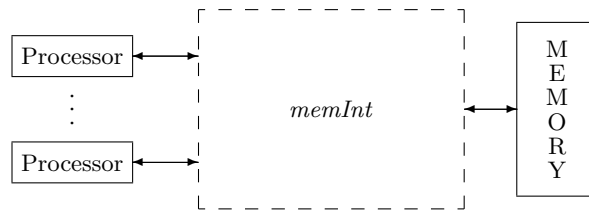


Chapter 5

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.

5.1 The Memory Interface

The asynchronous interface described in Chapter 3 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 4.4, 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.¹ 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

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.² We can assert that the value is a Boolean by the assumption

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*(...) is a Boolean is then superfluous as an explanation. But it's a good idea to include it anyway.

¹Even 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.

²We 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*.

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.

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 only 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 1.2 on page 11.)

The memory responds to a read request with the memory value it read. We will also have it 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 $\text{CHOOSE } x : F$ 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



MODULE <i>MemoryInterface</i>	
VARIABLE <i>memInt</i>	
CONSTANTS <i>Send</i> ($-, -, -, -$),	A <i>Send</i> ($p, d, memInt, memInt'$) step represents processor p sending value d to the memory.
<i>Reply</i> ($-, -, -, -$),	A <i>Reply</i> ($p, d, memInt, memInt'$) step represents the memory sending value d to processor p .
<i>InitMemInt</i> ,	The set of possible initial values of <i>memInt</i> .
<i>Proc</i> ,	The set of processor identifiers.
<i>Adr</i> ,	The set of memory addresses.
<i>Val</i>	The set of memory values.
ASSUME $\forall p, d, miOld, miNew : \wedge Send(p, d, miOld, miNew) \in \text{BOOLEAN}$ $\wedge Reply(p, d, miOld, miNew) \in \text{BOOLEAN}$	
$MReq \triangleq [op : \{\text{"Rd"}\}, adr : Adr] \cup [op : \{\text{"Wr"}\}, adr : Adr, val : Val]$	
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 <i>Val</i> .

Figure 5.1: The specification of a memory interface.

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 6.6 on page 73.

The complete memory interface specification is module *MemoryInterface* in Figure 5.1 on this page.

5.2 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 $\text{DOMAIN } f$, 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 values of the form $f[x]$ with x in $\text{DOMAIN } f$. 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 $x \in S$.³ 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 $\{\text{"val"}, \text{"ack"}, \text{"rdy"}\}$ 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

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

The EXCEPT construct for records, explained in Section 3.2, 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,

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

In general,

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

³The \in in $[x \in S \mapsto e]$ is just part of the syntax; TLA⁺ uses that particular symbol to help you remember what the construct means. Computer scientists write $\lambda x : S. e$ to represent something similar to $[x \in S \mapsto e]$, except that their λ expressions aren't quite the same as the functions of ordinary mathematics that are used in TLA⁺.

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

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

Functions correspond to the arrays of programming languages. The domain of a function corresponds to the index set of an array. 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 $[f \text{ EXCEPT } ![c_1] \dots [c_n] = e]$ 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 $\sigma["c"]$. For example, this implies

$$\begin{aligned} [f \text{ EXCEPT } ![c].d = e] &= [f \text{ EXCEPT } ![c][\text{"d"}] = e] \\ &= [f \text{ EXCEPT } ![c] = [\text{@ EXCEPT } !.d = e]] \end{aligned}$$



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

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

So far, we have seen only functions of a single argument, which are the mathematical analog of the one-dimensional arrays of programming languages. Mathematicians also use functions of multiple arguments, which are the analog of multi-dimensional arrays. In TLA⁺, as in ordinary mathematics, a function of multiple arguments is one whose domain is a set of tuples. For example, $f[5, 3, 1]$ is an abbreviation for $f[\langle 5, 3, 1 \rangle]$, the value of the function f applied to the triple $\langle 5, 3, 1 \rangle$.

The function constructs of TLA⁺ have extensions for functions of multiple arguments. For example, $[g \text{ EXCEPT } ![a, b] = e]$ is the function \hat{g} that is the same as g except with $\hat{g}[a, b]$ equal to e . The expression

$$(5.1) \quad [n \in \text{Nat}, r \in \text{Real} \mapsto n * r]$$

equals the function f such that $f[n, r]$ equals $n * r$, for all $n \in \text{Nat}$ and $r \in \text{Real}$. Just as $\forall i \in S : \forall j \in S : P$ can be written as $\forall i, j \in S : P$, we can write the function $[i \in S, j \in S \mapsto e]$ as $[i, j \in S \mapsto e]$.

