Hiding, Refinement, and Auxiliary Variables

Leslie Lamport

27 June 2019
Corrected: 2 March 2020, 18 May 2020

This note is a version of the folk tale of a stone soup. A stone soup is made by boiling a stone in a cauldron of water and gradually adding vegetables and meat. At the end, one eats everything but the stone.

The stone in the TLA$^+$ soup is the temporal existential quantifier $\exists$. This operator was an important part of the original TLA logic. As other ingredients were added (the "+" in TLA$^+$) and the soup was consumed (by engineers), I realized that $\exists$ provided no nourishment and should be left behind. But like the stone in the folk tales, $\exists$ serves a purpose. It is used here to help explain the meaning of refinement.

Text colored like this in the table of contents or like this elsewhere is a clickable link.

Contents

1 Hiding

Consider a specification of a FIFO (first-in-first-out) queue of natural numbers. There are two operations that can be performed to the queue: *Put* appends a number to the end of the queue, and *Get* removes a number from the front of the queue. For simplicity, let's assume an interface in which performing a command is described by setting the value of a variable *op*. Performing a *Put* operation that appends the value 7 to the end of the queue is described by setting *op* to the pair $\langle \text{“put”}, 7 \rangle$; and executing a *Get* that removes the value 42 from the front of the queue is described by setting *op* to $\langle \text{“get”}, 42 \rangle$. A TLA$^+$ spec of this queue is formula *Spec* in Module *FIFO* of Figure 1, where the variable *queue* describes the current state of the queue and *op* initially equals $\langle \text{“init”}, 0 \rangle$. No liveness properties are specified in this module or in any of our specifications. Adding liveness doesn't change anything discussed in this note. (Sophisticated TLA$^+$ users may realize that there's a possible problem with this spec; it will be corrected later.)

In the early days of specifying concurrent systems, some researchers argued that only the sequence of operations that are performed is observable; the queue itself is an internal implementation detail that should not appear

MODULE *FIFO*

EXTENDS *Integers, Sequences*

VARIABLES *op, queue*

$$
\begin{aligned}
&\mathit{vars} \triangleq \langle \mathit{op}, \mathit{queue} \rangle \\
&\mathit{Init} \triangleq \land\ \mathit{op} = \langle \text{“init”}, 0 \rangle \\
&\qquad\quad \land\ \mathit{queue} = \langle\,\rangle \\
&\mathit{Put} \triangleq \exists\, v \in \mathit{Nat} : \land\ \mathit{op}' = \langle \text{“put”}, v \rangle \\
&\qquad\qquad\qquad\quad \land\ \mathit{queue}' = \mathit{Append}(\mathit{queue}, v) \\
&\mathit{Get} \triangleq \land\ \mathit{queue} \neq \langle\,\rangle \\
&\qquad\quad \land\ \mathit{op}' = \langle \text{“get”}, \mathit{Head}(\mathit{queue}) \rangle \\
&\qquad\quad \land\ \mathit{queue}' = \mathit{Tail}(\mathit{queue}) \\
&\mathit{Next} \triangleq \mathit{Put} \lor \mathit{Get} \\
&\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}
\end{aligned}
$$

Figure 1: The FIFO Queue Specification.

in the spec. Thus, a spec should contain only the variable *op*; the variable *queue* is just one way of implementing those operations and should not appear in the spec. I was not impressed by this argument for three reasons:

1. In practice, it is impossible to write specs that don't in some way mention such "unobservable" state as the queue.

2. If you want, you can make the variable *queue* disappear from formula *Spec* by simply writing $\exists\, queue : Spec$.

3. Any implementation of the spec must implement *queue* in its state, and there's a simple, precise definition of what that means.

To understand reason 1, try writing a specification of the FIFO queue in English without mentioning the state of the queue. Now imagine trying to specify a complicated distributed system by mentioning only what's visible on users' screens. I have seen that trying to specify even simple systems in terms only of an externally observable interface, with nothing like an internal state, doesn't work in practice.

Reasons 2 and 3 are the topics of Sections 2 and 3, which explain and discuss at length the $\exists$ operator. Section 4 explains auxiliary variables, which are sometimes needed to show that one spec implements another. Finally, Section 5 explains why the operator $\exists$ is of no practical importance, but the concepts introduced in Sections 2–4 are useful in practice.

2 Temporal Existential Quantification

The Operator $\exists$

The operator $\exists$ of ordinary math (written `\E` in TLA$^+$) is existential quantification over a constant. For example, the formula

(1) $\quad \exists\, x \in Real : a * x^2 + b * x + c = 0$

asserts that there is some real number that can be assigned to x that makes the formula $a * x^2 + b * x + c = 0$ true.

The operator $\exists$ (written `\EE` in TLA$^+$) is existential quantification over a variable. Think of the temporal formula $\exists\, queue : Spec$ as being true of (satisfied by) a behavior iff (if and only if) there is some infinite sequence of values that can be assigned to the variable *queue*, one value for each state of the behavior, that makes the formula *Spec* true for that behavior. A more precise definition of $\exists$ is given below.

Formula (1) is an assertion about the values of a, b, and c. It happens to be true if the values of those three variables are real numbers and $b^2 - 4 * a * c > 0$ is true. The formula says nothing about the value of x. Formula (1) asserts that there is some real number x that makes $a * x^2 + b * x + c = 0$ true, but it doesn't say anything about what the actual value of x is. We obtain a completely equivalent formula by replacing each x with y.

Similarly, the formula $\exists\, queue : Spec$ asserts that there exists an infinite sequence of values for *queue* that would make *Spec* true. It says nothing about the actual value of the variable *queue* in any state. We'd get an equivalent formula by replacing *queue* with y in that formula and in the definition of *Spec*. We can say that the "$\exists\, queue$" hides the variable *queue*, effectively making it disappear.

