.) The collection of all finite sets is too big to be a set. (The operator $\mathit{SMap}(S) \triangleq \{S\}$ maps every set S to a different set $\{S\}$ of cardinality 1.) The *Cardinality* operator has a simple intuitive definition:

- $\mathit{Cardinality}(\{\}) = 0$.
- If S is a nonempty finite set, then

  $$\mathit{Cardinality}(S) = 1 + \mathit{Cardinality}(S \setminus \{x\})$$

  where x is an arbitrary element of S. (The set $S \setminus \{x\}$ contains all the elements of S except x.)

Using the CHOOSE operator to describe an arbitrary element of S, we can write this as the more formal-looking, but still illegal, definition:

$$\begin{array}{lcl} \mathit{Cardinality}(S) & \triangleq & \textsc{if}\ \ S = \{\}\ \ \textsc{then}\ \ 0 \\ & & \qquad\qquad\quad\ \textsc{else}\ \ 1 + \mathit{Cardinality}(S \setminus \{\textsc{choose}\ x : x \in S\}) \end{array}$$

This definition is illegal because it's circular—only in a recursive function definition can the symbol being defined appear to the right of the $\triangleq$.

To turn this into a legal definition, observe that, for a given set S, we can define a function CS such that $CS[T]$ equals the cardinality of T for every subset T of S. The definition is

$$\begin{array}{lcl} CS[T \in \textsc{subset}\ S] & \triangleq & \textsc{if}\ \ T = \{\}\ \ \textsc{then}\ \ 0 \\ & & \qquad\qquad\quad\ \textsc{else}\ \ 1 + CS[T \setminus \{\textsc{choose}\ x : x \in T\}] \end{array}$$

Since S is a subset of itself, this defines $CS[S]$ to equal $\mathit{Cardinality}(S)$, if S is a finite set. (We don't know or care what $CS[S]$ equals if S is not finite.) So, we can define the *Cardinality* operator by:

$$\begin{array}{lcl} \mathit{Cardinality}(S) & \triangleq & \textsc{let}\ \ CS[T \in \textsc{subset}\ S]\ \triangleq \\ & & \qquad\quad \textsc{if}\ \ T = \{\}\ \ \textsc{then}\ \ 0 \\ & & \qquad\qquad\qquad\qquad\ \textsc{else}\ \ 1 + CS[T \setminus \{\textsc{choose}\ x : x \in T\}] \\ & & \textsc{in}\quad CS[S] \end{array}$$

Operators also differ from functions in that an operator can take an operator as an argument. For example, we can define an operator *IsPartialOrder* so that $\mathit{IsPartialOrder}(R, S)$ equals true iff the operator R defines an irreflexive partial order[16] on S. The definition is

$$\begin{array}{lcl} \mathit{IsPartialOrder}(R(\_, \_), S) & \triangleq & \land\ \forall x, y, z \in S : R(x, y) \land R(y, z) \Rightarrow R(x, z) \\ & & \land\ \forall x \in S : \neg R(x, x) \end{array}$$

We could also use an infix-operator symbol like $\prec$ instead of R as the parameter of the definition, writing:

$$\begin{array}{lcl} \mathit{IsPartialOrder}(\_ \prec \_, S) & \triangleq & \land\ \forall x, y, z \in S : (x \prec y) \land (y \prec z) \Rightarrow (x \prec z) \\ & & \land\ \forall x \in S : \neg(x \prec x) \end{array}$$

[16] If you don't know what an irreflexive partial order is, read the following definition of *IsPartialOrder* to find out.

The first argument of *IsPartialOrder* is an operator that takes two arguments; its second argument is an expression. Since $>$ is an operator that takes two arguments, the expression $IsPartialOrder(>, Nat)$ is syntactically correct. In fact, it's valid—if $>$ is defined to be the usual operator on numbers. The expression $IsPartialOrder(+, 3)$ is also syntactically correct, but it's silly and we have no idea whether or not it's valid.

The last difference between operators and functions has nothing to do with mathematics and is an idiosyncrasy of TLA$^+$: the language doesn't permit you to define infix functions. So, if we want to define $/$, we have no choice but to make it an operator.

When defining an object V, you may have to decide whether to make V an operator or a function. The differences between operators and functions will often determine the decision. For example, if a variable may have V as its value, then V must be a function. Thus, in the memory specification of Section 6.3, we had to represent the state of the memory by a function rather than an operator, since the variable *mem* couldn't equal an operator. If these differences don't determine whether to use an operator or a function, then it's a matter of taste. I usually prefer operators.

7.5 Using Functions

Consider the following two formulas:

(11) $f' = [i \in Nat \mapsto i + 1]$

(12) $\forall\, i \in Nat : f'[i] = i + 1$

These formulas both imply that $f'[i] = i + 1$ for every natural number i, but they are not equivalent. Formula (11) uniquely determines f', asserting that it's a function with domain Nat. Formula (12) is satisfied by lots of different values of f'—for example, by the function

$$[i \in Real \mapsto \text{IF } i \in Nat \text{ THEN } i + 1 \text{ ELSE } \sqrt{i}]$$

In fact, from (12), we can't even deduce that f' is a function. Formula (11) implies formula (12), but not vice-versa.

When writing specifications, we almost always want to specify the new value of a variable f rather than the new values of $f[i]$ for all i in some set. We therefore usually write (11) rather than (12),

7.6 Choose

The CHOOSE operator[17] was introduced in the memory interface of Section 6.1 in the simple idiom CHOOSE $v : v \notin S$, which is an expression whose value is not an element of S. In Section 7.3 above, we saw that it is a powerful tool that can be used in rather subtle ways.

The most common use for the CHOOSE operator is to "name" a uniquely specified value. For example, one possible definition of division on the set *Real* of real numbers is:

$$r/s \;\triangleq\; \text{CHOOSE } v \in Real : (v * s = r)$$

(The expression CHOOSE $x \in S : e$ means CHOOSE $x : (x \in S) \land e$.) If r is a nonzero real number, then there is no real number v such that $v * 0 = r$. Therefore, $r/0$ has an unspecified value. We don't know what a real number times a string equals, so we cannot say whether or not there is a real number v such that $v *$ "abc" equals r. Hence, we don't know what the value of $r/$"abc" is.

People who do a lot of programming and not much mathematics often think that CHOOSE must be a nondeterministic operator. In mathematics, there is no such thing as a nondeterministic operator or a nondeterministic function. If some expression equals 42 today, then

[17] The CHOOSE operator is known to logicians as Hilbert's ε [10].

it will equal 42 tomorrow, and it will still equal 42 a million years from tomorrow. The specification

$$(x = \text{CHOOSE } n : n \in Nat) \ \wedge\ \Box[x' = \text{CHOOSE } n : n \in Nat]_x$$

allows only a single behavior—one in which x always equals the number $\text{CHOOSE } n : n \in Nat$, some particular, unspecified natural number. It is very different from the specification

$$(x \in Nat) \ \wedge\ \Box[x' \in Nat]_x$$

that allows all behaviors in which x is always a natural number—possibly a different number in each state. This specification is highly nondeterministic, allowing lots of different behaviors.

8 Liveness and Fairness

The specifications we have written so far say what a system must *not* do. The clock must not advance from 11 to 9; the receiver must not receive a message if the FIFO is empty. They don't require that the system ever do anything. The clock need never tick; the sender need never send any messages. Our specifications have described what are called *safety properties.* If a safety property is violated, it is violated at some particular point in the behavior—by a step that advances the clock from 11 to 9, or that reads the wrong value from memory.

We now learn how to specify that something *does* happen: the clock keeps ticking; a value is eventually read from memory. We specify *liveness properties*, which cannot be violated at any particular instant. Only by examining an entire infinite behavior can we tell that the clock has stopped ticking, or that a message is never sent.

8.1 Temporal Formulas

To specify liveness properties we must learn to express them as temporal formulas. We now take a more rigorous look at what a temporal formula means.

Recall that a state assigns a value to every variable, and a behavior is an infinite sequence of states. A temporal formula is true or false of a behavior. Let $\sigma \models F$ be the truth value of the formula F for the behavior σ, so σ satisfies F iff $\sigma \models F$ equals TRUE. To define the meaning of a temporal formula F, we have to explain how to determine the value of $\sigma \models F$ for any behavior σ. For now, we consider only temporal formulas that don't contain the temporal existential quantifier $\boldsymbol{\exists}$.

It's easy to define the meaning of a Boolean combination of temporal formulas in terms of the meanings of those formulas. The formula $F \wedge G$ is true of a behavior σ iff both F and G are true of σ, and $\neg F$ is true of σ iff F is false for σ:

$$\sigma \models (F \wedge G) \ \triangleq\ (\sigma \models F) \wedge (\sigma \models G) \qquad\qquad \sigma \models \neg F \ \triangleq\ \neg(\sigma \models F)$$

This defines the meanings of $\wedge$ and $\neg$ as operators on temporal formulas. The meanings of the other Boolean operators are similarly defined. In the same way, we can define the ordinary predicate-logic quantifiers $\forall$ and $\exists$ as operators on temporal formulas—for example:

$$\sigma \models (\exists\, r : F) \ \triangleq\ \exists\, r : (\sigma \models F)$$

We will discuss quantifiers in Section 8.6. For now, we ignore quantification in temporal formulas.