Section 16.1.7 on page 301 describes the general versions of the TLA⁺ function constructs for functions with any number of arguments. However, functions of a single argument are all you're likely to need. You can almost always replace a function of multiple arguments with a function-valued function—for example, writing $f[a][b]$ instead of $f[a, b]$.

5.3 A Linearizable Memory

We now 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. Less restrictive, more practical memory specifications are described in Section 11.2.

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:

$$\begin{array}{c} \left[\begin{array}{l} ctl[p] = \text{"rdy"} \\ buf[p] = \dots \\ mem[a] = \dots \end{array} \right] \xrightarrow{Req(p)} \left[\begin{array}{l} ctl[p] = \text{"busy"} \\ buf[p] = req \\ mem[a] = \dots \end{array} \right] \\ \xrightarrow{Do(p)} \left[\begin{array}{l} ctl[p] = \text{"done"} \\ buf[p] = NoVal \\ mem[a] = v \end{array} \right] \xrightarrow{Rsp(p)} \left[\begin{array}{l} ctl[p] = \text{"rdy"} \\ buf[p] = NoVal \\ mem[a] = v \end{array} \right] \end{array}$$

A $Req(p)$ step represents the issuing of a request by processor p . It is enabled when $ctl[p] = \text{"rdy"}$; 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 $ctl[p] = \text{"busy"}$ 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 $ctl[p] = \text{"done"}$ 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* on the following two pages. The memory specification, which hides the three internal variables, is module *Memory* in Figure 5.3 on page 53.