Using $\exists$ in TLA$^+$

If *Spec* is defined as in module *FIFO*, the formula $\exists\, queue : Spec$ I've been writing cannot be a legal formula in any TLA$^+$ module. Since *Spec* is defined in terms of the variable *queue*, it can appear only in a context in which *queue* has been declared. In such a context, TLA$^+$ doesn't allow you to locally redefine *queue* by writing $\exists\, queue$. Instead of $\exists\, queue : Spec$, in TLA$^+$ we can write $\exists\, queue : FI(queue)!Spec$ in a module containing:

VARIABLE *op*
$FI(queue) \triangleq$ INSTANCE *FIFO*

However, for simplicity, I will simply write $\exists\, queue : Spec$ in this document.

The Precise Definition

The precise definition of the operator $\exists$ is a bit complicated because, for reasons explained on the TLA$^+$ Advanced Topics web page, TLA$^+$ formulas are stuttering insensitive. This means that whether a formula is true of a behavior isn't changed by adding and/or removing stuttering steps to/from the behavior, where a stuttering step is a pair of successive states that are identical. To ensure that the formula $\exists\, v : F$ is stuttering invariant, for any formula F, we define it to be true for a behavior b iff there is a behavior $\widehat{b}$ obtained by adding and/or removing stuttering steps from b such that some infinite sequence of values can be assigned to the variable *queue*, one value for each state of the behavior $\widehat{b}$, that makes the formula F true for $\widehat{b}$.

A Problem With Our Spec

There is a problem with letting the queue specification be $\exists\, queue : Spec$ (or its equivalent TLA$^+$ formula in another module). This formula specifies the allowed sequences of values of the single variable *op* in a behavior. As a TLA$^+$ formula, it's stuttering-insensitive. Thus, if it allows a behavior with a step in which the value of *op* changes from $\langle\, \text{“get”}, 7 \rangle$ to $\langle\, \text{“put”}, 2 \rangle$, then it also allows a behavior in which that step is replaced by a sequence of steps in which *op* has the following values:

$$op = \langle\, \text{“get”}, 7 \rangle \;\rightarrow\; op = \langle\, \text{“get”}, 7 \rangle \;\rightarrow\; op = \langle\, \text{“get”}, 7 \rangle \;\rightarrow$$
$$op = \langle\, \text{“get”}, 7 \rangle \;\rightarrow\; op = \langle\, \text{“put”}, 2 \rangle$$

That is, if a behavior allows a step in which a *Get* of 7 is performed, then it also allows one containing three additional such steps, without any additional *Put* step. There's no way to distinguish between a stuttering step and one in which the same operation is performed twice. (We could tell the difference in formula *Spec* because a stuttering step doesn't change the value of *queue*.)

I will not consider the philosophical question of whether this is a problem with formula *Spec*, since in that spec the variable *queue* distinguishes repeated operations and stuttering steps. The important observation is that, if we want the variable *op* by itself to describe the sequence of operations, a repeated operation must change its value. Our queue specification is a high-level abstraction of a real system composed of silicon and software. The user of the queue communicates with the queue through some interface. If the user performs two successive *Get* operations with the same value, something in that interface must have changed to indicate that two separate operations are being performed. I find the most natural way to represent that is to have every operation change the value of *op*; and the easiest way to modify formula *Spec* to do that is to make the value of *op* a triple whose third component alternates between two possible values. Letting those two values be 1 and -1, we can modify module *FIFO* to include the definitions of *Init*, *Put*, and *Get* shown in Figure 2.

3 Refinement with Hiding

Above, I made this assertion about the queue specification:

3. Any implementation of the spec must implement *queue* in its state, and there's a simple, precise definition of what that means.

To explain it, we will consider a TLA$^+$ specification that implements the queue specification. We often use the term *refinement* instead of *implementation* when a spec is being implemented by another spec rather than by executable code, but there is no logical difference between the two.

The spec that refines the queue specification is formula *Spec* of module *FIFO*2 in Figure 3. The queue described by variable *queue* has been replace by two queues described by variables qP and qG. The *Put* operation appends a number to the end of qP; the *Get* operation removes a number from the head of qG. A *Move* operation moves numbers from the head of qP to the end of qG.[1]

Let $Spec_1$ be formula *Spec* of module *FIFO* and let $Spec_2$ be formula *Spec* of *FIFO*2. It should be intuitively clear that $Spec_2$ implements $\exists\, queue : Spec_1$, because the sequence of values of the variable *op* in any behavior satisfying $Spec_2$ is the sequence of values of *op* in some behavior satisfying $\exists\, queue : Spec_1$. In other words, this should be a true theorem—that is, the formula following "THEOREM" is true for all behaviors:

$$(2) \quad \text{THEOREM} \;\; Spec_2 \;\Rightarrow\; \exists\, queue : Spec_1$$

Theorem (2) is an informal way of writing the theorem that can be written as follows in module *FIFO*2:

$$\begin{array}{l} FI(queue) \;\triangleq\; \text{INSTANCE } FIFO \\ \text{THEOREM} \;\; Spec \;\Rightarrow\; \exists\, queue \,:\, FI(queue)!Spec \end{array}$$

[1] If you think there's an error in the *Move* action, you're probably thinking of $qP' = Tail(qP)$ as an assignment statement rather than a subformula of a single mathematical formula—a subformula that could just as well be written $Tail(qP) = qP'$, except that TLC couldn't check the spec if the formula were written that way.

$$\begin{array}{lcl} Init & \triangleq & \land\; op = \langle \text{“init”},\, 0,\, 1 \rangle \\ & & \land\; queue = \langle\, \rangle \\ \\ Put & \triangleq & \exists\, v \in Nat \,:\, \land\; op' = \langle \text{“put”},\, v,\, -op[3] \rangle \\ & & \qquad\qquad \land\; queue' = Append(queue,\, v) \\ \\ Get & \triangleq & \land\; queue \neq \langle\, \rangle \\ & & \land\; op' = \langle \text{“get”},\, Head(queue),\, -op[3] \rangle \\ & & \land\; queue' = Tail(queue) \end{array}$$

Figure 2: The modified definitions for the FIFO Queue Specification.