All the temporal formulas not containing $\boldsymbol{\exists}$ that we've seen have been Boolean combinations of the following three simple kinds of formulas. (Recall the definitions of state function and state predicate in Section 4.1.)

- A state predicate. It is interpreted as a temporal formula that is true of a behavior iff it is true in the first state of the behavior.

- A formula $\Box P$, where P is a state predicate. It is true of a behavior iff P is true in every state of the behavior.

- A formula $\Box[N]_v$, where N is an action and v is a state function. It is true of a behavior iff every successive pair of steps in the behavior is a $[N]_v$ step.

Since a state predicate is an action that contains no primed variables, we can both combine and generalize these three kinds of temporal formulas into the two kinds of formulas A and $\Box A$, where A is an action.

Generalizing from state functions, we interpret an arbitrary action A as a temporal formula by defining $\sigma \models A$ to be true iff the first two states of σ are an A step. For any behavior σ, let $\sigma_0, \sigma_1, \ldots$ be the sequence of states that make up σ. Then the meaning of an action A as a temporal formula is defined by letting $\sigma \models A$ be true iff the pair $\langle \sigma_0, \sigma_1 \rangle$ of states is an A step. We define $\sigma \models \Box A$ to be true iff, for all $n \in \mathit{Nat}$, the pair $\langle \sigma_n, \sigma_{n+1} \rangle$ of states is an A step. We now generalize this to the definition of $\sigma \models \Box F$ for an arbitrary temporal formula F.

For any behavior σ and natural number n, let σ^{+n} be the suffix of σ obtained by deleting its first n states:

$$\sigma^{+n} \triangleq \sigma_n, \sigma_{n+1}, \sigma_{n+2}, \ldots$$

The successive pair of states $\langle \sigma_n, \sigma_{n+1} \rangle$ of σ is the first pair of states of σ^{+n}, and $\langle \sigma_n, \sigma_{n+1} \rangle$ is an A step iff σ^{+n} satisfies A. In other words:

$$(\sigma \models \Box A) \equiv \forall n \in \mathit{Nat} : \sigma^{+n} \models A$$

So, we can generalize the definition of $\sigma \models \Box A$ to

$$\sigma \models \Box F \triangleq \forall n \in \mathit{Nat} : \sigma^{+n} \models F$$

for any temporal formula F. In other words, σ satisfies $\Box F$ iff every suffix σ^{+n} of σ satisfies F. This defines the temporal operator $\Box$.

We have now defined the meaning of any temporal formula built from actions (including state predicates), Boolean operators, and the $\Box$ operator. For example:

$$\begin{array}{ll}
\sigma \models \Box((x = 1) \Rightarrow \Box(y > 0)) & \\
\quad \equiv \forall n \in \mathit{Nat} : \sigma^{+n} \models ((x = 1) \Rightarrow \Box(y > 0)) & \text{By the meaning of } \Box. \\
\quad \equiv \forall n \in \mathit{Nat} : (\sigma^{+n} \models (x = 1)) \Rightarrow (\sigma^{+n} \models \Box(y > 0)) & \text{By the meaning of } \Rightarrow. \\
\quad \equiv \forall n \in \mathit{Nat} : (\sigma^{+n} \models (x = 1)) \Rightarrow & \\
\qquad\qquad (\forall m \in \mathit{Nat} : (\sigma^{+n})^{+m} \models (y > 0)) & \text{By the meaning of } \Box.
\end{array}$$

Thus, $\sigma \models \Box((x = 1) \Rightarrow \Box(y > 0))$ is true iff, for all $n \in \mathit{Nat}$, if $x = 1$ is true in state σ_n, then $y > 0$ is true in all states σ_{n+m} with $m \geq 0$.

We saw in Section 3.2 that a specification should allow stuttering steps—ones that leave unchanged all the variables appearing in the formula. A stuttering step represents a change only to some part of the system not described by the formula; adding it to the behavior should not affect the truth of the formula. We say that a formula F is *invariant under stuttering*[18] if adding a stuttering step to a behavior σ does not affect whether F is true of σ. (This implies that removing a stuttering step from σ also does not affect the truth of $\sigma \models F$.) A sensible formula should be invariant under stuttering. There's no point writing formulas that aren't sensible, so TLA$^+$ allows you to write only temporal formulas that are invariant under stuttering.

---

[18]This is a completely new sense of the word *invariant*; it has nothing to do with the concept of invariance discussed already.

An arbitrary action, viewed as a temporal formula, is not invariant under stuttering. The action $x' = x+1$ is true for a behavior in which x is incremented by 1 in the first step; adding an initial stuttering step makes it false. A state predicate is invariant under stuttering, since its truth depends only on the first state of a behavior, and adding a stuttering step doesn't change the first state. The formula $\Box[N]_v$ is also invariant under stuttering, for any action N and state function v. It's not hard to see that invariance under stuttering is preserved by $\Box$ and by the Boolean operators. So, state predicates, formulas of the form $\Box[N]_v$, and all formulas obtainable from them by applying $\Box$ and Boolean operators are invariant under stuttering. For now, let's take these to be our temporal formulas. (Later, we'll add quantification.)

To understand temporal formulas intuitively, think of σ_n as the state of the universe at time instant n during the behavior σ.[19] For any state predicate P, the expression $\sigma^{+n} \models P$ asserts that P is true at time instant n. Thus, $\Box((x = 1) \Rightarrow \Box(y > 0))$ asserts that any time $x = 1$ is true, $y > 0$ is true from then on. For an arbitrary temporal formula F, we also interpret $\sigma^{+n} \models F$ as the assertion that F is true at time instant n. The formula $\Box F$ then asserts that F is true at all times. We can therefore read $\Box$ as *always* or *henceforth* or *from then on.*

We now examine five especially important classes of formulas that are constructed from arbitrary temporal formulas F and G. We introduce new operators for expressing the first three.

$\Diamond F$ is defined to equal $\neg\Box\neg F$. It asserts that F is not always false, which means that F is true at some time:

$$\begin{array}{rcll}
\sigma \models \Diamond F & \equiv & \sigma \models \neg\Box\neg F & \text{By definition of } \Diamond. \\
& \equiv & \neg(\sigma \models \Box\neg F) & \text{By the meaning of } \neg. \\
& \equiv & \neg(\forall\, n \in Nat : \sigma^{+n} \models \neg F) & \text{By the meaning of } \Box. \\
& \equiv & \neg(\forall\, n \in Nat : \neg(\sigma^{+n} \models F)) & \text{By the meaning of } \neg. \\
& \equiv & \exists\, n \in Nat : \sigma^{+n} \models F & \text{Because } \neg\forall\neg \text{ is equivalent to } \exists.
\end{array}$$

We usually read $\Diamond$ as *eventually*, taking eventually to include now.

$F \leadsto G$ is defined to equal $\Box(F \Rightarrow \Diamond G)$. The same kind of calculation we've done above shows

$$\sigma \models (F \leadsto G) \;\equiv\; \forall\, n \in Nat : (\sigma^{+n} \models F) \Rightarrow (\exists\, m \in Nat : (\sigma^{+(n+m)} \models G))$$

The formula $F \leadsto G$ asserts that whenever F is true, G is eventually true—that is, true then or at some later time. We read $\leadsto$ as *leads to.*

$\Diamond\langle A\rangle_v$ is defined to equal $\neg\Box[\neg A]_v$, where A is an action and v a state function. It asserts that not every step is a $(\neg A) \vee (v' = v)$ step, so some step is a $\neg((\neg A) \vee (v' = v))$ step. But $\neg((\neg A) \vee (v' = v))$ is equivalent to $A \wedge (v' \neq v)$, so $\Diamond\langle A\rangle_v$ asserts that some step is an $A \wedge (v' \neq v)$ step. We define $\langle A\rangle_v$ to equal $A \wedge (v' \neq v)$, so $\Diamond\langle A\rangle_v$ asserts that eventually an $\langle A\rangle_v$ step occurs. We think of $\Diamond\langle A\rangle_v$ as the formula obtained by applying the operator $\Diamond$ to $\langle A\rangle_v$, although technically it's not because $\langle A\rangle_v$ isn't a temporal formula.

$\Box\Diamond F$ asserts that at all times, F is true then or at some later time. For time instant 0, this implies that F is true at some time instant $n_0 \geq 0$. For time instant $n_0 + 1$, it implies that F is true at some time instant $n_1 \geq n_0 + 1$. For time instant $n_1 + 1$, it implies

[19] It is because we think of σ_n as the state at time n, and because we usually measure time starting from 0, that I start numbering the states of a behavior with 0 rather than 1.

that F is true at some time instant $n_2 \geq n_1 + 1$. Continuing the process, we see that F is true at an infinite sequence of time instants $n_0, n_1, n_2, \ldots$. So, $\Box\Diamond F$ asserts that F is *infinitely often* true. In particular, $\Box\Diamond\langle A\rangle_v$ asserts that infinitely many $\langle A\rangle_v$ steps occur.

$\Diamond\Box F$ asserts that eventually (at some time), F becomes true and remains true thereafter. In other words, $\Diamond\Box F$ asserts that F is *eventually always* true. In particular, $\Diamond\Box[N]_v$ asserts that eventually, every step is a $[N]_v$ step.

The operators $\Box$ and $\Diamond$ have higher precedence (bind more tightly) than the Boolean operators, so $\Diamond F \vee \Box G$ means $(\Diamond F) \vee (\Box G)$. The operator $\leadsto$ has the same precedence as $\Rightarrow$.

