Auxiliary Variables in TLA+

Leslie Lamport, Stephan Merz

Auxiliary Variables in TLA$^+$

Leslie Lamport and Stephan Merz

27 May 2017

Abstract

Auxiliary variables are often needed for verifying that an implementation is correct with respect to a higher-level specification. They augment the formal description of the implementation without changing its semantics—that is, the set of behaviors that it describes. This paper explains rules for adding history, prophecy, and stuttering variables to TLA$^+$ specifications, ensuring that the augmented specification is equivalent to the original one. The rules are explained with toy examples, and they are used to verify the correctness of a simplified version of a snapshot algorithm due to Afek et al.

Contents

References

1 Introduction

With state-based methods, checking that an implementation satisfies a higher-level specification requires describing how the higher-level concepts in the specification are represented by the lower-level data structures of the implementation. This approach was first proposed in the domain of sequential systems by Hoare in 1972 [5]. Hoare called the description an *abstraction function*. The generalization to concurrent systems was called a *refinement mapping* by Abadi and Lamport [2]. They observed that constructing a refinement mapping may require adding auxiliary variables to the implementation—variables that do not alter the behavior of the actual variables and need not be implemented.

This paper is about adding auxiliary variables to TLA$^+$ specifications. The ideas we present should be applicable to other state-based specification methods, but we make no attempt to translate them into those other methods. We hope that a future paper will present the basic ideas in a language-independent way and will contain soundness and completeness proofs. Our goal here is to teach engineers writing TLA$^+$ specifications how to add auxiliary variables when they need them.

We assume the reader can understand TLA$^+$ specifications. A basic understanding of refinement mappings will be helpful but isn't necessary. TLA$^+$ and refinement mappings are explained in the book *Specifying Systems* [8] and in material listed on the TLA web page [7].

This is a long paper, in part because it contains 25 figures with actual TLA$^+$ specifications. The paper contains hyperlinks, and we recommend reading the pdf version on line. If you are doing that, you can download the source files for all the TLA$^+$ specifications described in this paper by clicking here. Otherwise, you can find the URL in the reference list [6]. We expect that engineers will have to study the specifications carefully to learn how to add auxiliary variables to their specifications.

We explain three kinds of auxiliary variables: history, prophecy, and stuttering variables. History variables record information about the system's past behavior. They have been used since at least the 1970s [9]. They were sometimes called "ghost" variables. Prophecy variables predict the future behavior of the system. They were introduced by Abadi and Lamport in 1991 [2]. The need for them was also implicit in an example presented in Herlihy and Wing's classic paper on linearizability [4]. We found the original prophecy variables very difficult to use in practice. The prophecy variables described here are new, and our experience with them so far indicates that they are reasonably easy to use in practice. Stuttering variables add "stuttering" steps—ones that leave the specification's actual variables unchanged. Abadi and Lamport originally used prophecy variables to add stuttering steps, but we have found it better to introduce stuttering steps with a separate kind of variable.

We will mostly ignore liveness and consider only safety specifications. The

canonical form of a TLA$^+$ specification consists of a safety specification of the form $Init \wedge \Box[Next]_{vars}$ conjoined with a liveness condition. An auxiliary variable is added by modifying the safety specification, but leaving the liveness condition unchanged. Liveness therefore poses no problem for auxiliary variables and is discussed only briefly.

2 Refinement Mappings

We will illustrate refinement mappings with a simple, useless example. A user presents a server with a sequence of integer inputs. The server responds to each input value i with one of the following outputs: *Hi* if i is the largest number input so far, *Lo* if it's the smallest number input so far, *Both* if it's both, and *None* if it's neither. We declare *Hi*, *Lo*, *Both*, and *None* in a CONSTANTS statement. They are assumed not to be integers.

2.1 Specification *MinMax*1

Our first specification appears in a module named *MinMax*1. It describes the interaction of the user and the server with two variables: a variable x to hold an input or a response, and a variable *turn* that indicates whether it's the user's turn to input a value or the server's turn to respond. The specification also uses a variable y to hold the set of values input so far. The initial predicate is

$$\begin{array}{lll} Init & \triangleq & \wedge\ x = None \\ & & \wedge\ turn = \text{“input”} \\ & & \wedge\ y = \{\} \end{array}$$

The next-state relation *Next* equals $InputNum \vee Respond$ where *InputNum* is the user's input action and *Respond* is the server's output action. The definition of *InputNum* is simple:

$$\begin{array}{lll} InputNum & \triangleq & \wedge\ turn = \text{“input”} \\ & & \wedge\ turn' = \text{“output”} \\ & & \wedge\ x' \in Int \\ & & \wedge\ y' = y \end{array}$$

To define the *Respond* action, we must first define operators *setMax* and *setMin* so that, for any finite nonempty set S of integers, $setMax(S)$ and $setMin(S)$ are the maximum and minimum element, respectively, of S. The definitions are:

$$\begin{array}{lll} setMax(S) & \triangleq & \text{CHOOSE } t \in S : \forall\, s \in S : t \geq s \\ setMin(S) & \triangleq & \text{CHOOSE } t \in S : \forall\, s \in S : t \leq s \end{array}$$

The definition of *Respond* is:

$$
\begin{array}{lll}
\mathit{Respond} & \triangleq & \land\ \mathit{turn} = \text{“output”} \\
& & \land\ \mathit{turn}' = \text{“input”} \\
& & \land\ y' = y \cup \{x\} \\
& & \land\ x' = \text{IF } x = \mathit{setMax}(y') \\
& & \qquad \text{THEN IF } x = \mathit{setMin}(y') \text{ THEN } \mathit{Both} \text{ ELSE } \mathit{Hi} \\
& & \qquad \text{ELSE IF } x = \mathit{setMin}(y') \text{ THEN } \mathit{Lo} \text{ ELSE } \mathit{None}
\end{array}
$$

Note that action *InputNum* is enabled iff *turn* equals “input”, and action *Respond* is enabled iff *turn* equals “output”. The complete specification is the formula

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

where *vars* is the tuple $\langle x, \mathit{turn}, y \rangle$ of variables. The module *MinMax*1 we have written thus far is shown in Figure 1.

2.2 The Hiding Operator ∃

Recall that a behavior is a sequence of states, where a state is an assignment of values to all possible variables. For specification *Spec* of module *MinMax*1, the interesting part of the state is the assignment of values to x, *turn*, and y. Our specification allows all other variables to have any value at any state of any behavior.

The purpose of this specification is to describe the interaction of the user and the server. This interaction is described by the values of x and *turn*. The value of y is needed only to describe how the values of x and *turn* can change. We consider x and *turn* to be the externally visible or observable values of the specification and y to be an internal variable. A philosophically correct specification of our user/server system would allow only behaviors in which the values of x and *turn* are as specified by *Spec*, but would not constrain the value of y. We can write such a specification in terms of the temporal-logic operator $\exists$.

For any temporal formula F and variable v, the formula $\exists\, v : F$ is defined approximately as follows. A behavior σ satisfies $\exists\, v : F$ iff there exists a behavior τ satisfying F such that τ is identical to σ except for the values its states assign to v. The precise definition is more complicated because a temporal formula of TLA$^+$ may neither require nor prohibit stuttering steps, but we will use the approximate definition for now. The operator $\exists$ is much like the ordinary existential quantifier $\exists$ except that $\exists\, v : F$ asserts the existence not of a single value for v that makes F true but rather of a sequence of values, one for each state in the behavior, that makes F true on the behavior. This temporal existential quantifier $\exists$ satisfies most of the properties of ordinary quantification. For example, if the variable v does not occur in formula F, then $\exists\, v : F$ is equivalent to F. We sometimes read the formula $\exists\, v : F$ as “F with v hidden”.

The philosophically correct specification of the *MinMax*1 system should consist of formula *Spec* with y hidden. The obvious way to write this specification is $\exists\, y : \mathit{Spec}$. However, we can't do that for the following reason. Suppose

MODULE *MinMax*1

EXTENDS *Integers*

$setMax(S) \triangleq$ CHOOSE $t \in S : \forall\, s \in S : t \geq s$
$setMin(S) \triangleq$ CHOOSE $t \in S : \forall\, s \in S : t \leq s$

CONSTANTS *Lo*, *Hi*, *Both*, *None*
ASSUME $\{Lo, Hi, Both, None\} \cap Int = \{\}$

VARIABLES x, *turn*, y
$vars \triangleq \langle x, turn, y \rangle$

$$
\begin{aligned}
Init \triangleq\ & \wedge x = None \\
& \wedge turn = \text{“input”} \\
& \wedge y = \{\}
\end{aligned}
$$

$$
\begin{aligned}
InputNum \triangleq\ & \wedge turn = \text{“input”} \\
& \wedge turn' = \text{“output”} \\
& \wedge x' \in Int \\
& \wedge y' = y
\end{aligned}
$$

$$
\begin{aligned}
Respond \triangleq\ & \wedge turn = \text{“output”} \\
& \wedge turn' = \text{“input”} \\
& \wedge y' = y \cup \{x\} \\
& \wedge x' = \text{IF } x = setMax(y') \\
& \qquad \text{THEN IF } x = setMin(y') \text{ THEN } Both \text{ ELSE } Hi \\
& \qquad \text{ELSE IF } x = setMin(y') \text{ THEN } Lo \text{ ELSE } None
\end{aligned}
$$

$Next \triangleq InputNum \vee Respond$

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

Figure 1: Module *MinMax*1.

a module M defines *exp* to equal some expression. TLA$^+$ does not allow the expression

(2.1) $\{v \in exp : v^2 > 42\}$

to appear at any point in module M where v is already declared or defined. Since *exp* must be defined for the expression to have a meaning, this means that (2.1) is illegal if v is a declared variable that appears in the definition of *exp*. Similarly, the formula $\exists\, y : Spec$ is illegal because y appears in the definition of *Spec*.[1]

[1]There are languages for writing math precisely that allow expression (2.1) even if v is

There are ways to write the formula *Spec* with v hidden in TLA$^+$. The most convenient ones involve writing it in another module that instantiates module *MinMax*1. Chapter 4 of *Specifying Systems* [8] explains one way to do this. However, there's little reason to do it since the TLA$^+$ tools cannot check specifications written with $\exists$. (The TLAPS proof system may eventually be able to reason about it.) Instead, we take the formula $\exists\, y : \mathit{Spec}$ to be an abbreviation for the formula $\exists\, y : [\![\mathit{Spec}]\!]$, where $[\![\mathit{Spec}]\!]$ is the formula obtained from *Spec* by expanding all definitions. Formula $[\![\mathit{Spec}]\!]$ contains only: TLA$^+$ primitives; the constants *Hi*, *Lo*, *Both*, and *None*; and the variables x, *turn*, and y. Thus $\exists\, y : \mathit{Spec}$ is meaningful in a context in which x and *turn* are declared variables. If used in a context in which y already has a meaning, we interpret $\exists\, y : \mathit{Spec}$ to be the formula obtained from $\exists\, y : [\![\mathit{Spec}]\!]$ by replacing y everywhere with a new symbol.

What it means to expand all definitions in an expression is not as simple as it might seem. Consider the following definition:

$$\mathit{NotUnique}(a) \triangleq \exists i : i \neq a \tag{2.2}$$

It's clear that the following theorem is true:

$$\text{THEOREM } \forall a : \mathit{NotUnique}(a) \tag{2.3}$$

Now suppose we follow the definition of *NotUnique* with:

$$\begin{array}{l}\text{CONSTANT } i \\ \text{THEOREM } \mathit{NotUnique}(i)\end{array} \tag{2.4}$$

Theorem (2.3) obviously implies the theorem of (2.4). However, a naive expansion of the definition of *NotUnique* tells us that $[\![\mathit{NotUnique}(i)]\!]$ equals $\exists i : i \neq i$, which equals FALSE. The problem is clear: the bound identifier i in the definition of *NotUnique* is not the same i as the one declared in the CONSTANT declaration. The following definition of *NotUnique* is equivalent to (2.2)

$$\mathit{NotUnique}(a) \triangleq \exists \mathit{jku} : \mathit{jku} \neq a$$

and with the naive expansion of this definition, $[\![\mathit{NotUnique}(i)]\!]$ equals the true formula $\exists \mathit{jku} : \mathit{jku} \neq i$ of (2.4).

The easiest way to define the meaning of expanding all definitions in an expression is to consider (2.2) to define $\mathit{NotUnique}(a)$ to equal something like $\exists v\_743 : v\_743 \neq a$, where v_743 is an identifier that cannot be used anywhere else. In general, every bound identifier in a definition is replaced by some unique identifier.

already declared. In such a language, $\exists\, y : \mathit{Spec}$ would be equivalent to $\exists\, w : \mathit{Spec}$ for any identifier w, which means it would be equivalent to *Spec*.