MODULE *FIFO2*

EXTENDS *Integers, Sequences*

VARIABLES *op, qP, qG*

$$
\begin{array}{lcl}
vars & \triangleq & \langle op,\ qP,\ qG \rangle \\
\\
Init & \triangleq & \land\ op = \langle \text{“init”},\ 0,\ 1 \rangle \\
 & & \land\ qP = \langle\,\rangle \\
 & & \land\ qG = \langle\,\rangle \\
\\
Put & \triangleq & \exists\, v \in Nat\ :\ \land\ op' = \langle \text{“put”},\ v,\ -op[3] \rangle \\
 & & \qquad\qquad\ \land\ qP' = Append(qP,\ v) \\
 & & \qquad\qquad\ \land\ qG' = qG \\
\\
Get & \triangleq & \land\ qG \neq \langle\,\rangle \\
 & & \land\ op' = \langle \text{“get”},\ Head(qG),\ -op[3] \rangle \\
 & & \land\ qG' = Tail(qG) \\
 & & \land\ qP' = qP \\
\\
Move & \triangleq & \land\ qP \neq \langle\,\rangle \\
 & & \land\ qP' = Tail(qP) \\
 & & \land\ qG' = Append(qG,\ Head(qP)) \\
 & & \land\ op' = op \\
\\
Next & \triangleq & Put \lor Get \lor Move \\
\\
Spec & \triangleq & Init \land \Box[Next]_{vars}
\end{array}
$$

Figure 3: Another FIFO Queue Specification.

How would we prove such a theorem? Let's consider how we would prove formula (1) on page 2, assuming a, b, and c are real numbers satisfying $b^2 - 4 * a * c > 0$. To prove that formula, we must show that there exists some number x satisfying:

(3) $\quad a * x^2 + b * x + c = 0$

The obvious way to do that is to write down an expression that, when substituted for x, makes (3) true. If you remember your high-school algebra, you will see that this is one such expression—assuming a, b, and c are real

numbers with $b^2 - 4 * a * c > 0$:

$$\begin{array}{l} \text{IF } a = 0 \text{ THEN } (-c)/b \\ \qquad \text{ELSE } (-b + \sqrt{b^2 - 4 * a * c})/(2 * a) \end{array}$$

Similarly, to prove (2), we must show that there exists some expression such that, when substituted for *queue*, makes $Spec_1$ true—assuming $Spec_2$ is true. That is, we want this this theorem to be true for some expression *exp*:

(4) THEOREM $Spec_2 \Rightarrow (Spec_1 \text{ WITH } queue \leftarrow exp)$

where "$Spec_1$ WITH *queue* $\leftarrow$ *exp*" is the formula obtained by substituting *exp* for *queue* in formula $Spec_1$. (This is not a legal TLA$^+$ formula.)

When you understand formulas $Spec_1$ and $Spec_2$, the obvious choice for the expression *exp* is $qG \circ qP$, where $\circ$ is sequence concatenation. We can write the formula

(5) $Spec_1$ WITH *queue* $\leftarrow qG \circ qP$

in module *FIFO*2 as $F!Spec$ by adding this statement to the module:

$F \triangleq$ INSTANCE *FIFO* WITH *queue* $\leftarrow qG \circ qP$

We can then write (4) with *exp* equal to $qG \circ qP$ in module *FIFO*2 as

(6) THEOREM $Spec \Rightarrow F!Spec$

We can check theorem (6) in module *FIFO*2 by having TLC check the temporal property $F!Spec$ for a model with: the behavior spec set to the temporal formula *Spec*, the definition of *Nat* overridden to equal some finite set, and a state constraint bounding the lengths of the queues qP and qG.

Theorem (6) asserts that every behavior b that satisfies formula *Spec* also satisfies $F!Spec$. To understand the theorem, you have to understand what it means for a behavior b to satisfy $F!Spec$, which we can write informally as (5). The meaning is explained by this observation:

> For any temporal formula P of module *FIFO*, formula $F!P$ is true for a behavior b iff formula P is true for the behavior $\tilde{b}$ obtained from b by replacing the value of *queue* in each state of b by the value of $qG \circ qP$.

The following example shows why this is true. Suppose P equals $\Box(queue \in Seq(Nat))$, so $F!P$ equals $\Box(qG \circ qP \in Seq(Nat))$. In this case, $F!P$ is true of the behavior b iff $qG \circ qP \in Seq(Nat)$ is true in every state of b.

Formula $qG \circ qP \in Seq(Nat)$ is true in a state s iff $queue \in Seq(Nat)$ is true in the state obtained from s by replacing the value of $queue$ with the value of $qG \circ qP$. Hence, $qG \circ qP \in Seq(Nat)$ is true in every state of b iff $queue \in Seq(Nat)$ is true in every state of $\tilde{b}$, since the states of $\tilde{b}$ are obtained from the corresponding states of b by replacing the value of $queue$ with the value of $qG \circ qP$. Hence, for this formula P, the behavior b satisfies $F!P$ iff the behavior $\tilde{b}$ satisfies P. You should convince yourself that this is true for every formula P of module $FIFO$, including $Spec$.

Knowing what theorem (6) of module $FIFO2$ means, we can see why it's true from traces (a)–(d) of Figure 4.

(a) This trace is the beginning of a behavior b satisfying formula $Spec$ of $FIFO2$. It is an error trace obtained by having TLC check that a non-invariant formula is an invariant of $Spec$.

(b) This was obtained from (a) by running the Toolbox's Trace Explorer to show the value of $qG \circ qP$ in each state.

(c) This was obtained from (b) by assigning to the variable $queue$ the value of $qG \circ qP$ in the current state. It is the beginning of the behavior $\tilde{b}$ defined above.

(d) The variables qG and qB, which don't appear in $Spec_1$ (formula $Spec$ of module $FIFO$), have been removed. You can check that this trace is the beginning of a behavior that satisfies $Spec_1$. Observe that $Move$ steps allowed by $Spec_2$ have become stuttering steps allowed by $Spec_1$.