8.2 *Weak Fairness*

Using the temporal operators $\Box$ and $\Diamond$, it's easy to specify liveness properties. For example, consider the hour-clock specification of module *HourClock* in Figure 1. We can require that the clock never stops by asserting that there must be infinitely many *HCnxt* steps. This is expressed by the formula $\Box\Diamond\langle HCnxt\rangle_{hr}$. (The $\langle\ \rangle_{hr}$ is needed to satisfy the syntax rules for temporal formulas; it's discussed in the next paragraph.) So, formula $HC \wedge \Box\Diamond\langle HCnxt\rangle_{hr}$ specifies a clock that never stops.

The syntax rules of TLA require us to write $\Box\Diamond\langle HCnxt\rangle_{hr}$ instead of the more obvious formula $\Box\Diamond HCnxt$. These rules are needed to guarantee that all TLA formulas are invariant under stuttering. In a behavior satisfying HC, an *HCnxt* step necessarily changes hr, so it is necessarily an $\langle HCnxt\rangle_{hr}$ step. Hence, $HC \wedge \Box\Diamond\langle HCnxt\rangle_{hr}$ is equivalent to the (illegal) formula $HC \wedge \Box\Diamond HCnxt$.

In a similar fashion, most of the actions we define do not allow stuttering steps. When we write $\langle A\rangle_v$ for some action A, the $\langle\ \rangle_v$ is usually needed only to satisfy the syntax rules. To avoid having to think about which variables A actually changes, we generally take the subscript v to be the tuple of all variables, which is changed if any variable changes. I will usually ignore the angle brackets and subscripts in informal discussions, and will describe $\Box\Diamond\langle HCnxt\rangle_{hr}$ as the assertion that there are infinitely many *HCnxt* steps.

Let's now modify specification *Spec* of module *Channel* (Figure 3) to require that every value sent is eventually received. We do this by conjoining a liveness condition to *Spec*. The analog of the liveness condition for the clock would be $\Box\Diamond\langle Rcv\rangle_{chan}$, which asserts that there are infinitely many *Rcv* steps. However, this would also require that infinitely many values are sent, and we don't want to make that requirement. In fact, we want to permit behaviors in which no value is ever sent, so no value is ever received. We just want to require that, if a value is ever sent, then it is eventually received.

It's enough to require only that the next value to be received always is eventually received, since this implies that all values sent are eventually received. More precisely, we require that it's always the case that, if there is a value to be received, then the next value to be received eventually is received. The next value is received by a *Rcv* step, so the requirement is:[20]

$$\Box(\text{There is an unreceived value} \;\Rightarrow\; \Diamond\langle Rcv\rangle_{chan})$$

There is an unreceived value iff action *Rcv* is enabled. (Recall that *Rcv* is enabled in a state iff it is possible to take a *Rcv* step from that state.) TLA$^+$ defines ENABLED A to be the predicate that is true iff action A is enabled. The liveness condition can then be written as:

$$(13)\quad \Box(\text{ENABLED}\,\langle Rcv\rangle_{chan} \;\Rightarrow\; \Diamond\langle Rcv\rangle_{chan})$$

[20] $\Box(F \Rightarrow \Diamond G)$ equals $F \leadsto G$, so we could write this formula more compactly with $\leadsto$. However, it is more convenient to keep it in the form $\Box(F \Rightarrow \Diamond G)$

In the ENABLED formula, it doesn't matter if we write Rcv or $\langle Rcv \rangle_{chan}$. We add the angle brackets so the two actions appearing in the formula are the same.

Because $\langle HCnxt \rangle_{hr}$ is always enabled during any behavior satisfying HC, we can rewrite the liveness condition $\Box\Diamond\langle HCnxt \rangle_{hr}$ for the hour clock as:

$$\Box(\text{ENABLED}\ \langle HCnxt \rangle_{hr} \Rightarrow \Diamond\langle HCnxt \rangle_{hr})$$

This suggests a general liveness condition on an action A:

$$\Box(\text{ENABLED}\ \langle A \rangle_v \Rightarrow \Diamond\langle A \rangle_v)$$

This formula asserts that if A ever becomes enabled, then an A step will eventually occur—even if A remains enabled for only a fraction of a nanosecond, and is never again enabled. The obvious difficulty of physically implementing such a requirement suggests that it's too strong. Instead, we define the weaker formula $\text{WF}_v(A)$ to equal:

(14) $\Box(\Box\text{ENABLED}\ \langle A \rangle_v \Rightarrow \Diamond\langle A \rangle_v)$

This formula asserts that if A ever becomes forever enabled, then an A step must eventually occur. WF stands for *W*eak *F*airness, and the condition $\text{WF}_v(A)$ is called *weak fairness on* A. Here are two formulas that are each equivalent to (14):

(15) $\Box\Diamond(\neg\text{ENABLED}\ \langle A \rangle_v) \lor \Box\Diamond\langle A \rangle_v$

(16) $\Diamond\Box(\text{ENABLED}\ \langle A \rangle_v) \Rightarrow \Box\Diamond\langle A \rangle_v$

These three formulas can be expressed in English as:

14. It's always the case that, if A is enabled forever, then an A step eventually occurs.
15. A is infinitely often disabled, or infinitely many A steps occur.
16. If A is eventually enabled forever, then infinitely many A steps occur.

The equivalence of these three formulas isn't obvious. Here's a proof that (14) is equivalent to (15), using some simple tautologies. Studying this proof, and these tautologies, will help you understand how to write liveness conditions.

$$\begin{array}{ll}
\multicolumn{2}{l}{\Box(\Box\text{ENABLED}\ \langle A \rangle_v \Rightarrow \Diamond\langle A \rangle_v)} \\
\quad \equiv \Box(\neg\Box\text{ENABLED}\ \langle A \rangle_v \lor \Diamond\langle A \rangle_v) & \textsf{Because } (F \Rightarrow G) \equiv (\neg F \lor G). \\
\quad \equiv \Box(\Diamond\neg\text{ENABLED}\ \langle A \rangle_v \lor \Diamond\langle A \rangle_v) & \textsf{Because } \neg\Box F \equiv \Diamond\neg F. \\
\quad \equiv \Box\Diamond(\neg\text{ENABLED}\ \langle A \rangle_v \lor \langle A \rangle_v) & \textsf{Because } \Diamond(F \lor G) \equiv \Diamond F \lor \Diamond G. \\
\quad \equiv \Box\Diamond(\neg\text{ENABLED}\ \langle A \rangle_v) \lor \Box\Diamond\langle A \rangle_v & \textsf{Because } \Box\Diamond(F \lor G) \equiv \Box\Diamond F \lor \Box\Diamond G.
\end{array}$$

The equivalence of (15) and (16) is proved as follows

$$\begin{array}{ll}
\multicolumn{2}{l}{\Box\Diamond(\neg\text{ENABLED}\ \langle A \rangle_v) \lor \Box\Diamond\langle A \rangle_v} \\
\quad \equiv \neg\Diamond\Box\text{ENABLED}\ \langle A \rangle_v \lor \Box\Diamond\langle A \rangle_v & \textsf{Because } \Box\Diamond\neg F \equiv \Box\neg\Box F \equiv \neg\Diamond\Box F. \\
\quad \equiv \Diamond\Box\text{ENABLED}\ \langle A \rangle_v \Rightarrow \Box\Diamond\langle A \rangle_v & \textsf{Because } (F \Rightarrow G) \equiv (\neg F \lor G).
\end{array}$$

We now show that the liveness conditions for the hour clock and the channel can be written as weak fairness conditions.

First, consider the hour clock. In any behavior satisfying its safety specification HC, an $\langle HCnxt \rangle_{hr}$ step is always enabled, so $\Box\Diamond(\text{ENABLED}\ \langle HCnxt \rangle_{hr})$ equals TRUE. Hence, HC implies that $\text{WF}_{hr}(HCnxt)$ is equivalent to $\Box\Diamond\langle HCnxt \rangle_{hr}$, our liveness condition for the hour clock.

Now, consider the liveness condition (13) for the channel. By (15), weak fairness on *Rcv* asserts that either (a) *Rcv* is disabled infinitely often, or (b) infinitely many *Rcv* steps occur (or both). Suppose *Rcv* becomes enabled at some instant. In case (a), *Rcv* must subsequently be disabled, which can occur only by a *Rcv* step. Case (b) also implies that there is a subsequent *Rcv* step. Weak fairness therefore implies that it's always the case that if *Rcv* is enabled, then a *Rcv* step eventually occurs. A closer look at this reasoning reveals that it is an informal proof of:

$$Spec \;\Rightarrow\; (\mathrm{WF}_{chan}(Rcv) \;\Rightarrow\; \Box(\textsc{Enabled}\,\langle Rcv\rangle_{chan} \Rightarrow \Diamond\langle Rcv\rangle_{chan}))$$

Because $\Box F$ implies F, for any formula F, it's not hard to check the truth of:

$$\Box(\textsc{Enabled}\,\langle Rcv\rangle_{chan} \Rightarrow \Diamond\langle Rcv\rangle_{chan}) \;\Rightarrow\; \mathrm{WF}_{chan}(Rcv)$$

Therefore, $Spec \land \mathrm{WF}_{chan}(Rcv)$ is equivalent to the conjunction of *Spec* and (13), so weak fairness of *Rcv* specifies the same liveness condition as (13) for the channel.