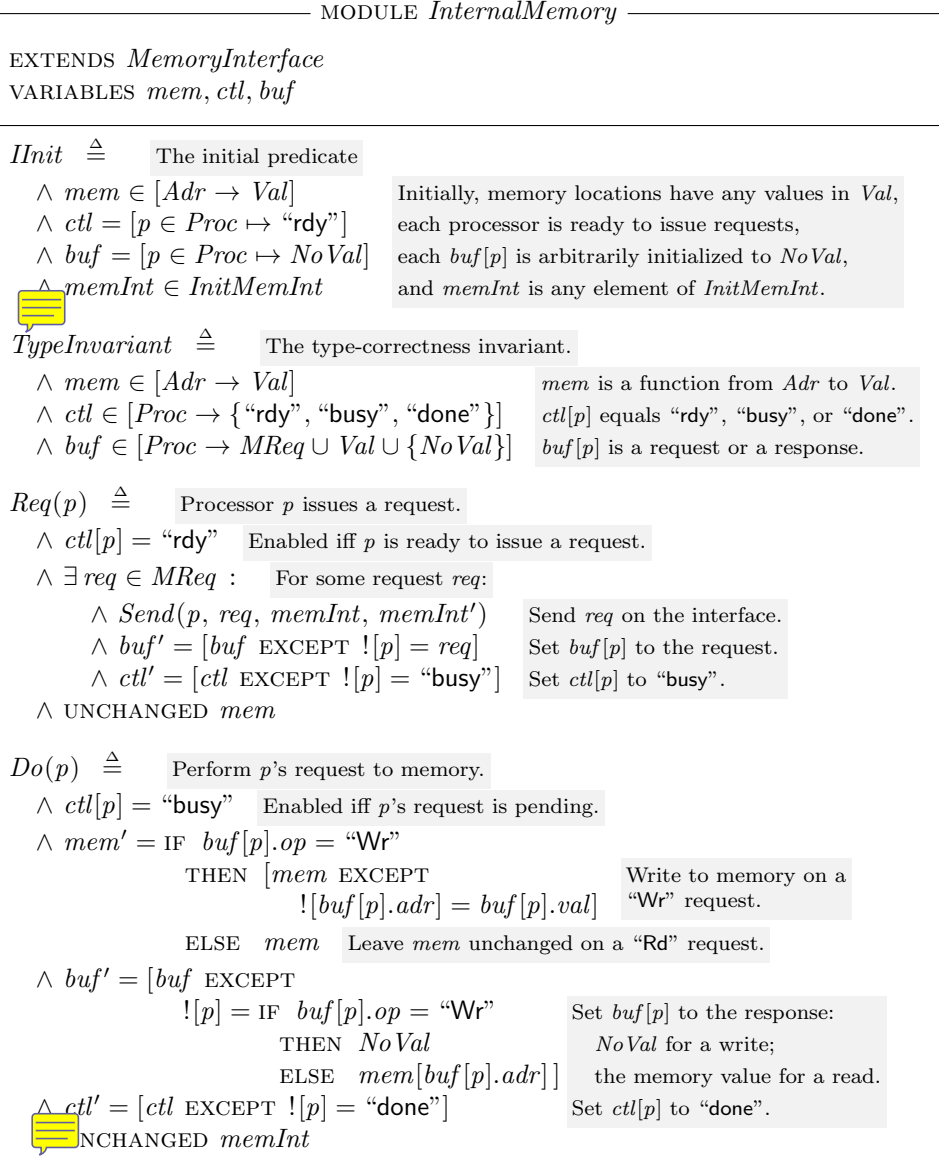


Figure 5.2a: The internal memory specification (beginning).

$Rsp(p) \triangleq$	Return the response to p 's request.
$\wedge ctl[p] = \text{"done"}$	Enabled iff req. is done but resp. not sent.
$\wedge Reply(p, buf[p], memInt, memInt')$	Send the response on the interface.
$\wedge ctl' = [ctl \text{ EXCEPT } ![p] = \text{"rdy"}]$	Set $ctl[p]$ to "rdy".
$\wedge \text{UNCHANGED } \langle mem, buf \rangle$	
$INext \triangleq$	$\exists p \in Proc : Req(p) \vee Do(p) \vee Rsp(p)$ The next-state action.
$ISpec \triangleq$	$IInit \wedge \Box [INext]_{\langle memInt, mem, ctl, buf \rangle}$ The specification.
THEOREM $ISpec \Rightarrow \Box TypeInvariant$	

Figure 5.2b: The internal memory specification (end).

5.4 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 forms $\langle a, p \rangle$ 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 $1 \dots n$. In fact, s is a sequence iff it equals $[i \in 1 \dots Len(s) \mapsto s[i]]$. Below are a few operator definitions from the *Sequences* module. (The meanings of the operators are described in Section 4.1.)

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

MODULE <i>Memory</i>	
EXTENDS <i>MemoryInterface</i>	
$Inner(mem, ctl, buf) \triangleq$	INSTANCE <i>InternalMemory</i>
$Spec \triangleq$	$\exists mem, ctl, buf : Inner(mem, ctl, buf) ! ISpec$



Figure 5.3: The memory specification.

5.5 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*.

The function definition notation has a straightforward generalization to definitions of functions of multiple arguments. For example,

$$\begin{aligned} Acker[m, n \in Nat] &\triangleq \\ &\text{IF } m = 0 \text{ THEN } n + 1 \\ &\quad \text{ELSE IF } n = 0 \text{ THEN } Acker[m - 1, 0] \\ &\quad \text{ELSE } Acker[m - 1, Acker[m, n - 1]] \end{aligned}$$

defines *Acker*[*m*, *n*] for all natural numbers *m* and *n*.

Section 6.3 explains exactly what recursive definitions mean. For now, we will just write recursive definitions without worrying about their meaning.

5.6 Write-Through Cache

We now specify a simple write-through cache that implements the memory specification. The system is described by the picture of Figure 5.4 on the next page. 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 (module *InternalMemory*). (However, as we will see below, *ctl*[*p*] can assume an additional value “waiting”.) These local controllers

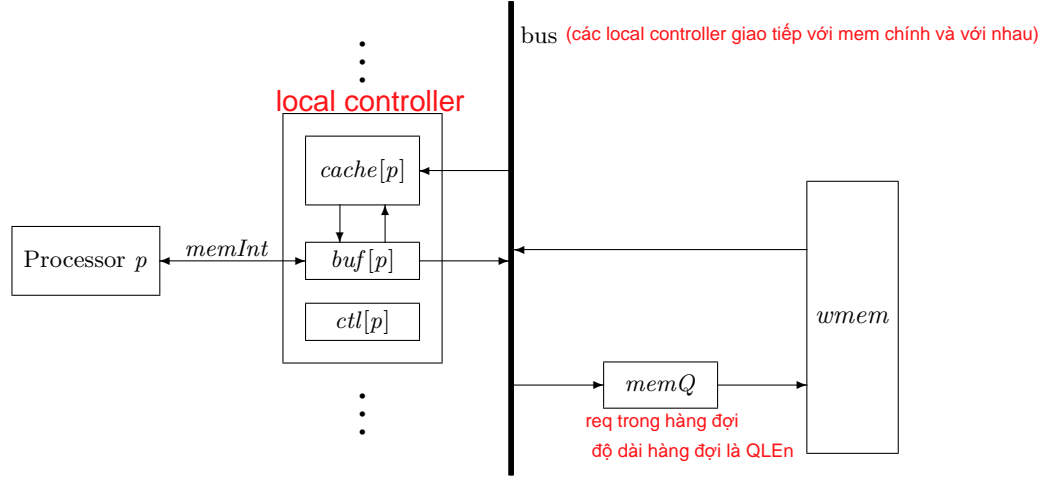


Figure 5.4: The write-through cache.