Let me recapitulate what we've done. We added this to module $FIFO2$:

(7) $F \triangleq$ INSTANCE $FIFO$ WITH $queue \leftarrow qG \circ qP$
THEOREM $Spec \Rightarrow F!Spec$

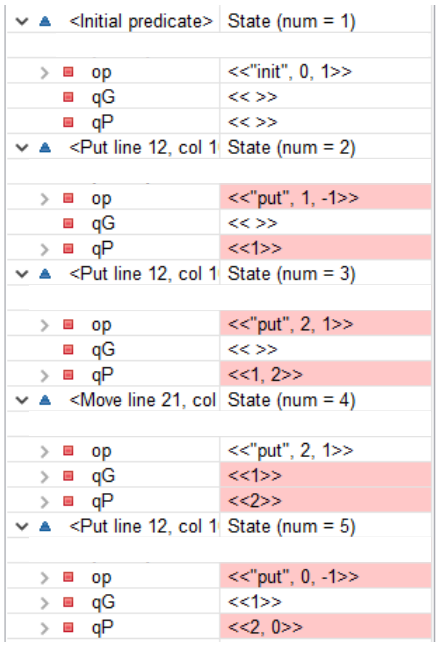
We saw that this theorem is correct, and that it implies the theorem written informally as:

(2) THEOREM $Spec_2 \Rightarrow \exists\, queue : Spec_1$

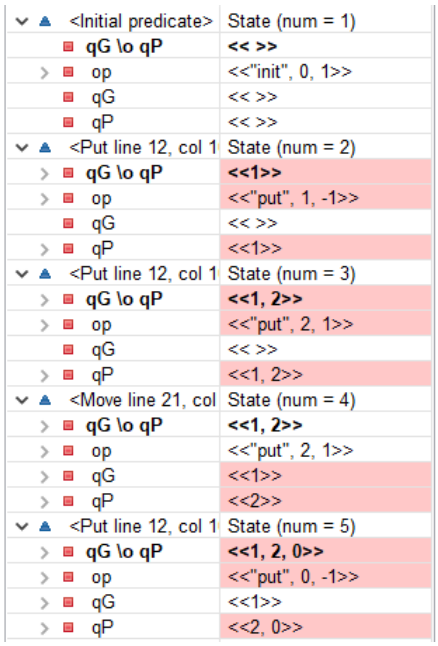
When the theorem of (7) is true, we say that $Spec_2$ implements (or refines) $Spec_1$ by implementing (or refining) $queue$ with $qG \circ qP$. The substitution $queue \leftarrow qG \circ qP$ is called a *refinement mapping*, and we say that (2) is proved with this refinement mapping.

In general, $Spec_1$ and $Spec_2$ are arbitrary specifications and (2) becomes

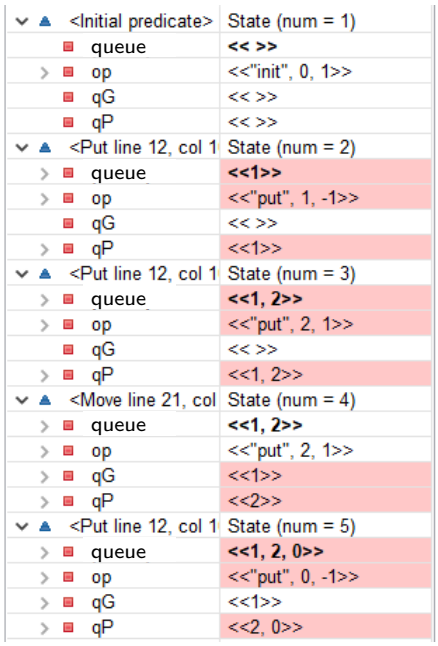
(8) THEOREM $Spec_2 \Rightarrow \exists\, v_1, \ldots, v_k : Spec_1$



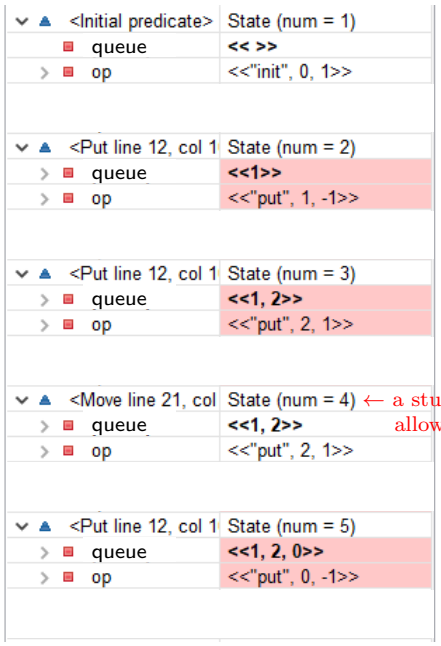
(a) A trace of $Spec_2$



(b) Adding values of $qG \circ qP$



(c) Assigning those values to *queue*



(d) Removing irrelevant variables

Figure 4: Obtaining a trace of $Spec_1$ implemented by a trace of $Spec_2$.

for variables v_i of $Spec_1$. We verify (8) by showing

(9) THEOREM $Spec_2 \Rightarrow (Spec_1 \text{ WITH } v_1 \leftarrow exp_1, \ldots, v_k \leftarrow exp_k)$

where the exp_i are expressions containing the variables of $Spec_2$. We express theorem (9) as follows in a module where $Spec_2$ is defined as $Spec$:

$Id \triangleq \text{INSTANCE } S1 \text{ WITH } v_1 \leftarrow exp_1, \ldots, v_k \leftarrow exp_k$
THEOREM $Spec \Rightarrow Id!Spec$

if $Spec_1$ is defined as $Spec$ in module $S1$. The substitutions in the WITH clause are called a refinement mapping.

4 Auxiliary Variables

Specifications $Spec_1$ and $Spec_2$ can satisfy theorem (8) without there being any refinement mapping that proves it. Consider a system in which a user and the system alternately perform steps in which the user inputs an integer i by setting the interface variable io to $\langle \text{“in”}, i \rangle$ and the system outputs the average avg of the integers input thus far by setting io to $\langle \text{“out”}, avg \rangle$. This system is specified in module $Avg1$ of Figure 5. It uses a variable $inputs$ to remember the sequence of integers input thus far. A user input step appends the integer to the end of $input$, and a system output step reports the average of the integers in $input$. To allow TLC to check the spec, the system uses integer division $\div$ to compute the average. The function $SeqSum$ is defined so $SeqSum[seq]$ is the sum of a sequence seq of integers.