8.3 Liveness for the Memory Specification

Let's now strengthen the memory specification with the liveness requirement that every request must receive a response. (We don't require that a request is ever issued.) The liveness requirement is conjoined to the internal memory specification, formula *ISpec* of module *InternalMemory* (Figure 8).

We will express the liveness requirement in terms of weak fairness. This requires understanding when actions are enabled. The action $Rsp(p)$ is enabled only if the action

(17) $Reply(p,\, buf[p],\, memInt,\, memInt')$

is enabled. Recall that the operator *Reply* is a constant parameter, declared in module *MemoryInterface* (Figure 7). Without knowing more about this operator, we can't say when action (17) is enabled.

We assume that *Reply* actions are always enabled. That is, for any processor p and reply r, and any old value *miOld* of *memInt*, there is a new value *miNew* of *memInt* such that $Repl(p, r, miOld, miNew)$ is true. For simplicity, we can just assume that this is true for all p and r, and add the following assumption to the *MemoryInterface* module:

$$\textsc{assume}\ \forall\, p, r, miOld \,:\, \exists\, miNew \,:\, Repl(p, r, miOld, miNew)$$

We should also make a similar assumption for *Send*, but we don't need it here.

We will subscript our weak-fairness formulas with the tuple of all variables, so let's add the following definition to the *InternalMemory* module:

$$vars \;\triangleq\; \langle memInt,\, mem,\, ctl,\, buf\rangle$$

When processor p issues a request, it enables the $Do(p)$ action, which remains enabled until a $Do(p)$ step occurs. The weak-fairness condition $\mathrm{WF}_{vars}(Do(p))$ implies that this $Do(p)$ step must eventually occur. A $Do(p)$ step enables the $Rsp(p)$ action, which remains enabled until an $Rsp(p)$ step occurs. The weak-fairness condition $\mathrm{WF}_{vars}(Rsp(p))$ implies that this $Rsp(p)$ step, which produces the desired response, must eventually occur. Hence, the requirement

(18) $\mathrm{WF}_{vars}(Do(p)) \land \mathrm{WF}_{vars}(Rsp(p))$

assures that every request issued by processor p must eventually receive a reply.

We can rewrite condition (18) in the slightly simpler form of weak fairness on the action $Do(p) \lor Rsp(p)$. The disjunction of two actions is enabled iff one or both of them are enabled. A $Req(p)$ step enables $Do(p)$, thereby enabling $Do(p) \lor Rsp(p)$. The only $Do(p) \lor Rsp(p)$

step then possible is a $Do(p)$ step, which enables $Rsp(p)$ and hence $Do(p) \lor Rsp(p)$. At this point, the only $Do(p) \lor Rsp(p)$ step possible is a $Rsp(p)$ step, which disables $Rsp(p)$ and leaves $Do(p)$ disabled, hence disabling $Do(p) \lor Rsp(p)$. This all shows that (18) is equivalent to $\mathrm{WF}_{vars}(Do(p) \lor Rsp(p))$, weak fairness on the single action $Do(p) \lor Rsp(p)$.

Weak fairness of $Do(p) \lor Rsp(p)$ guarantees that every request by processor p receives a response. We want every request from every processor to receive a response. So, the liveness condition for the memory specification asserts weak fairness of $Do(p) \lor Rsp(p)$ for every processor p:[21]

$$Liveness \;\triangleq\; \forall\, p \in Proc \,:\, \mathrm{WF}_{vars}(Do(p) \lor Rsp(p))$$

The example of actions $Do(p)$ and $Rsp(p)$ raises the general question: when is the conjunction of weak fairness on actions $A_1, \ldots, A_n$ equivalent to weak fairness of their disjunction $A_1 \lor \ldots \lor A_n$? The general answer is complicated, but here's a sufficient condition:

> **WF Conjunction Rule** If $A_1, \ldots, A_n$ are actions such that, for any distinct i and j, whenever action A_i is enabled, action A_j cannot become enabled until an A_i step occurs, then $\mathrm{WF}_v(A_1) \land \ldots \land \mathrm{WF}_v(A_n)$ is equivalent to $\mathrm{WF}_v(A_1 \lor \ldots \lor A_n)$.

This rule is stated rather informally. It can be interpreted as an assertion about a particular behavior σ, in which case its conclusion is

$$\sigma \models (\mathrm{WF}_v(A_1) \land \ldots \land \mathrm{WF}_v(A_n)) \;\equiv\; \mathrm{WF}_v(A_1 \lor \ldots \lor A_n)$$

Alternatively, it can be formalized as the assertion that if a formula F implies the hypothesis, then F implies the equivalence of $\mathrm{WF}_v(A_1) \land \ldots \land \mathrm{WF}_v(A_n)$ and $\mathrm{WF}_v(A_1 \lor \ldots \lor A_n)$.

Conjunction and disjunction are special cases of universal and existential quantification, respectively. For example, $A_1 \lor \ldots \lor A_n$ is equivalent to $\exists\, i \in 1 \mathrel{..} n : A_i$. So, we can trivially restate the WF Conjunction Rule as a condition on when formulas $\forall\, i \in S : \mathrm{WF}_v(A_i)$ and $\mathrm{WF}_v(\exists\, i \in S : A_i)$ are equivalent, for a finite set S. The resulting rule is actually valid for any set S:

> **WF Quantifier Rule** If the A_i are actions, for all $i \in S$, such that, for any distinct i and j in S, whenever action A_i is enabled, action A_j cannot become enabled until an A_i step occurs, then $\forall\, i \in S : \mathrm{WF}_v(A_i)$ and $\mathrm{WF}_v(\exists\, i \in S : A_i)$ are equivalent.

8.4 Strong Fairness

Formulations (15) and (16) of $\mathrm{WF}_v(A)$ contain the operators infinitely often ($\Box\Diamond$) and eventually always ($\Diamond\Box$). Eventually always is stronger than (implies) infinitely often. We define $\mathrm{SF}_v(A)$, *strong fairness* of action A, to be either of the following equivalent formulas:

(19) $\Diamond\Box(\neg \textsc{Enabled}\, \langle A \rangle_v) \lor \Box\Diamond\langle A \rangle_v$

(20) $\Box\Diamond\textsc{Enabled}\, \langle A \rangle_v \Rightarrow \Box\Diamond\langle A \rangle_v$

Intuitively, these two formulas assert:

(19) A is eventually disabled forever, or infinitely many A steps occur.

(20) If A is infinitely often enabled, then infinitely many A steps occur.

[21] Although we haven't yet discussed quantification in temporal formulas, the meaning of the formula $\forall\, p \in Proc : \ldots$ should be clear.

The proof that these two formulas are equivalent is similar to the proof of equivalence of (15) and (16).

The analogs of the WF Conjunction and WF Quantifier Rules hold for strong fairness—for example:

> **SF Conjunction Rule** If $A_1, \ldots, A_n$ are actions such that, for any distinct i and j, whenever action A_i is enabled, action A_j cannot become enabled until an A_i step occurs, then $\mathrm{SF}_v(A_1) \wedge \ldots \wedge \mathrm{SF}_v(A_n)$ is equivalent to $\mathrm{SF}_v(A_1 \vee \ldots \vee A_n)$.

It's not hard to see that strong fairness is stronger than weak fairness—that is, $\mathrm{SF}_v(A)$ implies $\mathrm{WF}_v(A)$, for any v and A. We can express weak and strong fairness as follows.

- Weak fairness of A asserts that an A step must eventually occur if A is *continuously* enabled.
- Strong fairness of A asserts that an A step must eventually occur if A is *continually* enabled.

Continuously means without interruption. *Continually* means repeatedly, possibly with interruptions.

Strong fairness need not be strictly stronger than weak fairness. Weak and strong fairness of an action A are equivalent iff A infinitely often disabled implies that either A is eventually always disabled, or infinitely many A steps occur. This is expressed formally by the tautology:

$$(\mathrm{WF}_v(A) \equiv (\mathrm{SF}_v(A)) \;\equiv\; (\Box\Diamond(\neg\textsc{Enabled}\,\langle A\rangle_v) \Rightarrow \Diamond\Box(\neg\textsc{Enabled}\,\langle A\rangle_v) \vee \Box\Diamond\langle A\rangle_v)$$

In the channel example, weak and strong fairness of *Rcv* are equivalent because *Spec* implies that, once enabled, *Rcv* can be disabled only by a *Rcv* step; so if it is disabled infinitely often, then it either eventually remains disabled forever, or else it is disabled infinitely often by *Rcv* steps.

Strong fairness can be more difficult to implement than weak fairness, and it is a less common requirement. A strong fairness condition should be used in a specification only if it is needed. When strong and weak fairness are equivalent, the fairness property should be written as weak fairness.

Liveness properties can be subtle. Expressing them with *ad hoc* temporal formulas can lead to errors. We will specify liveness as the conjunction of fairness properties whenever possible—and it almost always is possible. Having a uniform way of expressing liveness makes specifications easier to understand. Section 8.7.2 discusses an even more compelling reason for using fairness to specify liveness.