communicate with the main memory $wmem$,⁴ 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 $memQ$. When the request reaches the head of $memQ$, the action $MemQWr$ stores the value in $wmem$. The $DoWr(p)$ action also updates $cache[q]$ for every other processor q that has a copy of the address in its cache.

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

⁴We use the name $wmem$ to distinguish this variable from variable mem of module *InternalMemory*. We don’t have to, since mem is not a free (visible) variable of the actual memory specification in module *Memory*, but it helps us avoid getting confused.

 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 in an implementation, 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, so it doesn't appear in the specification. 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 a read request whose $DoRd(p)$ action has not yet been performed. This is the case when $ctl[p]$ equals “waiting” and $buf[p].adr$ equals the cached address.


The actions $Req(p)$ and $Rsp(p)$, 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, and they leave unchanged $vmem$ instead of mem .

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* on pages 57–59. 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 reuse some of the definitions from the *InternalMemory* module, so an `INSTANCE` statement instantiates a copy of that module with $wmem$ substituted for mem . (The other parameters of module *InternalMemory* are instantiated by the parameters of the same name in module *WriteThroughCache*.)

The initial predicate *Init* contains the conjunct $M!Init$, which asserts that ctl and buf have the same initial values as in the internal memory specification, and that $wmem$ has the same initial value as mem does in that 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 reuse the internal memory's type invariant $M!TypeInvariant$. Formula *TypeInvariant* therefore explicitly describes the types of $wmem$, ctl , and buf . The type of $memQ$ is the set of sequences of $\langle \text{processor}, \text{request} \rangle$ 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 both have copies of address a in their caches, then those copies are equal. Note the trick of writing $x \notin \{y, z\}$ instead of the equivalent but longer formula $(x \neq y) \wedge (x \neq z)$. 

MODULE <i>WriteThroughCache</i>	
EXTENDS <i>Naturals</i> , <i>Sequences</i> , <i>MemoryInterface</i>	
VARIABLES <i>wmem</i> , <i>ctl</i> , <i>buf</i> , <i>cache</i> , <i>memQ</i>	
CONSTANT <i>QLen</i>	
ASSUME $(QLen \in Nat) \wedge (QLen > 0)$	
$M \triangleq$ INSTANCE <i>InternalMemory</i> WITH $mem \leftarrow wmem$	
<i>Init</i> \triangleq	The initial predicate
\wedge <i>M!Init</i>	<i>wmem</i> , <i>buf</i> , and <i>ctl</i> are initialized as in the internal memory spec.
\wedge <i>cache</i> =	All caches are initially empty ($cache[p][a] = NoVal$ for all p, a).
	$[p \in Proc \mapsto [a \in Adr \mapsto NoVal]]$
\wedge <i>memQ</i> = $\langle \rangle$	The queue <i>memQ</i> is initially empty.
<i>TypeInvariant</i> \triangleq	The type invariant.
\wedge <i>wmem</i> $\in [Adr \rightarrow Val]$	
\wedge <i>ctl</i> $\in [Proc \rightarrow \{\text{"rdy"}, \text{"busy"}, \text{"waiting"}, \text{"done"}\}]$	
\wedge <i>buf</i> $\in [Proc \rightarrow MReq \cup Val \cup \{NoVal\}]$	
\wedge <i>cache</i> $\in [Proc \rightarrow [Adr \rightarrow Val \cup \{NoVal\}]]$	
\wedge <i>memQ</i> $\in Seq(Proc \times MReq)$	<i>memQ</i> is a sequence of $\langle proc., request \rangle$ pairs.
<i>Coherence</i> \triangleq	Asserts that if two processors' caches both have copies of an address, then those copies have equal values.
$\forall p, q \in Proc, a \in Adr :$	$(NoVal \notin \{cache[p][a], cache[q][a]\}) \Rightarrow (cache[p][a] = cache[q][a])$
<i>Req</i> (<i>p</i>) \triangleq	Processor <i>p</i> issues a request.
$M!Req(p) \wedge \text{UNCHANGED } \langle cache, memQ \rangle$	
<i>Rsp</i> (<i>p</i>) \triangleq	The system issues a response to processor <i>p</i> .
$M!Rsp(p) \wedge \text{UNCHANGED } \langle cache, memQ \rangle$	
<i>RdMiss</i> (<i>p</i>) \triangleq	Enqueue a request to write value from memory to <i>p</i> 's cache.
$\wedge (ctl[p] = \text{"busy"}) \wedge (buf[p].op = \text{"Rd"})$	Enabled on a read request when the address is not in <i>p</i> 's cache and <i>memQ</i> is not full.
$\wedge cache[p][buf[p].adr] = NoVal$	
$\wedge Len(memQ) < QLen$	Append $\langle p, request \rangle$ to <i>memQ</i> . Set <i>ctl</i> [<i>p</i>] to "waiting".
$\wedge memQ' = Append(memQ, \langle p, buf[p] \rangle)$	
$\wedge ctl' = [ctl \text{ EXCEPT } !p] = \text{"waiting"}$	
$\wedge \text{UNCHANGED } \langle memInt, wmem, buf, cache \rangle$	