Recursive definitions are not a problem for complete expansion of definitions because in TLA$^+$, a recursive definition is just an abbreviation for a non-recursive one. For example

$$f[i \in Nat] \triangleq \text{IF } i = 0 \text{ THEN } 1 \text{ ELSE } i * f[i-1]$$

is an abbreviation for

$$f \triangleq \text{CHOOSE } f : f = [i \in Nat \mapsto \text{IF } i = 0 \text{ THEN } 1 \text{ ELSE } i * f[i-1]]$$

so the bound identifier f to the right of the "$\triangleq$" is not the same symbol as the f being defined. (A recursive operator definition is an abbreviation for a much more complicated ordinary definition.)

2.3 Specification *MinMax*2

The specification of our system in module *MinMax*1 uses the variable y to remember the set of all values that the user has input. Module *MinMax*2 specifies the same user/server interaction that remembers only the smallest and largest values input so far, using the variables *min* and *max*. Representing the initial state, before any values have been input, is a little tricky. It would be simpler if the standard *Integers* module defined a value ∞ such that $-\infty < i < \infty$ for all integers i. So, we will write the spec pretending that it did. Afterwards, we'll describe how to obtain an actual TLA$^+$ spec.

The initial predicate of the specification is:

$$\begin{aligned} Init \triangleq\ & \land x = None \\ & \land turn = \text{“input”} \\ & \land min = \infty \\ & \land max = -\infty \end{aligned}$$

The user's *InputNum* action is the same as for the *MinMax*1 specification, except it leaves *min* and *max* rather than y unchanged:

$$\begin{aligned} InputNum \triangleq\ & \land turn = \text{“input”} \\ & \land turn' = \text{“output”} \\ & \land x' \in Int \\ & \land \text{UNCHANGED } \langle min, max \rangle \end{aligned}$$

Here is the system's *Respond* action:

$$\begin{aligned} Respond \triangleq\ & \land turn = \text{“output”} \\ & \land turn' = \text{“input”} \\ & \land min' = \text{IF } x \leq min \text{ THEN } x \text{ ELSE } min \\ & \land max' = \text{IF } x \geq max \text{ THEN } x \text{ ELSE } max \\ & \land x' = \text{IF } x = max' \\ & \qquad \text{THEN IF } x = min' \text{ THEN } Both \text{ ELSE } Hi \\ & \qquad \text{ELSE IF } x = min' \text{ THEN } Lo \text{ ELSE } None \end{aligned}$$

As usual, the complete specification is

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

where this time *vars* is the tuple $\langle x, turn, min, max \rangle$ of variables.

To turn this into a TLA$^+$ specification, we replace ∞ and $-\infty$ by two constants *Infinity* and *MinusInfinity*. In the definition of *Respond*, we replace $x \leq min$ and $x \geq max$ by $IsLeq(x, min)$ and $IsGeq(x, max)$, where *IsLeq* and *IsGeq* are defined by

$$\begin{aligned} IsLeq(i, j) &\triangleq (j = Infinity) \vee (i \leq j) \\ IsGeq(i, j) &\triangleq (j = MinusInfinity) \vee (i \geq j) \end{aligned}$$

These definitions must be preceded by declarations or definitions of *Infinity* and *MinusInfinity*. They can equal any values, except that they must not be equal and neither of them should be an integer—otherwise, the spec wouldn't mean what we want it to mean. We could declare *Infinity* and *MinusInfinity* in a CONSTANT declaration and then add an ASSUME statement asserting that they aren't in *Int*. However, we prefer to define them like this:

$$\begin{aligned} Infinity &\triangleq \text{CHOOSE } n : n \notin Int \\ MinusInfinity &\triangleq \text{CHOOSE } n : n \notin (Int \cup \{Infinity\}) \end{aligned}$$

This completes module *MinMax*2, which is shown in its entirety in Figure 2.

As before, we consider x and *turn* to be the externally visible variables, and *min* and *max* to be internal variables. The philosophically correct specification, which hides the internal variables *min* and *max*, is $\exists\, min, max : Spec$. Of course, this is an abbreviation for $\exists\, min : (\exists\, max : Spec)$, which is equivalent to $\exists\, max : (\exists\, min : Spec)$.

2.4 The Relation Between the Two Specifications

Using the standard TLA$^+$ naming convention, we have given the two specifications the same name *Spec*. To distinguish them, let $Spec_1$ be the specification *Spec* of module *MinMax*1 and $Spec_2$ be the *Spec* of module *MinMax*2.

It should be clear that both specifications describe the same behavior of the external variables x and *turn*. This means that if we hide the internal variable y of $Spec_1$ and the internal variables *min* and *max* of $Spec_2$, we should obtain equivalent specifications. More precisely, we expect to this to be true:

$$(2.5) \quad (\exists\, y : Spec_1) \equiv (\exists\, min, max : Spec_2)$$

This formula is equivalent to the conjunction of these two formulas

$$(2.6) \quad (\exists\, y : Spec_1) \Rightarrow (\exists\, min, max : Spec_2)$$

$$(2.7) \quad (\exists\, min, max : Spec_2) \Rightarrow (\exists\, y : Spec_1)$$

MODULE *MinMax2*

EXTENDS *Integers*, *Sequences*

CONSTANTS *Lo*, *Hi*, *Both*, *None*
ASSUME $\{Lo, Hi, Both, None\} \cap Int = \{\}$

$Infinity \triangleq$ CHOOSE $n : n \notin Int$
$MinusInfinity \triangleq$ CHOOSE $n : n \notin (Int \cup \{Infinity\})$

$IsLeq(i, j) \triangleq (j = Infinity) \lor (i \leq j)$
$IsGeq(i, j) \triangleq (j = MinusInfinity) \lor (i \geq j)$

VARIABLES x, *turn*, *min*, *max*
$vars \triangleq \langle x, turn, min, max \rangle$

$$
\begin{aligned}
Init \triangleq\ & \land x = None \\
& \land turn = \text{“input”} \\
& \land min = Infinity \\
& \land max = MinusInfinity
\end{aligned}
$$

$$
\begin{aligned}
InputNum \triangleq\ & \land turn = \text{“input”} \\
& \land turn' = \text{“output”} \\
& \land x' \in Int \\
& \land \text{UNCHANGED } \langle min, max \rangle
\end{aligned}
$$

$$
\begin{aligned}
Respond \triangleq\ & \land turn = \text{“output”} \\
& \land turn' = \text{“input”} \\
& \land min' = \text{IF } IsLeq(x, min) \text{ THEN } x \text{ ELSE } min \\
& \land max' = \text{IF } IsGeq(x, max) \text{ THEN } x \text{ ELSE } max \\
& \land x' = \text{IF } x = max' \text{ THEN IF } x = min' \text{ THEN } Both \text{ ELSE } Hi \\
& \qquad\qquad \text{ELSE IF } x = min' \text{ THEN } Lo \text{ ELSE } None
\end{aligned}
$$

$Next \triangleq InputNum \lor Respond$

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

Figure 2: Module *MinMax2*.

We verify (2.5) by separately verifying (2.6) and (2.7). We first consider (2.6).

Formula (2.6) asserts of a behavior σ that if there exists some way of assigning values to y in the states of σ to make it satisfy $Spec_1$, then σ satisfies $\exists\, min, max : Spec_2$. Since the variable y does not appear in $\exists\, min, max : Spec_2$, changing the values of y in the states of σ doesn't affect whether it satisfies that formula. This implies that to verify (2.6), it suffices to show that any behavior σ that satisfies $Spec_1$ also satisfies $\exists\, min, max : Spec_2$. In other words, to verify

(2.6), it suffices to verify

(2.8) $\quad Spec_1 \;\Rightarrow\; (\exists\, min, max \;:\; Spec_2)$

To verify (2.8), we must show that for any behavior σ that satisfies $Spec_1$, there exists a way of assigning values to the variables min and max in the states of σ that makes the resulting behavior satisfy $Spec_2$. A standard way of doing that is to find explicit expressions $\overline{min}$ and $\overline{max}$ such that, if in each state of a behavior we assign to the variables min and max the values of $\overline{min}$ and $\overline{max}$ in that state, then the resulting behavior satisfies $Spec_2$. We do this by showing that any behavior satisfying $Spec_1$ satisfies the formula obtained by substituting $\overline{min}$ for min and $\overline{max}$ for max in $Spec_2$. Let's write that formula $\overline{[\![Spec_2]\!]}$, emphasizing that we must expand all definitions in $Spec_2$ before substituting $\overline{min}$ for min and $\overline{max}$ for max. So, we verify (2.8) by verifying

(2.9) $\quad Spec_1 \;\Rightarrow\; \overline{[\![Spec_2]\!]}$

We can write formula $\overline{[\![Spec_2]\!]}$ (or more precisely, a formula equivalent to it) in module $MinMax1$ as follows. We first add the statement

$$M \;\triangleq\; \text{INSTANCE } MinMax2 \text{ WITH } min \leftarrow \overline{min},\; max \leftarrow \overline{max}$$

For every defined symbol def in module $MinMax2$, this statement defines $M\,!\,def$ to be equivalent to $\overline{[\![def]\!]}$, the formula whose definition is obtained by substituting $\overline{min}$ for min and $\overline{max}$ for max in the formula obtained by expanding all definitions in the definition of def in $MinMax2$.[2] This INSTANCE statement therefore defines $M\,!\,Spec$ to be equivalent to $\overline{[\![Spec_2]\!]}$, allowing us to write (2.9) in module $MinMax1$ as the theorem:

$$\text{THEOREM } Spec \;\Rightarrow\; M\,!\,Spec$$

We can write a TLA$^+$ proof of this theorem and check it with the TLAPS theorem prover. We can also have TLC check this theorem by creating a model for module $MinMax1$ with specification $Spec$ that substitutes a finite set of integers for Int and checks the property $M\,!\,Spec$. But before we can do that, we have to determine what the expressions $\overline{min}$ and $\overline{max}$ are in the INSTANCE statement.

Formula (2.9) asserts that in a behavior σ satisfying $Spec_1$, if in each state of σ we assign to min and max the values of $\overline{min}$ and $\overline{max}$ in that state, then the resulting behavior satisfies $Spec_2$. One way of thinking about this is that in a behavior satisfying $Spec_1$, the values of $\overline{min}$ and $\overline{max}$ simulate the values that $Spec_2$ requires min and max to assume.

A little thought reveals that $\overline{min}$ and $\overline{max}$ should be defined as indicated in this statement:

[2] Note that the declared constants Hi, Lo, $Both$, and $None$ of module $MinMax2$ have been implicitly instantiated by the constants of the same name declared in $MinMax1$.

$$
\begin{array}{lll}
\mathit{Infinity} & \triangleq & \text{CHOOSE } n : n \notin \mathit{Int} \\
\mathit{MinusInfinity} & \triangleq & \text{CHOOSE } n : n \notin (\mathit{Int} \cup \{\mathit{Infinity}\})
\end{array}
$$

$$
\begin{array}{l}
M \triangleq \text{INSTANCE } \mathit{MinMax2} \\
\qquad \text{WITH } \mathit{min} \leftarrow \text{IF } y = \{\} \text{ THEN } \mathit{Infinity} \quad \text{ELSE } \mathit{setMin}(y), \\
\qquad\qquad \mathit{max} \leftarrow \text{IF } y = \{\} \text{ THEN } \mathit{MinusInfinity} \text{ ELSE } \mathit{setMax}(y)
\end{array}
$$

Figure 3: Additions to module *MinMax*1.

$$
\begin{array}{l}
M \triangleq \text{INSTANCE } \mathit{MinMax2} \\
\qquad \text{WITH } \mathit{min} \leftarrow \text{IF } y = \{\} \text{ THEN } \mathit{Infinity} \quad \text{ELSE } \mathit{setMin}(y), \\
\qquad\qquad \mathit{max} \leftarrow \text{IF } y = \{\} \text{ THEN } \mathit{MinusInfinity} \text{ ELSE } \mathit{setMax}(y)
\end{array}
$$

Of course, we need to define *Infinity* and *MinusInfinity* before we can write that statement. They should be defined to be the same values as in *MinMax*2, so we just copy the definitions from that module into module *MinMax*1. We have added the statements in Figure 3 to the bottom of the module in Figure 1

2.5 Refinement In General

In general, we have two specs: Spec_1 with variables $x_1, \ldots, x_m, y_1, \ldots, y_n$, and Spec_2 with variables $x_1, \ldots, x_m, z_1, \ldots, z_p$. For compactness let $\mathbf{x}$ denote $x_1, \ldots, x_m$, let $\mathbf{y}$ denote $y_1, \ldots, y_n$ and let $\mathbf{z}$ denote $z_1, \ldots, z_p$. We consider $\mathbf{x}$ to be the externally visible variables of both specifications, and we consider $\mathbf{y}$ and $\mathbf{z}$ to be internal variables.