8.5 Liveness for the Write-Through Cache

Let's now add liveness to the write-through cache, specified in Figures 10 and 11. We want our specification to guarantee that every request eventually receives a response, without requiring that any requests are issued. This requires fairness on all the actions that make up the next-state action *Next* except the $Req(p)$ action (which issues a request) and the $Evict(p, a)$ action (which evicts an address from the cache). If any other action were ever enabled without being executed, then some request might not generate a response—except for one special case. If the *memQ* queue contains only write requests, and *memQ* is not full (has fewer than *QLen* elements), then not executing a *MemQWr* action would not prevent any responses. (Remember that a response to a write request can be issued before the value is written to memory.) We'll return to this exception later. For simplicity, let's require fairness for the *MemQWr* action too.

Our liveness condition has to assert fairness of the following actions:

$$MemQWr \quad MemQRd \quad Rsp(p) \quad RdMiss(p) \quad DoRd(p) \quad DoWr(p)$$

for all p in *Proc*. We now must decide whether to assert weak or strong fairness for these actions. Weak and strong fairness are equivalent for an action that, once enabled, remains enabled until it is executed. This is the case for all of these actions except $RdMiss(p)$ and $DoWr(p)$. These two actions append a request to the *memQ* queue, and are disabled if that queue is full. A $RdMiss(p)$ or $DoWr(p)$ could be enabled, and then become disabled because a $RdMiss(q)$ or $DoWr(q)$, for a different processor q, appends a request to *memQ*. We therefore need strong fairness for the $RdMiss(p)$ and $DoWr(p)$ actions. So, the fairness conditions we need are:

Weak Fairness $\quad Rsp(p)$, $DoRd(p)$, $MemQWr$, and $MemQRd$

Strong Fairness $\quad RdMiss(p)$ and $DoWr(p)$

As before, let's define *vars* to be the tuple of all variables.

$$vars \triangleq \langle memInt, mem, buf, ctl, cache, memQ \rangle$$

We could just write the liveness condition as

$$\begin{array}{ll}(21) & \land \forall p \in Proc : \land \mathrm{WF}_{vars}(Rsp(p)) \land \mathrm{WF}_{vars}(DoRd(p)) \\ & \qquad\qquad\qquad\quad \land \mathrm{SF}_{vars}(RdMiss(p)) \land \mathrm{SF}_{vars}(DoWr(p)) \\ & \land \mathrm{WF}_{vars}(MemQWr) \land \mathrm{WF}_{vars}(MemQRd)\end{array}$$

However, I prefer replacing the conjunction of fairness conditions by a single fairness condition on a disjunction, as we did in Section 8.3 for the memory specification. The WF and SF Conjunction Rules (page 8.3 and 8.4) easily imply that the liveness condition (21) can be rewritten as

$$\begin{array}{ll}(22) & \land \forall p \in Proc : \land \mathrm{WF}_{vars}(Rsp(p) \lor DoRd(p)) \\ & \qquad\qquad\qquad\quad \land \mathrm{SF}_{vars}(RdMiss(p) \lor DoWr(p)) \\ & \land \mathrm{WF}_{vars}(MemQWr \lor MemQRd)\end{array}$$

We can now try to simplify (22) by applying the WF Quantifier Rule (Section 8.3) to replace $\forall p \in Proc : \mathrm{WF}_{vars}(\ldots)$ with $\mathrm{WF}_{vars}(\exists p \in Proc : \ldots)$. However, that rule doesn't apply; it's possible for $Rsp(p) \lor DoRd(p)$ to be enabled for two different processors p at the same time. In fact, the two formulas are not equivalent. Similarly, the analogous rule for strong fairness doesn't apply. Formula (22) is as simple as we can make it.

Let's return to the observation that we don't have to execute *MemQWr* if the *memQ* queue is not full and contains only write requests. Let's define *QCond* to be the assertion that *memQ* is not full and contains only write requests:

$$\begin{array}{rcl}QCond & \triangleq & \land Len(memQ) < QLen \\ & & \land \forall i \in 1..Len(memQ) : memQ[i][2].op = \text{“Wr”}\end{array}$$

We have to eventually execute a *MemQWr* action only when it's enabled and *QCond* is true, which is the case iff the action $QCond \land MemQWr$ is enabled. In this case, a *MemQWr* step is a $QCond \land MemQWr$ step. Hence, it suffices to require weak fairness of the action $QCond \land MemQWr$. We can therefore replace the second conjunct of (22) with

$$\mathrm{WF}_{vars}((QCond \land MemQWr) \lor MemQRd)$$

We would do this if we wanted the specification to describe the weakest liveness condition that implements the memory specification's liveness condition. However, if the specification were a description of an actual device, then that device would probably implement weak fairness on all *MemQWr* actions, so we would take (22) as the liveness condition.

8.6 Quantification

I've already mentioned, in Section 8.1, that the ordinary quantifiers of predicate logic can be applied to temporal formulas. For example, the meaning of the formula $\exists\, r : F$, for any temporal formula F, is given by

$$\sigma \models (\exists\, r : F) \;\triangleq\; \exists\, r : (\sigma \models F)$$

where σ is any behavior. The meaning of $\forall\, r : F$ is similarly defined.

The symbol r in $\exists\, r : F$ is usually called a bound variable. But we've been using the term *variable* to mean something else—something that's declared by a VARIABLE statement in a module. The bound "variable" r is actually a constant in these formulas—a value that is the same in every state of the behavior.[22] For example, the formula $\exists\, r : \Box(x = r)$ asserts that x always has the same value.

The bounded quantifiers are defined in a similar way—for example,

$$\sigma \models (\exists\, r \in S : F) \;\triangleq\; \exists\, r \in S : (\sigma \models F)$$

For this formula to make sense, S must be a constant.[23] The symbol r is declared to be a constant in formula F. The expression S lies outside the scope of the declaration of r, so the symbol r cannot occur in S.

One can, in a similar way, define CHOOSE to be a temporal operator. However, it's not needed for writing specifications, so we won't.

We have been using the operator $\boldsymbol{\exists}$ as a hiding operator. Intuitively, $\boldsymbol{\exists}\, x : F$ means F with variable x hidden. In this formula, x is declared to be a variable in formula F. Unlike $\exists\, r : F$, which asserts the existence of a single value r, the formula $\boldsymbol{\exists}\, x : F$ asserts the existence of a value for x in each state of a behavior.

The precise definition of $\boldsymbol{\exists}$ is a bit tricky because, as discussed in Section 8.1, the formula $\boldsymbol{\exists}\, x : F$ should be invariant under stuttering. To define it, we first define $\sigma \sim_x \tau$ to be true iff σ can be obtained from τ (or vice-versa) by adding and/or removing stuttering steps and changing the values assigned to x by its states. To define $\sim_x$ precisely, we define two behaviors σ and τ to be *stuttering-equivalent* iff removing all stuttering steps from each of them produces the same sequence of states. Next, let $\sigma_{x \leftarrow 0}$ be the behavior that is the same as σ except that, in each state, x is assigned the value 0.[24] We can then define $\sigma \sim_x \tau$ to be true iff $\sigma_{x \leftarrow 0}$ and $\tau_{x \leftarrow 0}$ are stuttering equivalent. Finally, the meaning of $\boldsymbol{\exists}$ is defined by letting $\sigma \models (\boldsymbol{\exists}\, x : F)$ be true iff there exists some behavior τ such that $\tau \sim_x \sigma$ and $\tau \models F$ are true. If you find this too confusing, don't worry about it. For writing specifications, it suffices to just think of $\boldsymbol{\exists}\, x : F$ as F with x hidden.

TLA also has a temporal universal quantifier $\boldsymbol{\forall}$, defined by:

$$\boldsymbol{\forall}\, x : F \;\triangleq\; \neg \boldsymbol{\exists}\, x : \neg F$$

This operator is hardly ever used.

TLA$^+$ does not allow bounded versions of the operators $\boldsymbol{\exists}$ and $\boldsymbol{\forall}$. If, for some reason, you want to write $\boldsymbol{\exists}\, x \in S : F$, you can simply write $\boldsymbol{\exists}\, x : (x \in S) \wedge F$ instead. However, I don't know why anyone would want to write such a formula.

---

[22] Logicians use the term *flexible variable* for a TLA variable, and the term *rigid variable* for a symbol like r that represents a constant.

[23] We can let $\exists\, r \in S : F$ equal $\exists\, r : (r \in S) \wedge F$, which makes sense if S is a state function, not just a constant. However, TLA$^+$ requires S to be a constant in $\exists\, r \in S : F$. If you want it to be a state function, you have to write $\exists\, r : (r \in S) \wedge F$.

[24] The use of 0 is arbitrary; any fixed value would do.

8.7 Temporal Logic Examined

8.7.1 A Review

Let's look at the shapes of the specifications that we've written so far. We started with the simple form

(23) $Init \wedge \Box[Next]_{vars}$

where *Init* is the initial predicate, *Next* the next-state action, and *vars* the tuple of all variables. This kind of specification is, in principle, quite straightforward.

We then introduced hiding: using $\exists$ to bind variables that should not appear in the specification. Those bound variables, also called *hidden* or *internal* variables, serve only to help describe how the values of the free variables (also called visible variables) change.

Hiding variables is easy enough, and it is mathematically elegant and philosophically satisfying. However, in practice, it doesn't make much difference to a specification. A comment can also tell a reader that a variable should be regarded as internal. Explicit hiding allows implementation to mean implication. A lower-level specification that describes an implementation can be expected to imply a specification only if the specification's internal variables, whose values don't really matter, are explicitly hidden. Otherwise, implementation means implementation under a refinement mapping. (See Section 6.8.)