There's a simple way to implement this specification without remembering the sequence of all inputs. Instead, we just remember the number of inputs and their sums. This is described by the specification in module $Avg2$ of Figure 6.

Let $Spec_1$ and $Spec_2$ be specifications $Spec$ of modules $Avg1$ and $Avg2$, respectively. It should be clear that the values of oi in a behavior satisfying $Spec_2$ also are values of oi allowed by a behavior satisfying $Spec_1$. In other words, this theorem should be true:

(10) THEOREM $Spec_2 \Rightarrow \exists\, inputs : Spec_1$

However, no refinement mapping can prove this because the variables of $Avg2$ do not contain the information about past inputs needed to write such a refinement mapping.

The way to prove (10) is by adding an auxiliary variable to $Spec_2$. Adding an auxiliary variable a to a specification S means finding a specification S^a

containing the variables of S plus the additional variable a such that $\exists\, a : S^a$ is equivalent to S. We prove (10) by adding to $Spec_2$ an auxiliary variable a that is not a variable of $Spec_1$ such that we can find a refinement mapping that proves:

(11) THEOREM $Spec_2{}^a \Rightarrow \exists\, inputs : Spec_1$

Since a is not a variable of $Spec_1$, if some behavior b satisfies $\exists\, inputs : Spec_1$ then the behavior obtained from b by letting variable a have any value in any of its states also satisfies it. This means that (11) implies

(12) THEOREM $(\exists\, a : Spec_2{}^a) \Rightarrow \exists\, inputs : Spec_1$

Since $\exists\, a : Spec_2{}^a$ is equivalent to $Spec_2$, theorem (12) implies (10).

There are three types of auxiliary variables: history variables, stuttering variables, and prophecy variables. They are explained in detail in the paper

MODULE *Avg1*

EXTENDS *Integers*, *Sequences*

$$SeqSum[seq \in Seq(Int)] \triangleq$$
$$\text{IF } seq = \langle\,\rangle \text{ THEN } 0 \text{ ELSE } Head(seq) + SeqSum[Tail(seq)]$$

VARIABLES *io*, *inputs*

$$vars \triangleq \langle io, inputs\rangle$$

$$Init \triangleq \land\ io = \langle \text{“out”}, 0\rangle \\ \land\ inputs = \langle\,\rangle$$

$$In \triangleq \land\ io[1] = \text{“out”} \\ \land\ \exists\, i \in Int : \land\ io' = \langle \text{“in”}, i\rangle \\ \land\ inputs' = Append(inputs, i)$$

$$Out \triangleq \land\ io[1] = \text{“in”} \\ \land\ io' = \langle \text{“out”}, SeqSum[inputs] \div Len(inputs)\rangle \\ \land\ inputs' = inputs$$

$$Next \triangleq In \lor Out$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

Figure 5: A system to input integers and output their integer average.

MODULE *Avg2*

EXTENDS *Integers*

VARIABLES io, sum, num

$$vars \triangleq \langle io, sum, num \rangle$$

$$Init \triangleq \land io = \langle \text{“out”}, 0 \rangle \\ \land sum = 0 \\ \land num = 0$$

$$In \triangleq \land io[1] = \text{“out”} \\ \land \exists\, i \in Int : \land io' = \langle \text{“in”}, i \rangle \\ \land sum' = sum + i \\ \land num' = num + 1$$

$$Out \triangleq \land io[1] = \text{“in”} \\ \land io' = \langle \text{“out”}, sum \div num \rangle \\ \land \text{UNCHANGED } \langle sum, num \rangle$$

$$Next \triangleq In \lor Out$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

Figure 6: Another specification of the integer averaging system.

Auxiliary Variables in TLA+ and its associated web page:

http://lamport.azurewebsites.net/tla/auxiliary/auxiliary.html

You should read that explanation if you need to add auxiliary variables to check the correctness of an industrial-sized spec. However, it is heavy going. The description here is shorter and simpler, but omits many details.

History Variables

History variables remember what happened in the past. They are the most commonly used auxiliary variables. A history variable is used to prove theorem (10), where $Spec_1$ and $Spec_2$ are specifications *Spec* of modules $Avg1$ and $Avg2$ of Figures 5 and 6 on pages 11 and 12. We add a history variable h in module $Avg2h$ of Figure 7, which imports the variables and definitions of module $Avg2$ with its EXTENDS statement. Module $Avg2h$ defines *SpecH* to be formula ${Spec_2}^h$—that is, formula *Spec* of module $Avg2$ with history variable h added.

Observe how the initial and next-state formulas and of *SpecH* are obtained from the corresponding formulas *Init* and *Next* of *Spec* by conjoining

MODULE $Avg2h$

EXTENDS $Avg2, Sequences$

VARIABLE h

$varsH \triangleq \langle vars, h \rangle$

$InitH \triangleq Init \land (h = \langle\rangle)$

$InH \triangleq In \land (h' = Append(h, io'[2]))$

$OutH \triangleq Out \land (h' = h)$

$NextH \triangleq InH \lor OutH$

$SpecH \triangleq InitH \land \Box[NextH]_{varsH}$

$A \triangleq$ INSTANCE $Avg1$ WITH $inputs \leftarrow h$

THEOREM $SpecH \Rightarrow A!Spec$

Figure 7: Adding the history variable a to the specification of module $Avg2$.

to *Init* a specification of the initial value of h and conjoining to each subaction of *Next* a specification of the value of h'. It should be clear that *SpecH* describes exactly the same possible sequences of values of the variables *oi*, *sum*, and *num* as does formula *Spec* does. Hence, $\exists\, h : \mathit{SpecH}$ allows the same behaviors as *SpecH*, so the two formulas are equivalent.