The specifications with their internal variables hidden are written $\exists\, \mathbf{y} : \mathit{Spec}_1$ and $\exists\, \mathbf{z} : \mathit{Spec}_2$. To verify that $\exists\, \mathbf{y} : \mathit{Spec}_1$ implements $\exists\, \mathbf{z} : \mathit{Spec}_2$, we must show that for each behavior satisfying Spec_1, there is some way to assign values of the variables $\mathbf{z}$ in each state so that the resulting behavior satisfies Spec_2. We do that by explicitly specifying those values of $\mathbf{z}$ in terms of the values of $\mathbf{x}$ and $\mathbf{y}$. More precisely, for each z_i we define an expression $\overline{z_i}$ in terms of the variables $\mathbf{x}$ and $\mathbf{y}$ and show that Spec_1 implements $\overline{[\![\mathit{Spec}_2]\!]}$, the specification obtained by expanding all definitions in Spec_2 and substituting $z_1 \leftarrow \overline{z_1}, \ldots, z_p \leftarrow \overline{z_p}$ in the resulting formula. This substitution is called a *refinement mapping*; and if Spec_1 implements $\overline{[\![\mathit{Spec}_2]\!]}$, then we say that Spec_1 implements Spec_2 under the refinement mapping.

The assertion that Spec_1 implements Spec_2 under the refinement mapping $z_1 \leftarrow \overline{z_1}, \ldots, z_p \leftarrow \overline{z_p}$ can be expressed in TLA$^+$ as follows. Suppose Spec_1 is formula *Spec*1 in a module named *Mod*1, and Spec_2 is formula *Spec*2 in a module named *Mod*2. (*Spec*1 and *Spec*2 can be the same identifier.) For some identifier

Id, we add the following statement to *Mod*1:[3]

$$Id \triangleq \text{INSTANCE } Mod2 \text{ WITH } z_1 \leftarrow \overline{z_1}, \ldots, z_p \leftarrow \overline{z_p}$$

The assertion that $Spec_1$ implements $Spec_2$ under the refinement mapping can then be expressed in module *Mod*1 by the following theorem:

$$\text{THEOREM } Spec1 \Rightarrow Id!Spec2$$

This theorem asserts that in any behavior satisfying $Spec_1$, the values of the expressions $\overline{z_i}$ are values that $Spec_2$ permits the variables z_i to have. The shape of the theorem makes it explicit that in TLA$^+$, implementation is implication. The correctness of the theorem can be checked (but seldom completely verified) with TLC for module *Mod*1 having *Spec*1 as the specification and $Id!Spec2$ as the temporal property to be checked.

As we will see, it is sometimes the case that $\exists\, \mathbf{y} : Spec_1$ implements $\exists\, \mathbf{z} : Spec_2$ but there does not exist a refinement mapping under which $Spec_1$ implements $Spec_2$. In that case, it is almost always possible to construct the necessary refinement mapping by adding auxiliary variables to $Spec_1$. Adding auxiliary variables $\mathbf{a}$ to the specification $Spec_1$ means finding a specification $Spec_1^{\mathbf{a}}$ such that $\exists\, \mathbf{a} : Spec_1^{\mathbf{a}}$ is equivalent to $Spec_1$. Showing that $\exists\, \mathbf{y}, \mathbf{a} : Spec_1^{\mathbf{a}}$ implements $\exists\, z : Spec_2$ shows that $\exists\, \mathbf{y} : Spec_1$ implements $\exists\, \mathbf{z} : Spec_2$, since $\exists\, \mathbf{y}, \mathbf{a} : Spec_1^{\mathbf{a}}$ equals $\exists\, \mathbf{y} : \exists\, a : Spec_1^{\mathbf{a}}$ which is equivalent to $\exists\, \mathbf{y} : Spec_1$. Even though we can't define the expressions $\overline{z_i}$ in terms of $\mathbf{x}$ and $\mathbf{y}$, we may be able to define them in terms of $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{a}$.

We will define three kinds of auxiliary variables: history variables that remember what happened in the past, prophecy variables that predict what will happen in the future, and stuttering variables that add stuttering steps (ones that don't change $\mathbf{x}$ and $\mathbf{y}$).

3 History Variables

3.1 Equivalence of *MinMax*1 and *MinMax*2

Let us return to the notation of Section 2.4, so $Spec_1$ is specification *Spec* of module *MinMax*1 and $Spec_2$ is specification *Spec* of module *MinMax*2. We observed that $\exists\, y : Spec_1$ and $\exists\, min, max : Spec_2$ are equivalent, meaning that each implements (implies) the other. We found a refinement mapping under which $Spec_1$ implements $Spec_2$. To prove the converse implication, we want to find a refinement mapping under which $Spec_2$ implements $Spec_1$. This means defining an expression $\overline{y}$ in terms of the variables x, *turn*, *min*, and *max* such

[3] If *Mod*2 has declared CONSTANTS, then the statement must also specify expressions to be substituted for those constants. A substitution of the form $id \leftarrow id$ for an identifier id can be omitted from the WITH clause.

that the values of x, *turn*, and $\overline{y}$ in any behavior allowed by $Spec_1$ are values of x, *turn*, and y allowed by $Spec_2$.

In a behavior of $Spec_1$, the value of y is the set of all values input by the user. However, in a behavior of $Spec_2$, the variables *min* and *max* record only the smallest and largest input values. There is no way to reconstruct the set of all values input from the variables of *MinMax*2. So, there is no refinement mapping under which $Spec_2$ implements $Spec_1$. To solve this problem, we write another spec $Spec_2^h$ that is the same as $Spec_2$, except that it also constrains the behavior of another variable h. More precisely, if we hide h in $Spec_2^h$, then we get a specification that's equivalent to $Spec_2$. Expressed mathematically, this means $\exists\, h : Spec_2^h$ is equivalent to $Spec_2$.

The initial predicate and next-state action of $Spec_2^h$ are the same as those of $Spec_2$, except they also describe the values that h may assume. In particular, the value of h records information about previous values of the variable x, but does not affect the current or future values of x or any of the other variables *turn*, *min*, and *max* of $Spec_2$. Thus $\exists\, h : Spec_2^h$ is equivalent to $Spec_2$. We call h a *history* variable.

We write $Spec_2^h$ as follows in a TLA$^+$ module *MinMax*2*H*. The module begins with the statement

EXTENDS *MinMax*2

that simply imports all the declarations and definitions from *MinMax*, defining *Spec* to be the specification we are calling $Spec_2$. The module declares the variable h and defines the initial predicate *InitH* of $Spec_2^h$ by

$$InitH \triangleq Init \land (h = \{\})$$

The next-state action *NextH* is defined to equal $InputNumH \lor RespondH$ where *InputNumH* and *RespondH* are defined as follows:

$$\begin{aligned} InputNumH &\triangleq \land\ InputNum \\ &\quad\ \land\ h' = h \end{aligned}$$

$$\begin{aligned} RespondH &\triangleq \land\ Respond \\ &\quad\ \land\ h' = h \cup \{x\} \end{aligned}$$

The specification $Spec_2^h$ is the following formula defined in the module:

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

where *varsH* equals $\langle vars, h\rangle$. (Because *vars* equals $\langle x, turn, min, max\rangle$, we can also define *varsH* to equal $\langle x, turn, min, max, h\rangle$; the two definitions give equivalent expressions UNCHANGED *varsH*.)

It's easy to see that this specification asserts that h is always equal to the set of all values that the user has input thus far, which is exactly what $Spec_1$ asserts about y. Therefore, $Spec_2^h$ implements $Spec_1$ under the refinement mapping

MODULE *MinMax2H*

EXTENDS *MinMax2*

VARIABLE h

$$
\begin{aligned}
&\mathit{varsH} \triangleq \langle \mathit{vars}, h \rangle \\
&\mathit{InitH} \triangleq \mathit{Init} \land (h = \{\}) \\
&\mathit{InputNumH} \triangleq \land \mathit{InputNum} \\
&\qquad\qquad\quad\ \land h' = h \\
&\mathit{RespondH} \triangleq \land \mathit{Respond} \\
&\qquad\qquad\quad\ \land h' = h \cup \{x\} \\
&\mathit{NextH} \triangleq \mathit{InputNumH} \lor \mathit{RespondH} \\
&\mathit{SpecH} \triangleq \mathit{InitH} \land \Box[\mathit{NextH}]_{\mathit{varsH}} \\
&M \triangleq \text{INSTANCE } \mathit{MinMax1} \text{ WITH } y \leftarrow h \\
&\text{THEOREM } \mathit{SpecH} \Rightarrow M\,!\,\mathit{Spec}
\end{aligned}
$$

Figure 4: Module *MinMax2H*.

$y \leftarrow h$—that is, with $\overline{y}$ equal to the expression h. We express this in module *MinMax2H* by

$$
\begin{aligned}
&M \triangleq \text{INSTANCE } \mathit{MinMax1} \text{ WITH } y \leftarrow h \\
&\text{THEOREM } \mathit{SpecH} \Rightarrow M\,!\,\mathit{Spec}
\end{aligned}
$$

The complete module *MinMax2H* is in Figure 4

3.2 Disjunctive Representation

The generalization from the *MinMax* example is intuitively clear. We add a history variable h to a specification by conjoining $h = \mathit{exp}_{\mathit{Init}}$ to its initial predicate and $(h' = \mathit{exp}_A)$ to each subaction A of its next-state action, where expression $\mathit{exp}_{\mathit{Init}}$ can contain the spec's variables and each expression exp_A can contain the spec's variables, both primed and unprimed, and h (unprimed). To make this precise, we have to state exactly what a *subaction* is.

In general, there may be many different ways to define the subactions of a next-state action. In defining *SpecH* in module *MinMax2H*, we took *InputNum* and *Respond* to be the subactions of the next-state action *Next* of *MinMax2*. However, we can also consider *Next* itself to be a subaction of *Next*. We can do this and get an equivalent specification *SpecH* by defining *NextH* as follows:

$$
\begin{array}{lcl}
\mathit{NextH} & \triangleq & \land\ \mathit{Next} \\
 & & \land\ \lor\ (\mathit{turn} = \text{“}\mathsf{input}\text{”}) \quad \land\ (h' = h) \\
 & & \quad\ \lor\ (\mathit{turn} = \text{“}\mathsf{output}\text{”}) \land\ (h' = h \cup \{x\})
\end{array}
$$

The two specifications are equivalent because the two definitions of *NextH* are equivalent. Their equivalence is asserted by adding the following theorem to module *MinMax2H*. The TLAPS proof system easily checks its BY proof.

$$
\begin{array}{l}
\text{THEOREM}\ \mathit{NextH} = \land\ \mathit{InputNum} \lor \mathit{Respond} \\
\qquad\qquad\qquad\qquad\ \land\ \lor\ (\mathit{turn} = \text{“}\mathsf{input}\text{”}) \quad \land\ (h' = h) \\
\qquad\qquad\qquad\qquad\quad\ \lor\ (\mathit{turn} = \text{“}\mathsf{output}\text{”}) \land\ (h' = h \cup \{x\}) \\
\text{BY}\ \ \text{DEF}\ \mathit{NextH},\ \mathit{Next},\ \mathit{InputNumH},\ \mathit{RespondH},\ \mathit{InputNum},\ \mathit{Respond}
\end{array}
$$

To define what a subaction is, we introduce the concept of a *disjunctive representation*. A disjunctive representation of a formula N is a way of writing N in terms of subactions $A_1, \ldots, A_m$ using only the operators $\lor$ and $\exists k \in K$, for some identifiers k and expressions K. For example, consider the formula:

$$
\begin{array}{ll}
(3.1) & B \lor C \lor D \lor (\exists i \in S, j \in T : \\
 & \qquad\qquad (\exists q \in U : E) \lor (\exists r \in W : F))
\end{array}
$$

where B, C, D, E, and F can be any formulas. Here is one of the 36 possible disjunctive representations of formula (3.1), where each boxed formula is a subaction:

$$
\boxed{B} \lor \boxed{C \lor D} \lor (\exists i \in S, j \in T : \boxed{(\exists q \in U : E)} \lor (\exists r \in W : \boxed{F}))
$$

In other words, this disjunctive representation of (3.1) has the four subactions B, $C \lor D$, $\exists q \in U : E$, and F.