To express liveness, the specification (23) is strengthened to the form

(24) $Init \wedge \Box[Next]_{vars} \wedge Liveness$

where *Liveness* is the conjunction of formulas of the form $\mathrm{WF}_{vars}(A)$ and/or $\mathrm{SF}_{vars}(A)$, for actions A. (I'm considering universal quantification to be a form of conjunction.)

8.7.2 Machine Closure

In the specifications of the form (24) we've written so far, the actions A whose fairness properties appear in formula *Liveness* have one thing in common: they are all *subactions* of the next-state action *Next*. An action A is a subaction of *Next* iff every A step is a *Next* step. Equivalently, A is a subaction of *Next* iff A implies *Next*. In almost all specifications of the form (24), formula *Liveness* should be the conjunction of weak and/or strong fairness formulas for subactions of *Next*. I'll now briefly explain why.

When we look at the specification (24), we expect *Init* to constrain the initial state, *Next* to constrain what steps may occur, and *Liveness* to describe only what must eventually happen. However, consider the following formula

(25) $(x = 0) \wedge \Box[x' = x + 1]_x \wedge \mathrm{WF}_x((x > 99) \wedge (x' = x - 1))$

The first two conjuncts of (25) assert that x is initially 0 and that any step either increments x by 1 or leaves it unchanged. Hence, they imply that if x ever exceeds 99, then it forever remains greater than 99. The weak fairness property asserts that, if this happens, then x must eventually be decremented by 1—contradicting the second conjunct. Hence, (25) implies that x can never exceed 99, so that formula is equivalent to

$$(x = 0) \wedge \Box[(x < 99) \wedge (x' = x + 1)]_x$$

Conjoining the weak fairness property to the first two conjuncts of (25) forbids an $x' = x + 1$ step when $x = 99$.

A specification of the form (24) is called *machine closed* iff the conjunct *Liveness* does not constrain the initial state or what steps may occur. We almost never want to write a specification that isn't machine closed. If we do write one, it's almost always by mistake. Specification (24) is guaranteed to be machine closed if *Liveness* is the conjunction of weak and/or strong fairness properties for subactions of *Next*.[25] This condition doesn't apply to specification (25), which is not machine closed, because $(x > 99) \wedge (x' = x - 1)$ is not a subaction of $x' = x + 1$.

Liveness requirements are philosophically satisfying. A specification of the form (23), which specifies only a safety property, allows behaviors in which the system does nothing. Therefore, the specification is satisfied by a system that does nothing. Expressing liveness requirements with fairness properties is less satisfying. These properties are subtle and it's easy to get them wrong. It requires some thought to determine that the liveness condition for the write-through cache, formula (22), does imply that every request receives a reply.

It's tempting to express liveness properties more directly, without using fairness properties. For example, it's easy to write a temporal formula asserting for the write-through cache that every request receives a response. When processor p issues a request, it sets $ctl[p]$ to "rdy". We just have to assert that a state in which $ctl[p] =$ "rdy" is true leads to a $Rsp(p)$ step—for every processor p:

$$(26)\quad \forall\, p \in Proc \,:\, (ctl[p] = \text{“}\mathsf{rdy}\text{”}) \leadsto \langle Rsp(p) \rangle_{vars}$$

(The operator $\leadsto$ is defined in Section 8.1.) While such formulas are appealing, they are dangerous. It's very easy to make a mistake and write a specification that isn't machine closed.

Except in unusual circumstances, you should express liveness with fairness properties for subactions of the next-state action. These are the most straightforward specifications, and hence the easiest to write and to understand. Most system specifications, even if very detailed and complicated, can be written in this straightforward manner. The exceptions are usually in the realm of subtle, high-level specifications that attempt to be very general. An example is the specification of sequential consistency in [6].

8.7.3 The Unimportance of Liveness

While philosophically important, in practice the liveness property of (24) is not as important as the safety part, $Init \wedge \Box[Next]_{vars}$. The ultimate purpose of writing a specification is to avoid errors. Experience shows that most of the benefit from writing and using a specification comes from the safety part. On the other hand, the liveness property is usually easy enough to write. It typically constitutes less than five percent of a specification. So, you might as well write the liveness part. However, when verifying the correctness of the specification, most of your effort should be devoted to the safety part.

8.7.4 Temporal Logic Considered Confusing

The most general type of specification I've discussed so far has the form

$$(27)\quad \boldsymbol{\exists}\, v_1, \ldots, v_n \,:\, Init \wedge \Box[Next]_{vars} \wedge Liveness$$

where *Liveness* is the conjunction of fairness properties of subactions of *Next*. This is a very restricted class of temporal-logic formulas. Temporal logic is quite expressive, and one can combine its operators in all sorts of ways to express a wide variety of properties. This suggests the following approach to writing a specification: express each property that the system must satisfy with a temporal formula, and then conjoin all these formulas. For example, formula (26) above expresses the property of the write-through cache that every request eventually receives a response.

This approach is philosophically appealing. It has just one problem: it's practical for only the very simplest of specifications—and even for them, it seldom works well. The unbridled

[25] More precisely, this is the case for a finite or countably infinite conjunction of properties.

use of temporal logic produces formulas that are hard to understand. Conjoining several of these formulas produces a specification that is impossible to understand.

The basic form of a TLA specification is (27). Most specifications should have this form. We can also use such specifications as building blocks, combining them to form larger specifications, as described in [1]. However, such specifications are of limited practical use. Most engineers need know only how to write specifications of the form (27). Indeed, they can get along quite well with specifications of the form (23).

9 Writing a Specification—Some Advice

You have now learned all you need to know about TLA$^+$ to write your own specifications. Here are a few additional hints to help you get started.

Why to Specify

Specifications are written to help eliminate errors. Writing a specification requires effort; the benefits it provides must be worth that effort. There are several benefits:

- Writing a TLA$^+$ specification can help the design process. Having to describe a design precisely often reveals problems—subtle interactions and "corner cases" that are easily overlooked. These problems are easier to correct when discovered in the design phase rather than after implementation has begun.

- A TLA$^+$ specification can provide a clear, concise way of communicating a design. It helps ensure that the designers agree on what they have designed, and it provides a valuable guide to the engineers who implement and test the system. It may also help users understand the system.

- A TLA$^+$ specification is a formal description to which tools can be applied to help find errors in the design and to help in testing the system. Some tools for TLA$^+$ specifications are being built.

Whether these benefits justify the effort of writing the specification depends on the nature of the project. Specification is not an end in itself; it is just one of many tools that an engineer should be able to use when appropriate.

What to Specify

Although we talk about specifying a system, that's not what we do. A specification is a mathematical model of a particular view of some part of a system. When writing a specification, the first thing you must choose is exactly what part of the system you want to describe. Sometimes the choice is obvious; often it isn't. The cache-coherence protocol of a real multiprocessor computer may be intimately connected with how the processors execute instructions. Finding an abstraction that describes the coherence protocol while suppressing the details of instruction execution may be difficult. It may require defining an interface between the processor and the memory that doesn't exist in the actual system design.

Remember that the purpose of a specification is to help avoid errors. You should specify those parts of the system for which a specification is most likely to reveal errors. TLA$^+$ is particularly effective at revealing concurrency errors—ones that arise through the interaction of asynchronous components. So, you should concentrate your efforts on the parts of the system that are most likely to have such errors.

The Grain of Atomicity

After choosing what part of the system to specify, you must choose the specification's level of abstraction. The most important aspect of the level of abstraction is the grain of atomicity, the choice of what system changes are represented as a single step of a behavior. Sending a message in an actual system involves multiple suboperations, but we usually represent it as a single step. On the other hand, the sending of a message and its receipt are usually represented as separate steps when specifying a distributed system.

The same sequence of system operations is represented by a shorter sequence of steps in a coarser-grained representation than in a finer-grained one. This almost always makes the coarser-grained specification simpler than the finer-grained one. However, the finer-grained specification more accurately describes the behavior of the actual system. A coarser-grained specification may fail to reveal important details of the system.

There is no simple rule for deciding on the grain of atomicity. However, there is one way of thinking about the granularity that can help. To describe it, we need the TLA$^+$ action-composition operator "$\cdot$". If A and B are actions, then the action $A{\cdot}B$ is executed by first executing A then B in a single step. More precisely, $A \cdot B$ is the action defined by letting $s \rightarrow t$ be an $A \cdot B$ step iff there exists a state u such that $s \rightarrow u$ is an A step and $u \rightarrow t$ is a B step.

When determining the grain of atomicity, we must decide whether to represent the execution of an operation as a single step or as a sequence of steps, each corresponding to the execution of a suboperation. Let's consider the simple case of an operation consisting of two suboperations that are executed sequentially, where those suboperations are described by the two actions R and L. (Executing R enables L and disables R.) When the operation's execution is represented by two steps, each of those steps is an R step or an L step. The operation is then described with the action $R \vee L$. When its execution is represented by a single step, the operation is described with the action $R{\cdot}L$.[26] Let $S2$ be the finer-grained specification in which the operation is executed in two steps, and let $S1$ be the coarser-grained specification in which it is executed as a single $R \cdot L$ step. To choose the grain of atomicity, we must choose whether to take $S1$ or $S2$ as the specification. Let's examine the relation between the two specifications.