Figure 5.5a: The write-through cache specification (beginning).

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

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

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

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

Figure 5.5b: The write-through cache specification (middle).

$MemQRd$	\triangleq	Perform an enqueued read to memory.
LET p	\triangleq	$Head(memQ)[1]$ The requesting processor.
r	\triangleq	$Head(memQ)[2]$ The request at the head of $memQ$.
IN $\wedge (memQ \neq \langle \rangle) \wedge (r.op = \text{"Rd"})$		Enabled if $Head(memQ)$ is a read.
$\wedge memQ' = Tail(memQ)$		Remove the head of $memQ$.
$\wedge cache' =$		Put value from memory or $memQ$ in p 's cache.
		$[cache \text{ EXCEPT } ![p][r.adr] = vmem[r.adr]]$
$\wedge \text{UNCHANGED } \langle memInt, wmem, buf, ctl \rangle$		
$Evict(p, a)$	\triangleq	Remove address a from p 's cache.
$\wedge (ctl[p] = \text{"waiting"}) \Rightarrow (buf[p].adr \neq a)$		Can't evict a if it was just read into cache from memory.
$\wedge cache' = [cache \text{ EXCEPT } ![p][a] = NoVal]$		
$\wedge \text{UNCHANGED } \langle memInt, wmem, buf, ctl, memQ \rangle$		
$Next$	\triangleq	$\vee \exists p \in Proc : \vee Req(p) \vee Rsp(p)$ $\vee RdMiss(p) \vee DoRd(p) \vee DoWr(p)$ $\vee \exists a \in Adr : Evict(p, a)$ $\vee MemQWr \vee MemQRd$
$Spec$	\triangleq	$Init \wedge \Box[Next]_{\langle memInt, wmem, buf, ctl, cache, memQ \rangle}$
THEOREM $Spec \Rightarrow \Box(TypeInvariant \wedge Coherence)$		
LM	\triangleq	INSTANCE $Memory$ The memory spec. with internal variables hidden.
THEOREM $Spec \Rightarrow LM!Spec$ Formula $Spec$ implements the memory spec.		

Figure 5.5c: The write-through cache specification (end).

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*. However, they must also 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 $memQ$.

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

The definition of *DoWr* introduces the TLA⁺ **LET/IN** construct. The LET clause consists of a sequence of definitions whose scope extends until the end of the IN clause. 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.

Next comes the definition of the state function *vmem*, which is used in defining action *MemQRd* below. It equals the value that the main memory *wmem* 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 function *vmem* is defined in terms of the recursively defined function f , where $f[i]$ is the value *wmem* 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 i^{th} element in the sequence *memQ*.

The next two actions, *MemQWr* and *MemQRd*, 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.

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 $ctl[p] = \text{“waiting”}$ 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 next.