Each subaction of a disjunction representation has a *context*, which is a pair $\langle \mathbf{k}; \mathbf{K} \rangle$, where $\mathbf{k}$ is an n-tuple of identifiers and $\mathbf{K}$ is an n-tuple of expressions, for some n. The contexts of the subactions in the disjunctive representation of (3.1) defined above are:

subaction	context
B	$\langle\,\rangle$
$C \lor D$	$\langle\,\rangle$
$\exists q \in U : E$	$\langle i, j;\ S, T \rangle$
F	$\langle i, j, r;\ S, T, W \rangle$

We let $\exists \langle i, j;\ S, T \rangle$ be an abbreviation for $\exists i \in S, j \in T$ and similarly for $\forall \langle i, j;\ S, T \rangle$.

The generalization of this example should be clear. The tuple of identifiers in the context of a subaction A are all the bound identifiers of existential quantifiers

within whose scope A lies.[4] If $\langle \mathbf{k}; \mathbf{K} \rangle$ is the empty context $\langle ; \rangle$, we let $\exists \langle \mathbf{k}; \mathbf{K} \rangle : F$ and $\forall \langle \mathbf{k}; \mathbf{K} \rangle : F$ equal F.

We can now define precisely what it means to add a history variable to a specification The definition is contained in the hypothesis of this theorem:

Theorem 1 (History Variable) *Let Spec equal* $Init \wedge \Box[Next]_{vars}$ *and let* $Spec^h$ *equal* $Init^h \wedge \Box[Next^h]_{vars^h}$, *where:*

- $Init^h$ *equals* $Init \wedge (h = exp_{Init})$, *for some expression* exp_{Init} *that may contain the specification's (unprimed) variables.*
- $Next^h$ *is obtained from Next by replacing each subaction A of a disjunctive representation of Next with* $A \wedge (h' = exp_A)$, *for some expression* exp_A *that may contain primed and unprimed specification variables, identifiers in the context of A, and constant parameters.*
- $vars^h$ *equals* $\langle vars, h \rangle$

Then Spec is equivalent to $\exists\, h : Spec^h$.

The hypotheses of this theorem are purely syntactic ones: conditions on the definitions of $Init^h$, $Next^h$, and $vars^h$ plus conditions on what variables and identifiers may appear in exp_{Init} and exp_A. By a variable or identifier appearing in an expression *exp*, we mean that it appears in the expression $[\![exp]\!]$ obtained by expanding all definitions if *exp*. (See the discussion of definition expansion on page 7.)

3.3 Equivalence of Next-State Actions

When adding an auxiliary variable, it is often useful to rewrite a specification *Spec*—that is, to replace *Spec* with a different but equivalent formula. This is most often done by rewriting the next-state action *Next*, which is done by rewriting one or more of the subactions in a disjunctive representation of *Next*. We now consider when we can replace a subaction A in a disjunctive representation of *Next* by the subaction B.

We can obviously replace A by B if A and B are equivalent formulas. However, this is often too stringent a requirement. For example, the two actions

$$(x' = \text{IF } y \geq 0 \text{ THEN } x + y \text{ ELSE } x - y) \wedge (y' = y)$$
$$(x' = \text{IF } y > 0 \text{ THEN } x + y \text{ ELSE } x - y) \wedge (y' = y)$$

are not equivalent. However, they are equivalent if y is a number. Thus, we can replace one by the other in the next-state action if $y \in Int$ is an invariant of the specification. The generalization of this observation is:

[4] Everything we do extends easily to handle unbounded quantification if we pretend that the unbounded quantifiers $\exists v$ and $\forall v$ are written $\exists v \in \Omega$ and $\forall v \in \Omega$, and we define $e \in \Omega$ to equal TRUE for every expression e. Since unbounded quantification seldom occurs in specifications, we will not discuss this further.

Theorem 2 (Subaction Equivalence) *Let A be a subaction with context $\langle \mathbf{k}; \mathbf{K} \rangle$ in a disjunctive representation of the next-state action of a specification Spec with tuple vars of variables, let Inv be an invariant of Spec, and let B be an action satisfying:*

$$(3.2) \quad Inv \;\Rightarrow\; \forall \langle \mathbf{k}; \mathbf{K} \rangle : A \equiv B$$

Then Spec is equivalent to the specification obtained by replacing A with B in the next-state action's disjunctive representation.

Formula (3.2) is an action formula, so it can be proved with TLAPS but cannot be checked with TLC.

TLC can check directly that two specifications are equivalent by checking that each specification implies the other. To check that specification *Spec* implies a specification *SpecB*, just run TLC with a model having *Spec* as the behavioral specification and *SpecB* as the property to be checked. If one spec is obtained by a simple modification of the other, it should suffice to use small models. But in that case, it should not be hard to prove (3.2) with TLAPS, where *Inv* is a simple type invariant.

3.4 Discussion of History Variables

As our example, we showed that specifications *MinMax*1 and *MinMax*2 are equivalent. In practice, we rarely care about checking equivalence of specifications. We almost always want to show that a specification $\mathcal{S}$ satisfies some property $\mathcal{P}$, which means that $\mathcal{S}$ implies $\mathcal{P}$. For example, $\mathcal{S} \Rightarrow \Box Inv$ asserts that *Inv* is an invariant of $\mathcal{S}$. In (2.6) and (2.7), the property $\mathcal{P}$ is, like $\mathcal{S}$, a complete system specification.

The most general form of correctness that TLC can check is that one specification implies another. For example, the assertion that *Inv* is an invariant of a specification $\mathcal{S}$ with tuple *vars* of variables is equivalent to the assertion that $\mathcal{S}$ implies the specification

$$Inv \;\wedge\; \Box[Inv' \equiv Inv]_{vars}$$

We often want to show that a specification $\mathcal{S}$ implies a higher-level, more abstract specification $\mathcal{T}$. The standard way of doing this is to find a refinement mapping that expresses the values of $\mathcal{T}$'s variables as functions of the values of $\mathcal{S}$'s variables. This can't be done if specification $\mathcal{T}$ remembers in its state information that is forgotten by $\mathcal{S}$. In that case, we show that $\mathcal{S}$ implies $\mathcal{T}$ by adding a history variable h to $\mathcal{S}$ to obtain the specification $\mathcal{S}^h$, and we find a refinement mapping to show $\mathcal{S}^h$ implies $\mathcal{T}$. Since $\exists\, h : \mathcal{S}^h$ is equivalent to $\mathcal{S}$, this shows that $\mathcal{S}$ implies $\mathcal{T}$.

One can argue that $\mathcal{T}$ is not a good higher-level specification if it keeps information about the past that doesn't have to be kept by its implementation $\mathcal{S}$.

However, sometimes that information about the past can simplify the higher-level specification. We may also add a history variable to a specification $\mathcal{S}$ so we can state the property we want to show that it satisfies, even if we aren't explicitly constructing a refinement mapping. For example the property that $\mathcal{S}$ requires one kind of action to occur before another can be expressed as an invariant if we add a history variable that remembers when actions have occurred. Because it's a history variable, it doesn't have to be implemented in the system being specified—that is, in the actual hardware or software. Only the variables of $\mathcal{S}$ must be implemented.

3.5 Liveness

A natural liveness requirement for our *MinMax* specs is that every input should produce an output. This requirement is added to both specifications by conjoining the fairness requirement $\mathrm{WF}_{vars}(Respond)$ to the formula *Spec*. (It is just a peculiarity of our example that the fairness requirements are exactly the same in both specifications.) Let us call the resulting specifications *LSpec*.

The two specifications are still equivalent when the internal variables are hidden. Formula (2.5), and hence formulas (2.6) and (2.7), remain true if we replace $Spec_1$ and $Spec_2$ by $LSpec_1$ and $LSpec_2$, respectively—where $LSpec_1$ and $LSpec_2$ are formulas *LSpec* of $MinMax1$ and $MinMax2$, respectively. We verify (2.6) the same as before by verifying the theorem $LSpec \Rightarrow M!LSpec$ of module $MinMax1$. To verify (2.7), we need to add a history variable h to $LSpec_2$ rather than to $Spec_2$. We now drop the subscripts; all the formulas we write will be ones defined in $MinMax2$ or $MinMax2H$.

To add a history variable to the specification *LSpec*, we add a history variable to its safety part and then conjoin the liveness part of *LSpec*. The resulting specification is defined in module $MinMax2H$ by

$$HLSpec \triangleq HSpec \wedge \mathrm{WF}_{vars}(Respond)$$

The equivalence of *LSpec* and $\exists\, h : HLSpec$ follows from the equivalence of *Spec* and $\exists\, h : HSpec$ by this argument:

1. $Spec \wedge \mathrm{WF}_{vars}(Respond) \equiv (\exists\, h : HSpec) \wedge \mathrm{WF}_{vars}(Respond)$
   PROOF: Because *Spec* is equivalent to $\exists\, h : HSpec$.
2. $(\exists\, h : HSpec) \wedge \mathrm{WF}_{vars}(Respond)$
   $\equiv \exists\, h : (HSpec \wedge \mathrm{WF}_{vars}(Respond))$
   PROOF: For any formulas F and G, if h does not occur in G, then $(\exists\, h : F) \wedge G$ is equivalent to $\exists\, h : (F \wedge G)$.
3. Q.E.D.
   PROOF: By definition of *LSpec* and *HLSpec*, 1 and 2 imply that *LSpec* is equivalent to $\exists\, h : HLSpec$.

What we have done for this example generalizes in the obvious way. For a specification written in the canonical form

$$Init \wedge \Box[Next]_{vars} \wedge L$$

with L a liveness condition, we add a history variable by adding it just to the safety part, keeping the same liveness condition. This method of adding a history variable to a specification with liveness produces unusual specifications. In our example, if we expand the definition of *HSpec*, we see that specification *HLSpec* equals

$$InitH \ \wedge \ \Box[NextH]_{varsH} \ \wedge \ \mathrm{WF}_{vars}(Respond)$$

This spec is unusual because it contains a fairness condition on the action *Respond* that is not a subaction of the next-state relation *NextH*. Such specs can be weird. However, specs obtained in this way by adding a history variable are not. Because (i) a *Respond* action is enabled iff a *RespondH* action is and (ii) a *NextH* step is a *Respond* step iff it is a *RespondH* step, specification *HLSpec* is equivalent to the normal specification:

$$InitH \ \wedge \ \Box[NextH]_{varsH} \ \wedge \ \mathrm{WF}_{varsH}(RespondH) \tag{3.3}$$

In general, if $Spec^h$ is obtained from a specification *Spec* by adding a history variable h, then replacing a fairness requirement on a subaction A by the same fairness requirement on A^h produces a specification equivalent to $Spec^h$.

The unusual nature of specification *HLSpec* affects neither the TLC model checker nor our ability to reason about the specification. The same refinement mapping as before shows that *HLSpec* implements *MinMax*1 with the added fairness condition when internal variables are hidden.

4 Prophecy Variables

As we have observed, the fundamental task of verification is to show that the specification $Spec_1$ of an implementation satisfies a specification $Spec_2$ of what the implementation is supposed to do. A history variable remembers the past. It is needed to find a refinement mapping to show that $Spec_1$ implements $Spec_2$ when $Spec_2$ remembers previous events longer than it has to. A prophecy variable is one that predicts the future. It is needed to find a refinement mapping to show that $Spec_1$ implements $Spec_2$ when $Spec_2$ makes decisions before it has to.

4.1 One-Prediction Prophecy Variables

We begin by showing how to add a simple prophecy variable that makes a single prediction at a time. Suppose a disjunctive representation of the next-state relation contains a subaction A such that

$$A \Rightarrow (\exists i \in \Pi : Pred_A(i)) \tag{4.1}$$

for some expression $Pred_A(i)$ and constant set Π. Formula (4.1) is equivalent to

$$(4.2) \quad A \;\equiv\; A \wedge (\exists i \in \Pi : Pred_A(i))$$

which means that any A step is an $A \wedge Pred_A(i)$ step for some i in Π. We introduce a one-prediction prophecy variable p whose value is an i for which the next A step is an $A \wedge Pred_A(i)$ step—if there is a next A step. (There could be more than one such i, since we don't require $Pred_A(i) \wedge Pred_A(j)$ to equal FALSE if $i \neq j$.) We give p that meaning by replacing the subaction A with a subaction A^p defined by

$$(4.3) \quad A^p \;\triangleq\; A \wedge Pred_A(p) \wedge Setp$$

where $Setp$ determines the value of p'.