We can transform any behavior σ satisfying $S1$ into a behavior $\widehat{\sigma}$ satisfying $S2$ by replacing each step $s \xrightarrow{R\cdot L} t$ with the pair of steps $s \xrightarrow{R} u \xrightarrow{L} t$, for some state u. If we regard σ as being equivalent to $\widehat{\sigma}$, then we can regard $S1$ as being a strengthened version of $S2$—one that allows fewer behaviors. Specification $S1$ requires that each R step be followed immediately by an L step, while $S2$ allows behaviors in which other steps come between the R and L steps. To choose the appropriate grain of atomicity, we must decide whether those additional behaviors allowed by $S2$ are important.

The additional behaviors allowed by $S2$ are not important if the actual system executions they describe are also described by behaviors allowed by $S1$. So, we can ask whether each behavior τ satisfying $S2$ has a corresponding behavior $\widetilde{\tau}$ satisfying $S1$ that is, in some sense, equivalent to τ. One way to construct $\widetilde{\tau}$ from τ is to transform a sequence of steps

$$(28)\quad s \xrightarrow{R} u_1 \xrightarrow{A_1} u_2 \xrightarrow{A_2} u_3 \ \ldots\ u_n \xrightarrow{A_n} u_{n+1} \xrightarrow{L} t$$

into the sequence

$$(29)\quad s \xrightarrow{A_1} v_1 \ \ldots\ v_{k-2} \xrightarrow{A_k} v_{k-1} \xrightarrow{R} v_k \xrightarrow{L} v_{k+1} \xrightarrow{A_{k+1}} v_{k+2} \ \ldots\ v_{n+1} \xrightarrow{A_n} t$$

where the A_i are other system actions that can be executed between the R and L steps. Both sequences start in state s and end in state t, but the intermediate states may be different.

[26] We actually describe the operation with an ordinary action, like the ones we've been writing, that is equivalent to $R{\cdot}L$. The operator "$\cdot$" rarely appears in an actual specification. If you're ever tempted to use it, look for a better way to write the specification; you can probably find one.

When is such a transformation possible? An answer can be given in terms of commutativity relations. We say that actions A and B commute if performing them in either order produces the same result. Formally, A and B commute iff $A \cdot B$ is equivalent to $B \cdot A$. A simple sufficient condition for commutativity is that two actions commute if they change the values of different variables and neither enables or disables the other. It's not hard to see that we can transform (28) to (29) in the following two cases:

- R commutes with each A_i. (In this case, $k = n$.)
- L commutes with each A_i. (In this case, $k = 0$.)

In general, if an operation consists of a sequence of m subactions, we must decide whether to choose the finer-grained representation $O_1 \lor O_2 \lor \ldots \lor O_m$ or the coarser-grained one $O_1 \cdot O_2 \cdots O_m$. The generalization of the transformation from (28) to (29) is one that transforms an arbitrary behavior satisfying the finer-grained specification into one in which the sequence of O_1, O_2, …, O_m steps come one right after the other. Such a transformation is possible if all but one of the actions O_i commute with every other system action. Commutativity can be replaced by weaker conditions [2, 3, 11]. However, it is the most common case.

By commuting actions and replacing a sequence $s \xrightarrow{O_1} \cdots \xrightarrow{O_m} t$ of steps by a single $O_1 \cdots O_m$ step, you may be able to transform any behavior of a finer-grained specification into a corresponding behavior of a coarser-grained one. But that doesn't mean that the coarser-grained specification is just as good as the finer-grained one. The sequences (28) and (29) are not the same, and a sequence of O_i steps is not the same as a single $O_1 \cdots O_m$ step. Whether you can consider the transformed behavior to be equivalent to the original one, and use the coarser-grained specification, depends on the particular system you are specifying and on the purpose of the specification. Understanding the relation between finer- and coarser-grained specifications can help you choose between them; it won't make the choice for you.

The Data Structures

Another aspect of a specification's level of abstraction is the accuracy with which it describes the system's data structures. For example, should the specification of a programming interface describe the actual layout of a procedure's arguments in memory, or should the arguments be represented more abstractly?

To answer such a question, you must remember that the purpose of the specification is to help catch errors. A precise description of the layout of procedure arguments will help prevent errors caused by misunderstandings about that layout, but at the cost of complicating the programming interface's specification. The cost is justified only if such errors are likely to be a real problem and the TLA$^+$ specification provides the best way to avoid them.

If the purpose of the specification is to catch errors caused by the asynchronous interaction of concurrently executing components, then detailed descriptions of data structures will be a needless complication. So, you will probably want to use high-level, abstract descriptions of the system's data structures in the specification. For example, to specify a program interface, you might introduce constant parameters to represent the actions of calling and returning from a procedure—parameters analogous to *Send* and *Reply* of the memory interface described in Section 6.1.

Writing the Specification

Once you've chosen the part of the system to specify and the level of abstraction, you're ready to start writing the TLA$^+$ specification. We've already seen how this is done; let's review the steps.

First, pick the variables and define the type invariant and initial predicate. In the course of doing this, you will determine the constant parameters and assumptions about them that you need. You may also have to define some additional constants.

Next, write the next-state action, which forms the bulk of the specification. Sketching a few sample behaviors may help you get started. You must first decide how to decompose the next-state action as the disjunction of actions describing the different kinds of system operations. You then define those actions. The goal is to make the action definitions as compact and easy to read as possible. This requires carefully structuring them. One way to reduce the size of a specification is to define state predicates and state functions that are used in several different action definitions. When writing the action definitions, you will determine which of the standard modules you will need to use and add the appropriate EXTENDS statement. You may also have to define some constant operators for the data structures that you are using.

You must now write the temporal part of the specification. If you want to specify liveness properties, you have to choose the fairness conditions. You then combine the initial predicate, next-state action, and any fairness conditions you've chosen into the definition of a single temporal formula that is the specification.

Finally, you can assert theorems about the specification. If nothing else, you may want to add a type-correctness theorem.

Some Further Hints

Here are a few miscellaneous suggestions that may help you write better specifications.

Don't be too clever. Cleverness can make a specification hard to read—and even wrong. The formula $q = \langle h' \rangle \circ q'$ may look like a nice, short way of writing:

(30) $(h' = \mathit{Head}(q)) \land (q' = \mathit{Tail}(q))$

But not only is $q = \langle h' \rangle \circ q'$ harder to understand than (30), it's also wrong. We don't know what $a \circ b$ equals if a and b are not both sequences, so we don't know whether $h' = \mathit{Head}(q)$ and $q' = \mathit{Tail}(q)$ are the only values of h' and q' that satisfy $q = \langle h' \rangle \circ q'$. There could be other values of h' and q', which are not sequences, that satisfy the formula $q = \langle h' \rangle \circ q'$.

Don't assume that values are unequal just because they look different. The rules of TLA$^+$ do not imply that $1 \neq$ "a". If the system can send a message that is either a string or a number, represent the message as a record with a *type* and *value* field—for example,

$$[\mathit{type} \mapsto \text{“String”},\ \mathit{value} \mapsto \text{“a”}] \quad \text{or} \quad [\mathit{type} \mapsto \text{“Nat”},\ \mathit{value} \mapsto 1]$$

Write comments as comments; don't put them into the specification itself. I have seen people write things like the following action definition:

$$\begin{array}{lll} A & \triangleq & \lor \land x \geq 0 \\ & & \land \ldots \\ & & \lor \land x < 0 \\ & & \land \text{FALSE} \end{array}$$

The second disjunct is meant to indicate that the writer intended A not to be enabled when $x < 0$. But that disjunct is completely redundant and serves only as a form of comment. It's better to write:

$$\begin{array}{lll} A & \triangleq & \land x \geq 0 \quad \textsf{A is not enabled if } x < 0 \\ & & \land \ldots \end{array}$$

When and How to Specify

Specifications are often written later than they should be. Engineers are usually under severe time constraints, and they may feel that writing a specification will slow them down. Only after a design has become so complex that they need help understanding it do engineers think about writing a precise specification.

Writing a specification helps you think clearly. Thinking clearly is hard; we can use all the help we can get. Making specification part of the design process can improve the design.

I have described how to write a specification assuming that the system design already exists. But it's better to write the specification as the system is being designed. The specification will start out being incomplete and probably incorrect. For example, an initial specification of the write-through cache of Section 6.6 might include the definition:

$RdMiss(p) \triangleq$ Enqueue a request to write value from memory to p's cache.
Some enabling condition must be conjoined here.
$\land\ memQ' = Append(memQ, buf[p])$ Append request to $memQ$.
$\land\ ctl' = [ctl \text{ EXCEPT } ![p] = \text{“?”}]$ Set $ctl[p]$ to value to be determined later.
$\land\ \text{UNCHANGED } \langle memInt, mem, buf, cache \rangle$

Some system functionality will at first be omitted; it can be included later by adding new disjuncts to the next-state action. Tools can be applied to these preliminary specifications to help find design errors.

References

[1] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[2] Ernie Cohen and Leslie Lamport. Reduction in tla. In David Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998.

[3] Thomas W. Doeppner, Jr. Parallel program correctness through refinement. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 155–169. ACM, January 1977.