5.7 Invariance

Module *WriteThroughCache* contains the theorem

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

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

THEOREM $Spec \Rightarrow \Box TypeInvariant$

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

THEOREM $TypeInvariant \wedge Next \Rightarrow TypeInvariant'$

(Recall that *TypeInvariant'* is the formula obtained by priming all the variables in formula *TypeInvariant*.) In general, when $P \wedge N \Rightarrow P'$ holds, we say that predicate P is an invariant of action N . 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*. For example, suppose s is a state in which

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

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

for two different processors $p1$ and $p2$ and some address a . Such a state s (an assignment of values to variables) exists, assuming that there are at least two processors and at least one address. 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 $cache[p2][a] = 2$ and $cache[p1][a] = 1$, 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

$$Init \wedge \Box[Next]_v \Rightarrow \Box P$$

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

$$Init \Rightarrow Inv, \quad Inv \wedge [Next]_v \Rightarrow Inv', \quad Inv \Rightarrow P$$

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

5.8 Proving Implementation

Module *WriteThroughCache* ends with the theorem



THEOREM $Spec \Rightarrow LM!Spec$

where $LM!Spec$ is formula *Spec* of module *Memory*. This theorem asserts that every behavior satisfying specification *Spec* of the write-through cache also satisfies $LM!Spec$, the specification of a linearizable memory. In other words, it asserts that the write-through cache implements a linearizable memory. In TLA, implementation is implication. A system described by a formula *Sys* implements a specification *Spec* iff *Sys* implies *Spec*—that is, iff $Sys \Rightarrow Spec$ is a theorem. TLA makes no distinction between system descriptions and specifications; they are both just formulas.

By definition of formula *Spec* of the *Memory* module (page 53), we can restate the theorem as

THEOREM $Spec \Rightarrow \exists mem, ctl, buf : LM!Inner(mem, ctl, buf)!ISpec$

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

state functions (ordinary expressions with no primes), which I'll call $omem$, $octl$, and $obuf$, that satisfy

$$(5.2) \quad Spec \Rightarrow LM!Inner(omem, octl, obuf)!ISpec$$

Formula $LM!Inner(omem, octl, obuf)!ISpec$ is formula $ISpec$ with the substitutions

$$mem \leftarrow omem, \quad ctl \leftarrow octl, \quad buf \leftarrow obuf$$

The tuple $\langle omem, octl, obuf \rangle$ of witness functions is called a *refinement mapping*, and we describe (5.2) as the assertion that $Spec$ implements formula $ISpec$ under this refinement mapping. Intuitively, this means $Spec$ implies that the value of the tuple $\langle memInt, omem, octl, obuf \rangle$ of state functions changes the way $ISpec$ asserts that the tuple $\langle memInt, mem, ctl, buf \rangle$ of variables should change.

I will now briefly describe how we prove (5.2); for details, see the technical papers about TLA, available through the TLA Web page. Let me first introduce a bit of non-TLA⁺ notation. For any formula F of module *InternalMemory*, 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.

With this notation, we can write (5.2) as $Spec \Rightarrow \overline{ISpec}$. Replacing $Spec$ and $ISpec$ by their definitions, this formula becomes

$$(5.3) \quad \begin{aligned} Init \wedge \Box[Next]_{\langle memInt, wmem, buf, ctl, cache, memQ \rangle} \\ \Rightarrow \overline{Init} \wedge \Box[\overline{Next}]_{\langle memInt, \overline{mem}, \overline{ctl}, \overline{buf} \rangle} \end{aligned}$$

\overline{memInt} equals $memInt$, since $memInt$ is a variable distinct from mem , ctl , and buf .

Formula (5.3) is then proved by finding an invariant Inv of $Spec$ such that

$$\begin{aligned} \wedge Init &\Rightarrow \overline{Init} \\ \wedge Inv \wedge Next &\Rightarrow \vee \overline{Next} \\ &\vee \text{UNCHANGED } \langle memInt, \overline{mem}, \overline{ctl}, \overline{buf} \rangle \end{aligned}$$

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{Next} step—a step that changes the 4-tuple $\langle memInt, omem, octl, obuf \rangle$ the way an \overline{Next} step changes $\langle memInt, mem, ctl, buf \rangle$ —or else it leaves that 4-tuple unchanged. For our memory specifications, the state functions $omem$, $octl$, and $obuf$ are defined by

$$\begin{aligned} omem &\triangleq wmem \\ octl &\triangleq [p \in Proc \mapsto \text{IF } ctl[p] = \text{“waiting” THEN “busy” ELSE } ctl[p]] \\ obuf &\triangleq buf \end{aligned}$$

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 theory to practice requires turning the mathematics

of proofs into an engineering discipline. This is a subject that deserves a book to itself, and I won't try to discuss it here.

You will probably never prove that one specification implements another. However, you should understand refinement mappings and step simulation. You will then be able to use TLC to check that one specification implements another; Chapter 14 explains how.