To ensure that adding the prophecy variable p allows all the behaviors of the other variables that the original spec does, we must ensure that p can always have any value in Π. We do this by initializing p to an arbitrary element of Π and changing p only by setting it to any arbitrary element of Π. Thus we modify the spec's initial predicate $Init$ to equal $Init \wedge (p \in \Pi)$ and we let $Setp$ equal $p' \in \Pi$, so

$$(4.4) \quad A^p \;\triangleq\; A \wedge Pred_A(p) \wedge (p' \in \Pi)$$

For another subaction A of the next-state relation whose effect is not being predicted by p, we let A^p leave the prediction unchanged, so it is defined simply as:

$$(4.5) \quad A^p \;\triangleq\; A \wedge (p' = p)$$

We illustrate this with a simple example: a system in which integers are sent and received, where sending an integer i is represented by setting the variable x to i, and receiving a value is represented by setting x to a value $NotInt$ that is not an integer. Our specification $SendInt2$ has the receiving action set an internal variable z to the next value to be sent. (The initial value of z is the first value to be sent.) This simple specification is in Figure 5.

Of course, we can describe the behavior of the variable x even more simply, without using any internal variable. Such a specification is in module $SendInt1$ of Figure 6. Let $Spec_1$ and $Spec_2$ be the formulas $Spec$ of modules $SendInt1$ and $SendInt2$, respectively. It should be obvious that $Spec_1$ is equivalent to $\exists\, z : Spec_2$. To verify $(\exists\, z : Spec_2) \Rightarrow Spec_1$, we just have to verify $Spec_2 \Rightarrow Spec_1$, which TLC easily checks for a model that substitutes a finite set of integers for Int. We now show how to verify $Spec_1 \Rightarrow (\exists\, z : Spec_2)$.

There is obviously no refinement mapping under which $Spec_1$ implements $Spec_2$. Such a refinement mapping would be an expression $\overline{z}$ involving only the variable x so that $\overline{z}$ is the value of z in any state satisfying $Spec_2$. This is

MODULE *SendInt*2

EXTENDS *Integers*

$NotInt \triangleq$ CHOOSE $n : n \notin Int$

VARIABLE x, z

$$
\begin{aligned}
Init &\triangleq \land x = NotInt \\
&\quad \land z \in Int \\
Send &\triangleq \land x = NotInt \\
&\quad \land x' = z \\
&\quad \land z' = NotInt \\
Rcv &\triangleq \land x \in Int \\
&\quad \land x' = NotInt \\
&\quad \land z' \in Int \\
Next &\triangleq Send \lor Rcv \\
Spec &\triangleq Init \land \Box[Next]_{\langle x, z \rangle}
\end{aligned}
$$

Figure 5: Specification *SendInt*2.

MODULE *SendInt*1

EXTENDS *Integers*

$NotInt \triangleq$ CHOOSE $n : n \notin Int$

VARIABLE x

$$
\begin{aligned}
Init &\triangleq x = NotInt \\
Send &\triangleq \land x = NotInt \\
&\quad \land x' \in Int \\
Rcv &\triangleq \land x \in Int \\
&\quad \land x' = NotInt \\
Next &\triangleq Send \lor Rcv \\
Spec &\triangleq Init \land \Box[Next]_x
\end{aligned}
$$

Figure 6: Specification *SendInt1*

impossible, since z could equal any integer in a state in which x equals $NotInt$, so there is no way to express its value as a function of the value of x. The variable z of $SendInt2$ is used to predict the value to be sent before it actually is sent. To be able to define the value of $\overline{z}$ for a refinement mapping, we add a prophecy variable p to $SendInt1$ that predicts what the next value to be sent is.

The prophecy variable p must predict the value sent by action $Send$ of $SendInt1$. Therefore $SendP$ must have the form of (4.4). A little thought shows that p makes the right prediction if we take $Pred_{Send}(p)$ to equal $x' = p$. Since TLA$^+$ doesn't allow identifiers to have subscripts we write $PredSend$ instead of $Pred_{Send}$ and define

$$PredSend(i) \triangleq x' = i$$

Condition (4.1) becomes

$$Send \Rightarrow \exists i \in \Pi : PredSend(i)$$

which is obviously true by definition of $Send$, if we let Π equal Int. Writing Pi instead of Π and $SendP$ instead of $Send^p$, we make the definitions:

$$Pi \triangleq Int$$
$$SendP \triangleq Send \land PredSend(p) \land (p' \in Pi)$$

(We could of course simply write Int instead of Pi, but writing Pi will help us understand what's going on.)

For the receive action, Rcv^p should have the form (4.5).

In a behavior of $SendInt2$, when x equals $NotInt$, the value of z is the next value sent; and when x equals an integer (the value sent), then z equals $NotInt$. Therefore, if $SpecP$ is the specification obtained from specification $Spec$ of $SendInt1$ by adding the prophecy variable p, then $SpecP$ implements the specification $Spec$ of $SendInt2$ under this refinement mapping:

$$\overline{z} \leftarrow \text{IF } x = NotInt \text{ THEN } p \text{ ELSE } NotInt$$

Note that $SpecP$ predicts the next value to be sent even before the $SendInt2$ specification does—when the previous value is sent rather than when it is received. Although it's not necessary, we'll see later how we could defer the prediction until the Rcv action is executed.

The complete specification is contained in module $SendInt1P$ in Figure 7. Observe that we defined $PredSend$ before the declaration of p to ensure that $PredSend$ is not defined in terms of p.

The module's theorem asserts that $SpecP$ implements formula $Spec$ of $SendInt2$ under the refinement mapping defined above. It can be checked with TLC by creating a model with temporal specification $SpecP$ and having it check the temporal property $SI2!Spec$. The model will have to substitute a finite set of integers for Int. The specification is very simple and doesn't depend on any properties of integers, so substituting a set with a few numbers will ensure that we didn't make a mistake.

MODULE *SendInt1P*

EXTENDS *SendInt1*

$Pi \triangleq Int$

$PredSend(i) \triangleq x' = i$

VARIABLE p

$varsP \triangleq \langle x, p \rangle$

$InitP \triangleq Init \land (p \in Pi)$

$SendP \triangleq Send \land PredSend(p) \land (p' \in Pi)$

$RcvP \triangleq Rcv \land (p' = p)$

$NextP \triangleq SendP \lor RcvP$

$SpecP \triangleq InitP \land \Box[NextP]_{varsP}$

$SI2 \triangleq$ INSTANCE *SendInt2* WITH $z \leftarrow$ IF $x = NotInt$ THEN p ELSE $NotInt$

THEOREM $SpecP \Rightarrow SI2!Spec$

Figure 7: Specification *SendInt1P*

4.2 One-Prediction Prophecy Variables in General

We generalize our description of a one-prediction prophecy variable in two ways. First, we can allow a prophecy variable to make predictions about more than one action by replacing more than one subaction A of a disjunctive representation by an action A^p of the form (4.3). If we do this for subactions $A_1, A_2, \ldots$, then the value of p makes a prediction about the next step to occur that is an A_1 or A_2 or ... step. We can express this a little more elegantly by generalizing (4.3) to allow *Setp* to depend on A and then letting each action A of the disjunctive representation be replaced with an action A^p defined by

(4.6) $A^p \triangleq A \land Pred_A(p) \land Setp_A$

For an action A about which no prediction is being made, $Pred_A(p)$ is the expression TRUE. We can then replace (4.4) and (4.5) by defining $Setp_A$ to be one of the following:

(4.7) (a) $Setp_A \triangleq p' = p$

(b) $Setp_A \triangleq p' \in \Pi$

where possibility (a) is allowed only if $Pred_A(p)$ is the expression TRUE (so p is making no prediction about A). This is more general because it allows an action that doesn't use the prediction made by p to make a new prediction.

Our second generalization is needed to handle subactions of a disjunctive representation having a nonempty context. For a subaction A with context $\langle \mathbf{k}; \mathbf{K} \rangle$, condition (4.1) contains the identifiers $\mathbf{k}$. That condition need only hold for values of those identifiers in the corresponding set in $\mathbf{K}$. Thus (4.1) can be generalized to

$$(4.8) \quad \forall \langle \mathbf{k}; \mathbf{K} \rangle : A \Rightarrow (\exists i \in \Pi : Pred_A(i))$$

Condition (4.8) is a condition on pairs of states. It needn't hold for all pairs of states, only for pairs of states that can occur in a behavior satisfying the original specification *Spec*. We can therefore replace (4.8) by the requirement[5]

$$(4.9) \quad Spec \;\Rightarrow\; \Box[\,\forall \langle \mathbf{k}; \mathbf{K} \rangle : A \Rightarrow (\exists i \in \Pi : Pred_A(i)\,)]_{vars}$$

TLC can check this condition with a model having the temporal formula *Spec* as its behavioral spec and

$$\Box[\forall \langle \mathbf{k}; \mathbf{K} \rangle : A \Rightarrow (\,\exists i \in \Pi : Pred_A(i)\,)]_{vars}$$

as a property to be checked.

4.3 Prophecy Array Variables

Our next example is based on one created by Martín Abadi [1]. It is similar to our *SendInt* specifications in that a sender sends a value v to a receiver with a *Send* action that sets the variable x to v, and the receiver receives the value by resetting x. Instead of sending integers, the values sent are elements of an unspecified constant set *Data*, and we let the initial value of x be a value *NonData* not in *Data*. A variable y contains a set of values to be sent. Those values are chosen by a *Choose* action, which adds a new data element to y. The high-level specification is formula *Spec* in module *SendSet* of Figure 8. We consider the variable x to be externally visible and y to be internal.

Our implementation adds to the specification of *SendSet* an *undo* operation that removes elements from y. Abadi reports that this example is a highly simplified abstraction of a real system in which the implementation contains an undo operation not present in the specification.

The implementation specification is formula *SpecU* in module *SendSetUndo* of Figure 9. Its initial predicate is the same as the initial predicate of the specification *Spec* of module *SendSet*, and its next-state action *NextU* is the same as the next-state action *Next* of that module except it allows *Undo*(S) steps that remove from y an arbitrarily chosen non-empty subset S of y.

[5] Formula (4.9) does not imply that (4.8) is true for stuttering steps that are allowed by A. It can be shown that this doesn't matter, and condition (4.9) is strong enough.

MODULE *SendSet*

CONSTANT $Data$

$NonData \triangleq \text{CHOOSE } d : d \notin Data$

VARIABLES $x,\ y$

$vars \triangleq \langle x,\ y \rangle$

$Init \triangleq (x = NonData) \land (y = \{\})$

$$Choose \triangleq \land \exists\, d \in Data \setminus y : y' = y \cup \{d\} \\ \land x' = x$$

$$Send \triangleq \land x = NonData \\ \land x' \in y \\ \land y' = y \setminus \{x'\}$$

$$Rcv \triangleq \land x \in Data \\ \land x' = NonData \\ \land y' = y$$

$Next \triangleq Choose \lor Send \lor Rcv$

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

Figure 8: Specification *SendSet*.

MODULE *SendSetUndo*

EXTENDS *SendSet*

$$Undo(S) \triangleq \land y' = y \setminus S \\ \land x' = x$$

$NextU \triangleq Next \lor (\exists\, S \in (\text{SUBSET } y) : Undo(S))$

$SpecU \triangleq Init \land \Box[NextU]_{vars}$

Figure 9: Specification *SendSetUndo*.

It's clear that the specifications *Spec* of module *SendSet* and *SpecU* of module *SendSetUndo* allow the same behaviors of the variable x. Hence, $\exists\, y : Spec$ and $\exists\, y : SpecU$ are equivalent. It's easy to show that *Spec* implements *SpecU* under the identity refinement mapping in which $\overline{y}$ is defined to equal y, since *NextU* allows all steps allowed by *Next*. This implies that $\exists\, y : Spec$ implies $\exists\, y : SpecU$. To construct a refinement mapping under which *SpecU* implements *Spec*, we must define $\overline{y}$ so that it contains a data value d iff that value is going to be sent by a *Send* step rather than being removed from y by an *Undo* step. This involves predicting, when d is added to y, whether it will later be sent or "undone".

We add a prophecy array variable p to *SpecU* that makes this prediction, setting $p[d]$ to either "send" or "undo" when d is added to y. So, we define our set *Pi* of possible predictions by

$$Pi \;\triangleq\; \{\text{“send”}, \text{“undo”}\}$$

The value of p in every state will be a function with domain equal to the set y, with $p[d] \in Pi$ for all $d \in y$. In other words $p \in [y \to Pi]$ will be an invariant of the spec *SpecUP* obtained by adding the prophecy variable p. The variable p is therefore making a predication $p[d]$ for every d in y, so p is making an array of prophecies. (This is a "dynamic" array, because the value of y can change.)