[4] M.P. Herlihy and J.M. Wing. Axioms for concurrent objects. In *Proceedings of the Fourteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 13–26, Munich, January 1987. ACM.

[5] Donald E. Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1994.

[6] Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy caching in TLA. To appear in *Distributed Computing*.

[7] Leslie Lamport. TLA—temporal logic of actions. At URL `http://www.research.digital.com/SRC/tla/` on the World Wide Web. It can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.

[8] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[9] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, July 1997.

[10] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach, New York, 1969.

[11] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[12] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.

Logic

$\land \quad \lor \quad \neg \quad \Rightarrow \quad \equiv$

TRUE FALSE BOOLEAN [the set {TRUE, FALSE}]

$\forall x : p \quad \exists x : p \quad \forall x \in S : p$ (1) $\quad \exists x \in S : p$ (1)

CHOOSE $x : p$ [An x satisfying p] CHOOSE $x \in S : p$ [An $x \in S$ satisfying p]

Sets

$= \quad \neq \quad \in \quad \notin \quad \cup \quad \cap \quad \subseteq \quad \setminus$ [set difference]

$\{e_1, \ldots, e_n\}$	[Set consisting of elements e_i]
$\{x \in S : p\}$ (2)	[Set of elements x in S satisfying p]
$\{e : x \in S\}$ (1)	[Set of elements e such that x in S]
SUBSET S	[Set of subsets of S]
UNION S	[Union of all elements of S]

Functions (3)

$f[e]$	[Function application]
DOMAIN f	[Domain of function f]
$[x \in S \mapsto e]$ (1)	[Function f such that $f[x] = e$ for $x \in S$]
$[S \rightarrow T]$	[Set of functions f with $f[x] \in T$ for $x \in S$]
$[f$ EXCEPT $![e_1] = e_2]$ (4)	[Function $\widehat{f}$ equal to f except $\widehat{f}[e_1] = e_2$]

Records

$e.h$	[The h-component of record e]
$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$	[The record whose h_i component is e_i]
$[h_1 : S_1, \ldots, h_n : S_n]$	[Set of all records with h_i component in S_i]
$[r$ EXCEPT $!.h = e]$ (4)	[Record $\widehat{r}$ equal to r except $\widehat{r}.h = e$]

Tuples

$e[i]$	[The i^{th} component of tuple e]
$\langle e_1, \ldots, e_n \rangle$	[The n-tuple whose i^{th} component is e_i]
$S_1 \times \ldots \times S_n$	[The set of all n-tuples with i^{th} component in S_i]

Strings and Numbers

"$\mathsf{c}_1 \ldots \mathsf{c}_n$"	[A literal string of n characters]
STRING	[The set of all strings]
$d_1 \ldots d_n \quad d_1 \ldots d_n.d_{n+1} \ldots d_m$	[Numbers]

Miscellaneous Constructs

IF p THEN e_1 ELSE e_2	[e_1 if p true, else e_2]
CASE $p_1 \rightarrow e_1 \,\Box\, \ldots \,\Box\, p_n \rightarrow e_n$	[Some e_i such that p_i true]
CASE $p_1 \rightarrow e_1 \,\Box\, \ldots \,\Box\, p_n \rightarrow e_n \,\Box$ OTHER $\rightarrow e$	[Some e_i such that p_i true, or e if all p_i are false]

LET $d_1 \triangleq e_1 \ \ldots \ d_n \triangleq e_n$ IN e [e in the context of the definitions]

$\land\ p_1$ [the conjunction $p_1 \land \ldots \land p_n$] $\quad \lor\ p_1$ [the disjunction $p_1 \lor \ldots \lor p_n$]

$\vdots \qquad\qquad \vdots$

$\land\ p_n \qquad\qquad \lor\ p_n$

(1) $x \in S$ may be replaced by a comma-separated list of items $v \in S$, where v is either a comma-separated list or tuple of identifiers.

(2) x may be an identifier or tuple of identifiers.

(3) We describe only functions of a single argument; TLA$^+$ also allows functions with multiple arguments.

(4) $![e_1]$ or $!.h$ may be replaced by a comma separated list of items $!a_1 \cdots a_n$, where each a_i is $[e_i]$ or $.h_i$.

Figure 12: The constant operators.

Predicate and Action Operators

e'	[The value of e in the final state of a step]
$[A]_e$	$[A \lor (e' = e)]$
$\langle A \rangle_e$	$[A \land (e' \neq e)]$
ENABLED A	[An A step is possible]
UNCHANGED e	$[e' = e]$
$A \cdot B$	[Composition of actions]

Temporal Operators

$\Box F$	[F is always true]
$\Diamond F$	[F is eventually true]
$\mathrm{WF}_e(A)$	[Weak fairness for action A]
$\mathrm{SF}_e(A)$	[Strong fairness for action A]
$F \leadsto G$	[F leads to G]
$F \stackrel{+}{\rhd} G$	[F guarantees G (see [1])]
$\boldsymbol{\exists}\, x : F$	[Temporal existential quantification (hiding).]
$\boldsymbol{\forall}\, x : F$	[Temporal universal quantification.]

Figure 13: The nonconstant operators of TLA$^+$.

Infix Operators

$+$ (1)	$-$ (1)	$*$ (1)	$/$ (2)	$\circ$ (3)	$++$
$\div$ (1)	% (1)	^ (1,4)	$..$ (1)	$...$	$--$
$\oplus$ (5)	$\ominus$ (5)	$\otimes$	$\oslash$	$\odot$	$**$
$<$ (1)	$>$ (1)	$\leq$ (1)	$\geq$ (1)	$\sqcap$	$//$
$\prec$	$\succ$	$\preceq$	$\succeq$	$\sqcup$	^^
$\ll$	$\gg$	$<:$	$:>$	&	&&
$\sqsubset$	$\sqsupset$	$\sqsubseteq$ (5)	$\sqsupseteq$	$\mid$	$\Vert$
$\subset$	$\supset$		$\supseteq$	$\star$	%%
$\vdash$	$\dashv$	$\models$	$=\mid$	$\bullet$	##
$\sim$	$\simeq$	$\approx$	$\cong$	$	$$
$:=$	$::=$	$\asymp$	$\doteq$	?	??
$\propto$	$\wr$	$\uplus$	$\bigcirc$	!!	@@

Postfix Operators (6)

^+ ^* ^#

Prefix Operator

$-$

(1) Defined by the *Naturals*, *Integers*, and *Reals* modules.
(2) Defined by the *Reals* module.
(3) x^y is printed as x^y.
(4) Defined by the *Sequences* module.
(5) Defined by the *Bags* module.
(6) e^+ is printed as e^+, and similarly for ^* and ^#.

Figure 14: User-definable operator symbols.

$\land$	`/\` or `\land`	$\lor$	`\/` or `\lor`	$\Rightarrow$	`=>`
$\neg$	`~` or `\lnot` or `\neg`	$\equiv$	`<=>` or `\equiv`	$\triangleq$	`==`
$\in$	`\in`	$\neq$	`#` or `/=`	$'$	`'`
$\langle$	`<<`	$\rangle$	`>>`	$\Box$	`[]`
$<$	`<`	$>$	`>`	$\Diamond$	`<>`
$\leq$	`\leq` or `=<`	$\geq$	`\geq` or `>=`	$\leadsto$	`~>`
$\ll$	`\ll`	$\gg$	`\gg`	$\stackrel{+}{\triangleright}$	`-+->`
$\prec$	`\prec`	$\succ$	`\succ`	$\mapsto$	`|->`
$\preceq$	`\preceq`	$\succeq$	`\succeq`	$\div$	`\div`
$\subseteq$	`\subseteq`	$\supseteq$	`\supseteq`	$\wr$	`\wr`
$\subset$	`\subset`	$\supset$	`\supset`	$\bullet$	`\bullet`
$\sqsubset$	`\sqsubset`	$\sqsupset$	`\sqsupset`	$\star$	`\star`
$\sqsubseteq$	`\sqsubseteq`	$\sqsupseteq$	`\sqsupseteq`	$\sim$	`\sim`
$\vdash$	`|-`	$\dashv$	`-|`	$\simeq$	`\simeq`
$\models$	`|=`	$\Dashv$	`=|`	$\asymp$	`\asymp`
$\rightarrow$	`->`	$\leftarrow$	`<-`	$\approx$	`\approx`
$\cap$	`\cap` or `\intersect`	$\cup$	`\cup` or `\union`	$\cong$	`\cong`
$\sqcap$	`\sqcap`	$\sqcup$	`\sqcup`	$\doteq$	`\doteq`
$\oplus$	`(+)` or `\oplus`	$\uplus$	`\uplus`	$\propto$	`\propto`
$\ominus$	`(-)` or `\ominus`	$\times$	`\X` or `\times`	x^y	`x^y` (2)
$\odot$	`(.)` or `\odot`	$\cdot$	`\cdot`	x^+	`x^+` (2)
$\otimes$	`\otimes`	$\circ$	`\o` or `\circ`	x^*	`x^*` (2)
$\oslash$	`(/)` or `\oslash`	“s”	`"s"` (1)	$x^\#$	`x^#` (2)
	`--------` (3)		`--------` (3)		
	`--------` (3)		`========` (3)		

(1) x and y are any expressions.
(2) s is any sequence of characters, not including `"`.
(3) a sequence of four or more `-` or `=` characters.

Figure 15: The ASCII representations of typeset symbols.