The variable h in *SpecH* remembers the same history of input values as does the variable *inputs* of Spec_1 (formula *Spec* of module *Avg*1). So, as shown by the theorem in module $\mathit{Avg}2h$, the refinement mapping $\mathit{inputs} \leftarrow h$ proves (10).

The common uses of history variables are straightforward generalizations of this example.

Stuttering Variables

Let's return to our queue example, where Spec_1 and Spec_2 are formulas *Spec* of modules *FIFO* and *FIFO*2. We used the refinement $\mathit{queue} \leftarrow qG \circ qP$ to show:

(2) THEOREM $\mathit{Spec}_2 \Rightarrow \exists\, \mathit{queue} : \mathit{Spec}_1$

It should be clear that Spec_1 and Spec_2 allow the same values for *op* in behaviors. Therefore $\exists\, qP, qG : \mathit{Spec}_2$ and $\exists\, \mathit{queue} : \mathit{Spec}_1$ should be equivalent. Theorem (2) proves that $\exists\, qP, qG : \mathit{Spec}_2$ implies $\exists\, \mathit{queue} : \mathit{Spec}_1$. To prove the converse implication, we need to prove

(13) THEOREM $\mathit{Spec}_1 \Rightarrow \exists\, qP, qG : \mathit{Spec}_2$

No refinement mapping can prove this because a behavior that satisfies Spec_2 takes more non-stuttering steps than one that satisfies Spec_1—namely, the steps that satisfy the *Move* action. Any expressions of *FIFO* that we substitute for qP and qG must be left unchanged by stuttering steps, so they can't change as often as qP and qG change.

To prove (13) we need to add a *stuttering variable* to Spec_1. This is an auxiliary variable s that allows all steps allowed by Spec_1 plus steps that leave the variables of Spec_1 unchanged (and hence are stuttering steps for Spec_1) but that change s. This is done in module *FIFOs* of Figure 8. Look first at the definitions of *InitS* through *SpecS*. You'll see that they allow the same *Put* and *Out* steps as Spec_1 (formula *Spec* of module *FIFO*), but requires that a *Stutter* step, which leaves the variables of *FIFO* unchanged, must follow every *Put* step.

We define the refinement mapping so that every *Stutter* step implements a *Move* step of Spec_2. This means that, under the refinement mapping, a

MODULE *FIFOs*

EXTENDS *FIFO*

$End(seq) \triangleq \langle seq[Len(seq)] \rangle$

$Front(seq) \triangleq [i \in 1..(Len(seq) - 1) \mapsto seq[i]]$

VARIABLE s

$varsS \triangleq \langle vars, s \rangle$

$InitS \triangleq Init \land (s = 0)$

$PutS \triangleq (s = 0) \land Put \land (s' = 1)$

$GetS \triangleq (s = 0) \land Get \land (s' = s)$

$Stutter \triangleq (s = 1) \land (\text{UNCHANGED } vars) \land (s' = 0)$

$NextS \triangleq PutS \lor GetS \lor Stutter$

$SpecS \triangleq InitS \land \Box[NextS]_{varsS}$

$qPbar \triangleq \text{IF } s = 0 \text{ THEN } \langle\rangle \text{ ELSE } End(seq)$

$qGbar \triangleq \text{IF } s = 0 \text{ THEN } queue \text{ ELSE } Front(queue)$

$F2 \triangleq \text{INSTANCE } FIFO2 \text{ WITH } qP \leftarrow qPbar,\ qG \leftarrow qGbar$

THEOREM $SpecS \Rightarrow F2!Spec$

Figure 8: Adding stuttering variable s to formula *Spec* of module *FIFO*.

behavior satisfying $Spec_1$ implements a behavior of $Spec_2$ in which every number put into qP is immediately moved to qG, so qP never contains more than one number. (To prove (13) we have to show only that every behavior of $Spec_1$ implements some behavior of $Spec_2$; all possible behaviors of $Spec_2$ don't have to be implemented.) We define *qPbar* and *qGbar* so the refinement mapping is written as $qP \leftarrow qPbar$, $qG \leftarrow qGbar$. The definitions use the operators *End* and *Front* defined so that if *seq* equals $\langle s_1, \ldots, s_n \rangle$ for $n > 0$, then $End(seq)$ equals $\langle s_n \rangle$ and $Front(seq)$ equals $\langle s_1, \ldots, s_{n-1} \rangle$. Note that *SpecS* maintains the invariant that $s \neq 0$ implies $queue \neq \langle\rangle$.

There are other ways to define a stuttering variable s that proves (13). For example, we can define it so that the refinement mapping substitutes

the sequence of the last s numbers in *queue* for qP. We can then use the following definitions for *PutS*, *GetS*, and *Stutter*:

$$\begin{aligned}
PutS &\triangleq Put \land (s' = s + 1) \\
GetS &\triangleq (s < Len(queue)) \land Get \land (s' = s) \\
Stutter &\triangleq (s > 0) \land (\text{UNCHANGED } vars) \land (s' = s - 1)
\end{aligned}$$

Defining the refinement mapping in this case is a nice little exercise.

Prophecy Variables

Module *FIFOp* in Figure 9 adds a variable p to specification *Spec* of module *FIFO* in much the same way that module $Avg2h$ added the history variable h to module $Avg2$. In this case, variable p always equals the next number to be appended to the queue. It's set to an arbitrary natural number initially and upon performing each *Put* action.

It's not hard to see that *SpecP* allows behaviors with exactly the same sequences of values of *op* and *queue* as formula *Spec* of module *FIFO*. Thus $\exists\, p : SpecP$ is equivalent to *Spec*, so *SpecP* is obtained from *Spec* by adding the auxiliary variable p. We call p a *prophecy variable* because it "predicts"

MODULE *FIFOp*

EXTENDS *FIFO*

VARIABLE p

$$\begin{aligned}
varsP &\triangleq \langle vars, p \rangle \\
InitP &\triangleq Init \land (p \in Nat) \\
PuP &\triangleq \land\ op' = \langle \text{“put”}, p, -op[3] \rangle \\
&\quad\ \land\ queue' = Append(queue, p) \\
&\quad\ \land\ p' \in Nat \\
GetP &\triangleq Get \land (p' = p) \\
NextP &\triangleq PutP \lor GetP \\
SpecP &\triangleq InitP \land \Box[NextP]_{varsP}
\end{aligned}$$

Figure 9: Adding the variable p to the specification of module *FIFO*.

what will happen in the future—in particular, what value the next *Put* step will choose to append to the queue.