We now define the specification *SpecUP*. As with a one-prediction prophecy variable, we obtain the next-state relation of *SpecUP* by replacing each subaction A in a disjunctive representation of the next-state relation with a new action A^p. Instead of defining A^p as in (4.6), we define it to equal

$$(4.10)\quad A^p \;\triangleq\; A \wedge Pred_A(p) \wedge (p' \in NewPSet_A)$$

for suitable expressions $Pred_A(p)$ and $NewPSet_A$. We need a condition corresponding to condition (4.9) for a one-prediction prophecy variable to assert that there is a possible value of p that makes $Pred_A(p)$ true. With an array prophecy variable, p is no longer an element of *Pi* but a function in $[Dom \to Pi]$ for some domain *Dom* that can change. In our example, *Dom* equals y. To make the generalization to an arbitrary spec easier, we define *Dom* to equal y and write *Dom* instead of y where appropriate. For our example, we can replace (4.9) with

$$(4.11)\quad SpecU \;\Rightarrow\; \Box[\forall\langle \mathbf{k}; \mathbf{K}\rangle \,:\, A \Rightarrow (\,\exists f \in [Dom \to \Pi] : Pred_A(f)\,)]_{vars}$$

where $\langle \mathbf{k}; \mathbf{K}\rangle$ is the context of A. (Remember that for the empty context $\langle\ ;\ \rangle$, we define $\forall\langle\ ;\ \rangle : F$ to equal F, for any formula F.)

We now define the formulas $Pred_A$ and $NewPSet_A$ for the disjunctive representation of *NextU* with subactions *Choose*, *Send*, *Rcv*, and *Undo*(S). The context of the first three subactions is empty; the context of *Undo*(S) is $\langle S;$ SUBSET $y\rangle$.

The variable p does not make any prediction about the *Choose* action, so $Pred_{Choose}(p)$ should equal TRUE. The action adds an element d to its domain,

so $Choose^p$ must allow $p'[d]$ to equal any element of Pi. For any element d in the domain Dom of p, the value of $p[d]$ can be left unchanged. Our definitions of $Pred_{Choose}(p)$ and $NewPSet_{Choose}(p)$ are then

$$\begin{array}{ll} PredChoose(p) & \triangleq \text{ TRUE} \\ NewPSetChoose(p) & \triangleq \{ f \in [Dom' \to Pi] \ : \ \forall\, d \in Dom : f[d] = p[d] \} \end{array}$$

The prophecy variable p should predict that if the next action is a *Send* action, then it sends a value d in Dom such that $p[d] =$ "send". The value sent by the action is x', so we define

$$PredSend(p) \ \triangleq \ p[x'] = \text{“send”}$$

The *Send* action removes the sent element d from Dom, thus erasing the prediction p made about d. The value of $p[d]$ is left unchanged for all other elements d in Dom. Thus $NewPSet_{Send}(p)$ is defined as follows to be a set consisting of a single function

$$NewPSetSend(p) \ \triangleq \ \{ [d \in Dom' \mapsto p[d]] \}$$

No prediction is made about the *Rcv* action, and it doesn't change Dom, so we have:

$$\begin{array}{ll} PredRcv(p) & \triangleq \text{ TRUE} \\ NewPSetRcv(p) & \triangleq \{p\} \end{array}$$

The $Undo(S)^p$ action should be enabled only when p has predicted that all the elements in S will not be sent—in other words, when $\forall d \in S : p[d] =$ "undo" is true. Since the identifier S appears in this formula, it must be an argument of the definition of $Undo(S)^p$. Thus, we define

$$PredUndo(p, S) \ \triangleq \ \forall d \in S \ : \ p[d] = \text{“undo”}$$

The $Undo(S)$ action removes from Dom all the elements for which p made a prediction about $Undo(S)$—namely, all the elements of S. We can define $NewPSet_{Undo(S)}$ the same way we defined $PSetSend_{Send}$, without explicitly mentioning S:

$$NewPSetUndo(p) \ \triangleq \ \{ [d \in Dom' \mapsto p[d]] \}$$

We can now declare the variable p and define the specification *SpecUP* by defining the initial predicate *InitUP* and defining the next-state relation *NextUP* in terms of the subactions A^p, using (4.10). The complete specification is in module *SendSetUndoP* shown in Figure 10. Note that the initial value of p is the unique function whose domain is the empty set. We could write that function as $[d \in \{\} \mapsto exp]$ for any expression *exp*—for example, 42. However, it's easier to write that function as $\langle\,\rangle$ (the empty sequence).

We can now define the refinement mapping under which *SpecUP* implements specification *Spec* of module *SendSet*. The refinement mapping defines $\overline{y}$ to equal the set of elements d in y with $p[d] =$ "send". We thus add to the module

MODULE *SendSetUndoP*

EXTENDS *SendSetUndo*

$Pi \triangleq \{\text{“send”}, \text{“undo”}\}$
$Dom \triangleq y$

$PredChoose(p) \triangleq \text{TRUE}$
$NewPSetChoose(p) \triangleq \{f \in [Dom' \to Pi] \ : \ \forall\, d \in Dom : f[d] = p[d]\}$

$PredSend(p) \triangleq p[x'] = \text{“send”}$
$NewPSetSend(p) \triangleq \{[d \in Dom' \mapsto p[d]]\}$

$PredRcv(p) \triangleq \text{TRUE}$
$NewPSetRcv(p) \triangleq \{p\}$

$PredUndo(p,\, S) \triangleq \forall\, d \in S : p[d] = \text{“undo”}$
$NewPSetUndo(p) \triangleq \{[d \in Dom' \mapsto p[d]]\}$

VARIABLE p
$varsP \triangleq \langle vars,\, p\rangle$

$InitUP \triangleq Init \land (p = \langle\rangle)$

$ChooseP \triangleq Choose \land PredChoose(p) \land (p' \in NewPSetChoose(p))$
$SendP \triangleq Send \land PredSend(p) \land (p' \in NewPSetSend(p))$
$RcvP \triangleq Rcv \land PredRcv(p) \land (p' \in NewPSetRcv(p))$
$UndoP(S) \triangleq Undo(S) \land PredUndo(p,\, S) \land (p' \in NewPSetUndo(p))$

$NextUP \triangleq ChooseP \lor SendP \lor RcvP \lor (\exists\, S \in \text{SUBSET } y : UndoP(S))$

$SpecUP \triangleq InitUP \land \Box[NextUP]_{varsP}$

Figure 10: Specification *SendSetUndoP*

$SS \triangleq \text{INSTANCE } SendSet \text{ WITH } y \leftarrow \{d \in y : p[d] = \text{“send”}\}$
THEOREM $SpecUP \Rightarrow SS!Spec$

We can have TLC check this theorem by creating a model having *SpecUP* as the behavior spec and checking the property $SS!Spec$.

We should also check that condition (4.11) holds for each subaction A. To do this, we need to create a model with specification *SpecU* and have TLC check the property:

$$\begin{aligned}\Box[\ &\land Choose \Rightarrow \exists f \in [Dom \to Pi] : PredChoose(f)\\ &\land Send \ \ \Rightarrow \exists f \in [Dom \to Pi] : PredSend(f)\\ &\land Rcv \ \ \ \Rightarrow \exists f \in [Dom \to Pi] : PredRcv(f)\end{aligned}$$

$$\land \forall\, S \in \text{SUBSET } y :$$
$$\quad Undo(S) \Rightarrow \exists f \in [Dom \to Pi] : PredUndo(f,\, S)$$
$$]_{vars}$$

However, there is a problem in doing this. TLC will not allow a model for module *SendSetUndoP* to have behavior specification *SpecU* because that spec doesn't describe the behavior of the variable p. We can solve this problem by modifying the specification—temporarily inserting "======" into the module before the declaration of p and then creating the necessary model. Alternatively, we can move all the definitions before the declaration of p into module *SendSetUndo* and check the condition in a model for that spec. This is inelegant because those definitions aren't part of the *SendSetUndo* specification. The proper solution is to move those definitions from *SendSetUndoP* and put them in a new module that extends *SendSetUndo* and is extended by *SendSetUndoP*. We can then check that the condition is satisfied using a model for that specification that has behavior spec *SpecU*. We won't bother doing this, instead putting them in module *SendSetUndoP*.

Recall that in the *SendInt* example of Section 4.1, the one-prediction prophecy variable we used predicted the next value to be sent when the previous value was sent, while the specification *SendInt*2 didn't choose the next value to be sent until the previous value was received. We can use an array prophecy variable to defer the prediction until it's needed. We let the domain *Dom* of p initially contain a single element—let's take that element to be "on". We let the *Send* action set *Dom* to the empty set, and we let the *Rcv* action set *Dom* to {"on"}.

4.4 Prophecy Data Structure Variables

It is easy to generalize from the *SendSet* example to an arbitrary prophecy-array variable. However, it is useful to generalize still further from an array to an arbitrary data structure. These prophecy data structure variables are the most general ones that we consider. We propose them as the standard way of defining prophecy variables in TLA$^+$, and we have created a module with definitions that simplify adding these prophecy variables. Both single-prophecy variables and prophecy array variables are easily expressed as special cases.

As our example of a prophecy data structure variable, we modify the specification *SendSet*, in which a set of items to be sent is chosen, to a specification *SendSeq* in which a sequence of items is chosen. The value of the variable y is changed from a set of data items to a sequence of data items. The next item to be sent is the one at the head of y, and each value chosen is appended to the tail of y. The specification is in module *SendSeq*, shown in Figure 11.

For our implementation, we add an undo action that removes an arbitrary element from the sequence y. The specification is in module *SendSeqUndo* of

MODULE *SendSeq*

EXTENDS *Sequences*, *Integers*

CONSTANT *Data*

$NonData \triangleq$ CHOOSE $v : v \notin Data$

VARIABLES x, y
$vars \triangleq \langle x, y \rangle$

$Init \triangleq (x = NonData) \land (y = \langle\rangle)$

$$Choose \triangleq \land \exists\, d \in Data : y' = Append(y,\, d) \\ \land x' = x$$

$$Send \triangleq \land x = NonData \land y \neq \langle\rangle \\ \land x' = Head(y) \\ \land y' = Tail(y)$$

$$Rcv \triangleq \land x \in Data \\ \land x' = NonData \\ \land y' = y$$

$Next \triangleq Choose \lor Send \lor Rcv$

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

Figure 11: Specification *SendSeq*

Figure 12. It defines $RemoveEltFrom(i, seq)$ to be the sequence obtained from a sequence seq by removing its i^{th} element, assuming $1 \leq i \leq Len(seq)$.

As before, we want to show that specification *SendU* of module *SendSeqUndo* implements $\exists\, y : Spec$, where *Spec* is the specification of module *SendSeq*. Again, we need to add a prophecy variable p that predicts whether each element of y will be sent or "undone". We do this by having p be an element of $Seq(\{\text{“send”}, \text{“undo”}\})$ that has the same length as y. The $Choose^p$ action should append either "send" or "undo" to the tail of p, the $Send^p$ action should remove the head of p, and the $Undo(i)$ action should remove the i^{th} element of p.

As we did for the *SendSet* example, we write a module *SendSeqUndoP* that extends *SendSeqUndo*. In it, we define the formulas $Pred_A(p)$ of (4.10) for each of the subactions *Choose*, *Send*, *Rcv*, and $Undo(i)$. The definitions are:

$$\begin{array}{ll} PredChoose(p) & \triangleq \text{ TRUE} \\ PredSend(p) & \triangleq p[1] = \text{“send”} \\ PredRcv(p) & \triangleq \text{ TRUE} \\ PredUndo(p,\, i) & \triangleq p[i] = \text{“undo”} \end{array}$$

MODULE *SendSeqUndo*

EXTENDS *SendSeq*

$RemoveEltFrom(i,\ seq) \triangleq [j \in 1\,..\,(Len(seq) - 1) \mapsto \text{IF } j < i \text{ THEN } seq[j] \text{ ELSE } seq[j+1]]$

$Undo(i) \triangleq \land\ y' = RemoveEltFrom(i,\ y)$
$\qquad\qquad\ \land\ x' = x$

$NextU \triangleq Next \lor (\exists\, i \in 1\,..\,Len(y) : Undo(i))$

$SpecU \triangleq Init \land \Box[NextU]_{vars}$

Figure 12: Specification *SendSeqUndo*

We now need to define the expression $NewPSet_A$ of (4.10) for each of these subactions.

Since a sequence of length n is a function with domain $1\,..\,n$, the value of p is a function—just as in *SendSetUndoP* above. This time its domain Dom is the set $1\,..\,Len(y)$. However, in that example, if d is in the domain Dom of p in two successive states, then $p[d]$ represents the same prediction in both states. This isn't true in the current example. If $s \to t$ is a *Send* step and $Len(p) > 1$ in state s, then the prediction made by $p[2]$ in state s is the prediction made by $p[1]$ in state t. If $s \to t$ is an $Undo(i)$ step and $j > i$, the prediction made by $p[j]$ in state s is the prediction made by $p[j-1]$ in state t.

In general, an action A defines a correspondence between some elements in the domain Dom of p and elements in the domain Dom' of p'. For example, for the *Send* action, each element $i > 1$ in Dom corresponds to the element $i - 1$ of Dom'. The formula $p' \in NewPSet_A$ in (4.10) should ensure that if an element d of Dom' either corresponds to an element c of Dom that makes a prediction about A or else does not correspond to any element of Dom, then $p'[d]$ can assume any value in Π; but if d corresponds to an element c of Dom that make no prediction about A, then $p'[d]$ equals $p[c]$. Instead of defining the formulas $NewPSet_A$ directly, we will define them in terms of the correspondence between elements of Dom and Dom' made by A and the set of elements d in Dom for which $p[d]$ makes a prediction about A.

To express formally a correspondence between elements of Dom and Dom', we introduce the concept of a partial injection. A *partial function* from a set U to a set V is a function from a subset of U to V. In other words, it is an element of $[D \to V]$ for some subset D of U. (Remember that U is a subset of itself.) An *injection* is a function that maps different elements in its domain to different values. In other words, a function f is an injection iff for all a and b in DOMAIN f, if $a \neq b$ then $f[a] \neq f[b]$. The set of all partial injections from U to

V is defined in TLA$^+$ by

$$\begin{array}{l} PartialInjections(U,\ V) \triangleq \\ \quad \text{LET } PartialFcns \triangleq \text{UNION } \{[D \to V] : D \in \text{SUBSET } U\} \\ \quad \text{IN } \ \{f \in PartialFcns : \forall\, a,\ b \in \text{DOMAIN } f : (a \neq b) \Rightarrow (f[a] \neq f[b])\} \end{array}$$

For each subaction A, we define a partial injection $DomInj_A$ from Dom to Dom' such that an element c of Dom corresponds to an element d of Dom' iff c is in the domain of $DomInj_A$ and $d = DomInj_A[c]$. Here are the definitions for the four subactions, which are put in module *SendSeqUndoP*:

$$\begin{array}{ll} DomInjChoose & \triangleq\ [d\ \in Dom \mapsto d] \\ DomInjSend & \triangleq\ [i\ \in 2\,..\,Len(y) \mapsto i-1] \\ DomInjRcv & \triangleq\ [d \in Dom \mapsto d] \\ DomInjUndo(i) & \triangleq\ [j\ \in 1\,..\,Len(y) \setminus \{i\} \mapsto \text{IF } j < i \text{ THEN } j \text{ ELSE } \ j-1] \end{array}$$

For the prophecy array variable described in Section 4.3 above, the function $DomInj_A$ maps each element d in Dom that is also in Dom' to itself. Thus, for each subaction A used in defining a prophecy array variable, we can define

$$DomInj_A \ \triangleq\ [d \in Dom \cap Dom' \mapsto d]$$

A function f such that $f[x] = x$ for all $x \in \text{DOMAIN } f$ is called an *identity* function. For convenience, the *Prophecy* module defines $IdFcn(S)$ to be the identify function with domain S. But we won't bother using it here. The *Prophecy* module also defines *EmptyFcn* to be the (unique) function whose domain is the empty set.

Let us return to our prophecy data structure example. For a subaction A, we can define $NewPSet_A$ in terms of $DomInj_A$ and the subset $PredDom_A$ of Dom, which consists of the elements in Dom such that $p[d]$ makes a prediction about A. We define these sets $PredDom_A$ in module *SendSeqUndoP* for our four subactions as follows:

$$\begin{array}{ll} PredDomChoose & \triangleq\ \{\} \\ PredDomSend & \triangleq\ \{1\} \\ PredDomRcv & \triangleq\ \{\} \\ PredDomUndo(i) & \triangleq\ \{i\} \end{array}$$

(Since $PredDom_{Undo(i)}$ depends on the identifier i in its context, we must define *PredDomUndo* to have a parameter.)

We can define $NewPSet_A$ to equal the set of all functions q in $[Dom' \to \Pi]$ such that for every element d in Dom that is not in $PredDom_A$ and has a corresponding element $DomInj_A[d]$ in Dom', the value of q on that corresponding element equals $p[d]$. More precisely, $NewPSet_A$ equals:

$$\begin{array}{l} \{\, q \in [Dom' \to \Pi] \ : \\ \qquad \forall d \in (\text{DOMAIN } DomInj_A) \setminus PredDom_A \ :\ q[DomInj_A[d]] = p[d] \,\} \end{array}$$

We encapsulate definitions like this in a module *Prophecy*. We find it most convenient to make this a constant module with constant parameters *Pi*, *Dom*, and *DomPrime*. This module is meant to be instantiated with the parameter *Pi* instantiated by Π, with the parameter *Dom* instantiated by the appropriate state function *Dom*, and with *DomPrime* instantiated by Dom'. The following definition from module *Prophecy* allows us to define $NewPSet_A(p)$ to equal $NewPSet(p, DomInj_A, PredDom_A)$:

$$\begin{array}{l} NewPSet(p,\ DomInj,\ PredDom) \triangleq \\ \quad \{\ q \in [DomPrime \to Pi] : \\ \qquad \forall\, d \in (\text{DOMAIN}\ DomInj) \setminus PredDom : q[DomInj[d]] = p[d]\ \} \end{array}$$

For each action in A in our disjunctive decomposition of *NextU*, we have written the definitions of $Pred_A$, $DomInj_A$, and $PredDom_A$. This allows us to define $NewPSet_A$, and therefore, by (4.10), to define A^p. The following operator *ProphAction* from the *Prophecy* module allows us to write A^p as $ProphAction(A, p, p', DomInj_A, PredDom_A, Pred_A)$:

$$\begin{array}{l} ProphAction(A,\ p,\ pPrime,\ DomInj,\ PredDom,\ \ Pred(\_)) \triangleq \\ \quad A\ \wedge\ Pred(p)\ \wedge\ (pPrime \in NewPSet(p,\ DomInj,\ PredDom)) \end{array}$$

In module *SendSeqUndoP*, we can define

$$\begin{array}{l} ChooseP\ \triangleq\ ProphAction(Choose,\ p,\ p',\ DomInjChoose, \\ \qquad\qquad\qquad\qquad\qquad PredDomChoose,\ PredChoose) \end{array}$$

The definitions of *SendP* and *RcvP* are similar. However, there is a problem with the definition of $Undo(i)^p$, which we write as $UndoP(i)$. Operator *ProphAction* requires its last argument, which represents $Pred_A$, to be an operator with a single argument. However, we defined *PredUndo* to have two arguments: p and its context identifier i. Since we are defining $UndoP(i)$, the fifth argument has to be an operator *Op* so that $Op(p)$ equals $PredUndo(p, i)$. So, we should define:

$$\begin{array}{l} UndoP(i)\ \triangleq\ \text{LET}\ Op(j)\ \triangleq\ PredUndo(j,\ i) \\ \qquad\qquad\qquad\quad \text{IN}\quad ProphAction(Undo(i),\ p,\ p',\ DomInjUndo(i), \\ \qquad\qquad\qquad\qquad\qquad\qquad PredDomUndo(i),\ Op) \end{array}$$

Using the TLA$^+$ LAMBDA construct (added since *Specifying Systems* was published), this can also be written as:

$$\begin{array}{l} UndoP(i)\ \triangleq \\ \quad ProphAction(\ Undo(i),\ p,\ p',\ DomInjUndo(i),\ PredDomUndo(i), \\ \qquad\qquad\qquad \text{LAMBDA}\ j : PredUndo(j,\ i)\ ) \end{array}$$

It's now straightforward to complete our definition of specification *SpecUP*. Doing so, gathering up the definitions made or implied so far, and rearranging them a bit, we get the beginning of module *SendSeqUndoP* shown in Figure 13.

MODULE *SendSeqUndoP*

EXTENDS *SendSeqUndo*

$Pi \triangleq \{\text{“send”}, \text{“undo”}\}$
$Dom \triangleq \text{DOMAIN } y$

INSTANCE *Prophecy* WITH $DomPrime \leftarrow Dom'$

$PredDomChoose \triangleq \{\}$
$DomInjChoose \triangleq [d \in Dom \mapsto d]$
$PredChoose(p) \triangleq \text{TRUE}$

$PredDomSend \triangleq \{1\}$
$DomInjSend \triangleq [i \in 2 \,..\, Len(y) \mapsto i - 1]$
$PredSend(p) \triangleq p[1] = \text{“send”}$

$PredDomRcv \triangleq \{\}$
$DomInjRcv \triangleq [d \in Dom \mapsto d]$
$PredRcv(p) \triangleq \text{TRUE}$

$PredDomUndo(i) \triangleq \{i\}$
$DomInjUndo(i) \triangleq [j \in 1 \,..\, Len(y) \setminus \{i\} \mapsto \text{IF } j < i \text{ THEN } j \text{ ELSE } j - 1]$
$PredUndo(p,\, i) \triangleq p[i] = \text{“undo”}$

VARIABLE p
$varsP \triangleq \langle vars,\, p\rangle$

$InitUP \triangleq Init \wedge (p \in [Dom \to Pi])$

$ChooseP \triangleq ProphAction(Choose,\, p,\, p',$
$\qquad DomInjChoose,\, PredDomChoose,\, PredChoose)$

$SendP \triangleq ProphAction(Send,\, p,\, p',\, DomInjSend,\, PredDomSend,\, PredSend)$

$RcvP \triangleq ProphAction(Rcv,\, p,\, p',\, DomInjRcv,\, PredDomRcv,\, PredRcv)$

$UndoP(i) \triangleq ProphAction(Undo(i),\, p,\, p',\, DomInjUndo(i),\, PredDomUndo(i),$
$\qquad \text{LAMBDA } j : PredUndo(j,\, i))$

$NextUP \triangleq ChooseP \vee SendP \vee RcvP \vee (\exists\, i \in 1 \,..\, Len(y) : UndoP(i))$

$SpecUP \triangleq InitUP \wedge \Box[NextUP]_{varsP}$

Figure 13: The specification of *SpecUP*.

Finally, we have to define the refinement mapping under which *SpecUP* implements specification *Spec* of module *SendSeq*. The idea is simple: we let $\overline{y}$ be the subsequence of y containing only those elements for which the corresponding element of the sequence p equals "send". The following formal definition is a bit tricky. It uses a local recursive definition of an operator R such that if *yseq* is any sequence and *pseq* is a sequence of the same length, then $R(yseq, pseq)$ is the subsequence of *yseq* that contains $yseq[i]$ iff $pseq[i]$ equals "send".

```
yBar ≜
  LET RECURSIVE R(_, _)
      R(yseq, pseq) ≜
          IF yseq = ⟨ ⟩
             THEN yseq
             ELSE IF Head(pseq) = "send"
                     THEN ⟨Head(yseq)⟩ ∘ R(Tail(yseq), Tail(pseq))
                     ELSE R(Tail(yseq), Tail(pseq))
  IN  R(y, p)
```

We then instantiate module *SendSeq* and state as follows the theorem asserting that *SpecUP* implements formula *Spec* of that model under the refinement mapping.

```
SS ≜ INSTANCE SendSeq WITH y ← yBar
THEOREM SpecUP ⇒ SS!Spec
```

TLC can check this theorem in the usual way.

4.5 Checking the Definitions

We have shown how to define a specification $Spec^p$ for an arbitrary specification *Spec* by defining a state function *Dom* and, for every subaction A of a disjunctive representation of the next-state action of *Spec*, defining $Pred_A$, $DomInj_A$, and $PredDom_A$. These definitions must satisfy certain conditions to ensure that $\exists\, p : Spec^p$ is equivalent to *Spec*. We now state those conditions.