The value of a prophecy variable may make multiple predictions. For example, module *FIFOp*2 in Figure 10 defines formula *SpecP*2, which is obtained by adding the prophecy variable $p2$ to formula *Spec* of module *FIFO*. It defines *ISeq*, *IHead*, and *ITail* to be the operators for infinite sequences that are the obvious analogs of the operators *Seq*, *Head*, and *Tail* for finite sequences. (An infinite sequence is represented as a function whose domain is the set of all positive integers.) At any point during the execution, the value of $p2$ is the sequence of all future input values.[2] Again, it's clear that *Spec*2*p* allows exactly the same sequences of values of *op* and *queue* as formula *Spec*. Thus $\exists\, p2 : SpecP2$ is equivalent to *Spec*.

A prophecy variable is needed to define a refinement mapping that shows

$$(8) \quad \text{THEOREM} \quad Spec_2 \Rightarrow \exists\, v_1, \ldots, v_k : Spec_1$$

if $Spec_1$ encodes in the variables v_i choices about the values of the other variables sooner than those choices are made in $Spec_2$. For example, $Spec_1$ might at some point set v_1 to a sequence of four values and then "reveal" those values by outputting them one at a time through other variables. This could be implemented in $Spec_2$ by choosing those values when they are output. Of course, this is a silly example. Here's a realistic example.

Suppose that we modify the FIFO queue spec so there are multiple processes that perform *Put* actions to append elements to *queue*. For simplicity, let's assume only a single process performs the *Get* action. It's easy to do this by simply adding to the value of *op* a fourth component indicating which process is performing the *Put* operation.

This spec is unrealistic because it requires the *Get* operation to obtain the values from the queue in the exact order in which the corresponding *Put* operations are executed. This means that if one process performs a *Put* a few nanoseconds before another process does, then the first process's value must be appended to *queue* before the second process's value. This is almost impossible to implement if those two processes are located on different continents.

A practical spec needs to decouple the action of changing *op* from the action of appending a value to *queue*. It also needs to provide some guarantee about when the value is actually appended to *queue*. One way of doing this is expressed by the following specification. The *Put* action of module *FIFO*

[2] Since we are specifying only safety, a behavior may terminate at any point. Therefore, the value of $p2$ predicts the sequence of all values that might be input.

MODULE *FIFOp2*

EXTENDS *FIFO*

$$
\begin{array}{lcl}
ISeq(S) & \triangleq & [Nat\ 0 \to S] \\
IHead(seq) & \triangleq & seq[1] \\
ITail(seq) & \triangleq & [i \in Nat\ 0 \mapsto seq[i+1]]
\end{array}
$$

VARIABLE $p2$

$$
\begin{array}{lcl}
varsP2 & \triangleq & \langle vars, p2 \rangle \\
InitP2 & \triangleq & Init \land (p2 \in ISeq(Nat)) \\
PutP2 & \triangleq & \land\ op' = \langle \text{“put”}, IHead(p2), -op[3] \rangle \\
 & & \land\ queue' = Append(queue, IHead(p2)) \\
 & & \land\ p2' = ITail(p2) \\
GetP2 & \triangleq & Get \land (p2' = p2) \\
NextP2 & \triangleq & PutP2 \lor GetP2 \\
SpecP2 & \triangleq & InitP2 \land \Box[NextP2]_{varsP2}
\end{array}
$$

Figure 10: Adding the variable $p2$ to the specification of module *FIFO*.

is split into three actions: *BeginPut* that changes *op* to indicate that the operation has begun; *DoPut* that appends the value to *queue* and leaves *op* unchanged; and *EndPut* that changes *op* to indicate that the operation has completed. Such a FIFO queue is called *linearizable*; it is easily generalized to an arbitrary data structure accessed by multiple users.

Because it is general and easy to understand, linearizability is a common requirement for multiuser systems. However, a linearizable FIFO queue spec implies that if two *Put* operations are executed concurrently, then immediately after both processes have performed their *EndPut* actions, their values have both been added to *queue*—which means that the order in which they will be removed from *queue* by a *Get* operation has been determined. However, if the spec has the form $\exists\, queue : \ldots$, then it can be implemented by an algorithm in which, at that point, it is still possible for the values to be removed by *Get* operations in either order. We must add a prophecy variable to the algorithm's spec to define a refinement mapping to show that such an algorithm implements the linearizability spec.

Completeness

It turns out that the ability to add auxiliary variables means we can, in principle, always prove implementation with a refinement mapping. If this theorem is true:

(8) THEOREM $Spec_2 \Rightarrow \exists\, v_1, \ldots, v_k : Spec_1$

then there exists a refinement mapping that proves it. The idea of the proof is as follows. Don't worry if you can't follow it; it's of no practical importance.

Much as we added the prophecy variable $p2$ to predict the sequence of future "put" values of *op* in the current behavior satisfying the *FIFO* algorithm's spec, we can add to any spec a prophecy variable *pAll* that always predicts the sequence of future system states (assignments of values to all the spec's variables). We can also add a history variable *hAll* that remembers the entire sequence of system states up to and including the current state. Thus, the concatenation of the values of *hAll* and *pAll* describe the complete behavior that the system is currently executing, and the system is currently in the j^{th} state of that behavior, where j equals the length of *hAll*.

For simplicity, let's ignore stuttering invariance. Theorem (8) then means that for every behavior b satisfying $Spec_2$, there is a behavior $F(b)$ satisfying $Spec_1$ such that the values of all variables of $Spec_1$ other than the v_i are the same in b and $F(b)$. If $Spec_1$ and $Spec_2$ are TLA$^+$ formulas, then $F(b)$ can, in principle, be written as a TLA$^+$ formula. Adding the auxiliary variables *pAll* and *hAll*, we can define the refinement mapping

$$v_1 \leftarrow exp_1, \ldots, v_k \leftarrow exp_k$$

that proves (8) by letting exp_i equal the value of v_i in the j^{th} state of $F(b)$, where the values of *pAll* and *hAll* indicate that the system is currently in the j^{th} state of an execution of behavior b.

When stuttering invariance is taken into account, things get more complicated because we have to add a stuttering variable to keep the behaviors b and $F(b)$ "in step". However, the basic idea is the same.

This result is of only theoretical interest because the proof does not provide a practical way to find a refinement mapping. However, it shows the expressive power of auxiliary variables, which is why I believe that the necessary refinement mapping can always be found in practice.

5 Refinement in General

The TLA$^+$ specs we write can contain two kinds of variables:

observable variables Also called *interface* or *externally visible* variables. The purpose of the spec is to describe the sequences of possible values of these variables.

internal variables Also called *unobservable* or *invisible* variables. These variables are the "internal gears" used to control the values of the interface variables.

In our FIFO queue specifications, *op* is an observable variable; all the other variables are internal. In the specifications of modules $Avg1$ and $Avg2$, the variable *io* is an observable variable; all other variables are internal.

The philosophically correct way to write a specification is

(14) $\exists\, v_1, \ldots, v_k \,:\, Spec$

where (the safety part of) *Spec* has the form $Init \wedge \Box[Next]_{vars}$ and the v_i are the internal variables. As explained in Section 2, it's a bit awkward to write (14) in TLA$^+$. Moreover, there is no practical reason to do so, since neither the TLC model checker nor the TLAPS prover can handle the $\exists$ operator. TLC is unlikely ever to handle it, since checking if a behavior satisfies such a formula is inherently difficult. While there is no problem reasoning about $\exists$, there is little incentive to implement it in TLAPS. Instead of writing the specification (14), we can just call *Spec* the specification and state in a comment that the v_i are internal variables.

A reason for writing philosophically correct specs is that they provide a simple, natural definition of what it means for one spec to implement another. If PC_1 and PC_2 have the form (14), then PC_2 implements (or refines) PC_1 iff the observable part of any behavior satisfying PC_2 is the observable part of a behavior satisfying PC_1—where the observable part of a behavior is the sequence of values assigned to the observable variables. Mathematically, "PC_2 implements PC_1" asserts the truth of:

(15) THEOREM $PC_2 \Rightarrow PC_1$

Reduced to a slogan, this means: implementation is implication.

In practice, implementation is implication only when both specs are written at the same level of abstraction. For example, (15) might hold if PC_2 describes a method in a Java class and PC_1 asserts a relation that must hold between an object's values before and after executing the method. However,

suppose PC_1 describes a message-passing algorithm and PC_2 describes an implementation of that algorithm with a packet-switching network. The values of observable variables of PC_1 will be described in terms of messages; the values of observable variables of PC_2 will be described in terms of packets, saying nothing about messages. Instead of (15), we would expect implementation to mean

(16) THEOREM $PC_2 \Rightarrow (PC_1 \text{ WITH } o_1 \leftarrow oexp_1, \ldots, o_m \leftarrow oexp_m)$

where the o_i are the observable variables of PC_1 and the $oexp_i$ are expressions involving the observable variables of PC_2. For example, if the value o_1 is a set of messages in transit in a state of a behavior satisfying PC_1, then $oexp_1$ might be the expression that describes this set of messages in terms of the set of packets in transit in a behavior satisfying PC_2.

Let PC_1 equal $\exists\, v_1, \ldots, v_k : Spec_1$ and let PC_2 equal $\exists \ldots : Spec_2$. To verify (16), we find a refinement mapping $v_1 \leftarrow exp_1, \ldots, v_k \leftarrow exp_k$ such that

$$\begin{aligned} \text{THEOREM } Spec_2 \Rightarrow (\,&(Spec_1 \text{ WITH } v_1 \leftarrow exp_1, \ldots, v_k \leftarrow exp_k) \\ &\text{WITH } o_1 \leftarrow oexp_1, \ldots, o_m \leftarrow oexp_m) \end{aligned}$$

which is equivalent to

(17) $$\begin{aligned} \text{THEOREM } Spec_2 \Rightarrow (&Spec_1 \text{ WITH } v_1 \leftarrow exp_1, \ldots, v_k \leftarrow exp_k, \\ &o_1 \leftarrow oexp_1, \ldots, o_m \leftarrow oexp_m) \end{aligned}$$

This is exactly what we did in Section 3 (equation (9) on page 10), except instead of a refinement mapping that substitutes expressions only for the internal variables v_i of $Spec_1$, we use one that substitutes for all the variables of $Spec_1$.

In practice, we forget about (16) and take (17) to be the definition of implementation. More precisely, we talk only about implementation under a refinement mapping, where (17) asserts that $Spec_2$ implements $Spec_1$ under the refinement mapping $v_1 \leftarrow exp_1, \ldots, o_m \leftarrow oexp_m$. This allows the expressions $oexp_i$ to mention the internal as well as the observable variables of $Spec_2$, which can be useful. Implementation in the sense of Section 3 is the special case when each observable variable o_i of $Spec_1$ is implemented by the variable with the same name o_i in $Spec_2$.

In the general case, it's meaningless to say that a spec $Spec_2$ implements a spec $Spec_1$ without saying what expressions are substituted for the observable variables of $Spec_1$. For *any* two specs $Spec_1$ and $Spec_2$, by adding

suitable auxiliary variables to $Spec_2$, it's possible to define a refinement mapping under which $Spec_2$ implements $Spec_1$. To decide if implementing $Spec_1$ under a refinement mapping is an interesting property of $Spec_2$, you have to examine carefully the expressions the refinement mapping substitutes for the variables of $Spec_1$—especially its observable variables. This is just a special case of the general observation that you should examine a property carefully to be sure that showing that a spec satisfies it tells you something useful.