The first condition is (4.11). The *Prophecy* module defines this operator:

$$ExistsGoodProphecy(Pred(\_)) \triangleq \exists\, q \in [Dom \to Pi] : Pred(q)$$

For a subaction A with an empty context, we can write (4.11) as

$$Spec \Rightarrow \Box[A \Rightarrow (ExistsGoodProphecy(Pred_A)]_{vars}$$

(Remember that the *Prophecy* module will be instantiated with the appropriate expression substituted for *Dom*.) To see how this definition is used if A has a non-empty context, here is how condition is (4.11) is expressed for the subaction $UndoP(i)$ of specification *SpecU* in our *SendSeq* example:

$$
\begin{array}{l}
SpecU \;\Rightarrow\; \Box[\forall\, i \in Dom : \\
\qquad\qquad Undo(i) \Rightarrow \\
\qquad\qquad\quad ExistsGoodProphecy(\text{LAMBDA } p : PredUndo(p,\, i))]_{vars}
\end{array}
$$

The only condition we require of $DomInj_A$ is that it be a partial function from Dom to Dom'. This is expressed as $IsDomInj(DomInj_A)$ using this definition from module *Prophecy*

$$IsDomInj(DomInj) \;\triangleq\; DomInj \in PartialInjections(Dom,\, DomPrime)$$

As with the *ExistsGoodProphecy* condition, it needs to hold only for A steps in a behavior satisfying the specification *Spec*. Hence the general requirement on $DomInj_A$ for an action A with context $\langle \mathbf{k}; \mathbf{K} \rangle$ is

$$Spec \;\Rightarrow\; \Box[\forall \langle \mathbf{k}; \mathbf{K} \rangle \,:\, A \Rightarrow IsDomInj(DomInj_A)]_{vars}$$

Because *IsDomInj* does not have an operator argument, no local definition or LAMBDA expression is needed even if the context is nonempty. For example, if A is subaction $Undo(i)$ of specification *SpecU* of the *SendSeq* example, this condition is written:

$$\forall i \in Dom \,:\, Undo(i) \Rightarrow IsDomInj(DomInjUndo(i))$$

Finally, we need a condition on $PredDom_A$. Remember that $PredDom_A$ should equal the set of elements d of Dom such that $p[d]$ is making predictions about A. Actually, it suffices that $PredDom_A$ contain all such elements. (It may contain other elements as well.) This is equivalent to the requirement that any element not in $PredDom_A$ does not make a prediction about A. Making a prediction about A means affecting the value of $Pred_A$, so not making a prediction means not affecting its value. Thus, $p[d]$ does not make a prediction about A iff setting $p[d]$ to any value in Π does not change the value of $Pred_A$. You should be able to convince yourself that the value of $Pred_A$ does not depend on the value of $p[d]$ for any d not in $PredDom_A$ iff the following formula is true:

$$
\begin{array}{l}
\forall q, r \in [Dom \to Pi] \,: \\
\quad (\forall d \in PredDom_A \,:\, q[d] = r[d]) \Rightarrow (Pred_A(q) = Pred_A(r))
\end{array}
$$

In addition to this requirement, to ensure that our formulas make sense, we also make the obvious requirement that $PredDom_A$ is a subset of Dom. The following definition appears in module *Prophecy*.

$$
\begin{array}{l}
IsPredDom(PredDom,\, Pred(\_)) \;\triangleq \\
\quad \wedge\, PredDom \subseteq Dom \\
\quad \wedge\, \forall\, q,\, r \in [Dom \to Pi] : \\
\qquad\quad (\forall\, d \in PredDom : q[d] = r[d]) \Rightarrow (Pred(q) = Pred(r))
\end{array}
$$

```
Condition ≜
  ∧ ProphCondition(Choose, DomInjChoose, PredDomChoose, PredChoose)
  ∧ ProphCondition(Send,   DomInjSend,   PredDomSend,   PredSend)
  ∧ ProphCondition(Rcv,    DomInjRcv,    PredDomRcv,    PredRcv)
  ∧ ∀ i ∈ Dom :
      ProphCondition(Undo(i), DomInjUndo(i), PredDomUndo(i),
                     LAMBDA p : PredUndo(p, i))

THEOREM Spec U ⇒ □[Condition]_vars
```

Figure 14: Action requirements for specification *SendSeqUndo*.

Remembering that the condition on $PredDom_A$ needs to hold only for A steps in a behavior satisfying the specification, we can express it for an action A with an empty context as:

$$Spec \;\Rightarrow\; \Box[A \Rightarrow IsPredDom(PredDom_A, Pred_A)]_{vars}$$

Because the second argument of *IsPredDom* is an operator argument, we again need to use a local definition or a LAMBDA expression to express the condition if the subaction A has a nonempty context. For example, here is how we can express it for the $Undo(i)$ subaction in our *SendSeq* example using a local definition:

```
□[∀ i ∈ Dom :
    Undo(i) ⇒ LET Op(p) ≜ PredUndo(p, i)
              IN  IsPredDom(PredDomUndo, Op) ]_vars
```

Here is how it is written with a LAMBDA expression:

```
□[∀ i ∈ Dom :
    Undo(i) ⇒ IsPredDom( PredDomUndo,
                         LAMBDA p : PredUndo(p, i) ) ]_vars
```

The following definition from module *Prophecy* allows us to combine the three conditions.

```
ProphCondition(A, DomInj, PredDom, Pred(_)) ≜
    A ⇒ ∧ ExistsGoodProphecy(Pred)
        ∧ IsDomInj(DomInj)
        ∧ IsPredDom(PredDom, Pred)
```

Using this definition, the requirements on the definitions are expressed for specification *SendSeqUndo* in Figure 14.

We encounter the same problem here that we encountered in checking condition (4.11) for the *SendSet* example. We would like to put these requirements

in module *SendSetUndoP* (Figure 10), right before the declaration of the variable p. However, TLC can't check the theorem in a model for that specification because *SpecU* does not specify the values of variable p. We can either move all the definitions that are now in *SendSetUndoP* before the declaration of p into a separate module, put them at the end of module *SendSetUndo*, or end the module before the declaration of p by adding "=====" when checking the condition.

You should check these conditions when adding a prophecy variable. They provide a good way to debug your definitions, before you try checking that the specification with prophecy variable implements the desired specification.

4.6 Liveness

As with our other auxiliary variables, we add a prophecy variable to the safety part of a specification, but we keep the liveness part the same. As we remarked in Section 3.5, this produces unusual specifications in which the liveness property can assert a fairness condition about an action that isn't a subaction of the next-state action.

For history variables, although the form of the specifications is unusual, the specifications are not. This is not the case for prophecy variables. If *Spec* has a liveness condition, the specification $Spec^p$ obtained from it by adding a prophecy variable can be weird. As an example, suppose that in the *SendInt* specifications of Section 4.1, instead of taking *Pi* to equal *Int*, we let it equal $Int \cup \{\infty\}$ for some value $\infty \notin Int$. Everything we did would work exactly as before, and the theorem at the end of module *SendInt*1*P* in Figure 7 would still be true. If a *SendP* step set p' to ∞, predicting that the next value to be sent is ∞, then the system would simply halt (stutter forever) before the next *SendP* step because $Send \land PredSend(\infty)$ equals FALSE.

Now suppose we add a liveness condition $\mathrm{WF}_{vars}(Next)$ to our *SendInt* specifications, requiring that they never halt. We would then have

$$SpecP \triangleq InitP \land \Box[NextP]_{varsP} \land \mathrm{WF}_{vars}(Next)$$

This formula *SpecP* implies that infinitely many *Next* steps must occur, so a behavior can't halt. The weak fairness conjunct therefore requires that *SpecP* not set p' to ∞. This is weird. The liveness property $\mathrm{WF}_{vars}(Next)$ doesn't just require that something must eventually happen; it also prevents something (setting p to ∞) from ever happening. The technical term for this weirdness is that the formula *SpecP* is *not machine closed* [2], which means that its liveness property affects safety as well as liveness.

Non-machine closed specs should never be used to describe *how* a system works. You can't understand how to implement a system if the next-state action doesn't describe what it can and cannot do next. In *extremely* rare cases, a non-machine closed high-level spec is the best way to describe *what* a system should

do, rather than how it should do it. You are very unlikely to encounter such a situation in practice.

While the non-machine closed spec we get by adding a prophecy variable to a spec with liveness can be weird, this weirdness causes no problem. We don't have to implement *SpecP*. We use it only to check the correctness of *Spec*; and the presence of the liveness property makes no difference in what we do. With our modified *SendInt* specifications, we can check that *SpecP* implies $SI2!Spec$ exactly as we did before.

5 Stuttering Variables

5.1 Adding Stuttering Steps to a Simple Action

Suppose $Spec_1$ is a specification of a (24-hour) clock that displays only the hour—a specification we can write as

$$Spec_1 \triangleq (h = 0) \land \Box[h' = (h+1) \,\%\, 24]_h \tag{5.1}$$

Let $Spec_2$ be this specification of an hour-minute clock:

$$\begin{aligned} Spec_2 \triangleq\ & \land\ (h = 0) \land (m = 0) \\ & \land\ \Box[\ \land\ m' = (m+1) \,\%\, 60 \\ & \qquad \land\ h' = \text{IF } m' = 0 \text{ THEN } (h+1) \,\%\, 24 \text{ ELSE } h\ ]_{\langle h,m \rangle} \end{aligned} \tag{5.2}$$

If we ignore the variable m in $Spec_2$, then we get a clock that displays only the hour. Thus, $\exists\, m : Spec_2$ should be equivalent to $Spec_1$. (The 59 steps each hour that change only m are stuttering steps that are allowed by $Spec_1$.) It's easy to see that $Spec_2$ implies $Spec_1$, so $\exists\, m : Spec_2$ implies $Spec_1$. There is no refinement mapping with which we can prove that $Spec_1$ implies $\exists\, m : Spec_2$. To construct the necessary refinement mapping we need to add an auxiliary variable s to $Spec_1$ to obtain a specification $Spec_1^s$ that adds 59 steps that change only s to each step that increments h. Such an auxiliary variable is called a *stuttering* variable because it changes $Spec_1$ only by requiring it to add steps that leave the variable h of $Spec_1$ unchanged.

To add such a variable s to a specification *Spec* to form $Spec^s$, we let the next-state action of $Spec^s$ take "normal" steps that satisfy the next-state action of *Spec* when s equals $\top$ (usually read "top"), which is some value that is not a positive integer. The value of s in the initial state equals $\top$. When s is set to a positive integer, the specification $Spec^s$ allows only stuttering steps that decrement s, leaving the variables of *Spec* unchanged. When s counts down to zero, it is set equal to $\top$ again. We add these stuttering steps before and/or after steps of some particular subaction of the next-state action. Here, we assume that this is a "simple" subaction, meaning that its context is empty.

Suppose we want the specification $Spec^s$ to add stuttering steps to each step of a particular subaction. We replace each subaction A by the action A^s defined as follows. For each A other than that particular subaction, we define A^s by:

$$A^s \triangleq (s = \top) \land A \land (s' = s) \tag{5.3}$$

To add *initVal* stuttering steps after a step of an action A, for a positive integer *initVal* (whose value may depend on the variables of *Spec*), we define

$$\begin{array}{lll} A^s & \triangleq & \text{IF } s = \top \\ & & \quad \text{THEN } A \land (s' = initVal) \\ & & \quad \text{ELSE } \land\ vars' = vars \\ & & \qquad \land\ s' = \text{IF } s = 1 \text{ THEN } \top \text{ ELSE } s - 1 \end{array}$$

We can generalize this by replacing the set of natural numbers with an arbitrary set Σ having a well-founded partial order $\prec$ with smallest element $\bot$ (read "bottom")[6], letting *initVal* be an arbitrary element of Σ, replacing $s = 1$ with $s = \bot$, and replacing $s - 1$ by $decr(s)$ for some operator *decr* such that $decr(s) \prec s$ for all $s \in \Sigma$. The generalization is:

$$\begin{array}{lll} A^s & \triangleq & \text{IF } s = \top \\ & & \quad \text{THEN } A \land (s' = initVal) \\ & & \quad \text{ELSE } \land\ vars' = vars \\ & & \qquad \land\ s' = \text{IF } s = \bot \text{ THEN } \top \text{ ELSE } decr(s) \end{array} \tag{5.4}$$

We can add *initVal* stuttering steps before an A step, rather than after it, as follows. The stuttering steps should only be taken when they can be followed by an A step, which is the case only when an A step is enabled. Remembering that ENABLED A is the state predicate that is true iff an A step is enabled, the stuttering steps can be added with this definition of A^s:

$$\begin{array}{lll} A^s & \triangleq & \land \text{ ENABLED } A \\ & & \land \text{ IF } s = \bot \text{ THEN } A \land (s' = \top) \\ & & \qquad \text{ELSE } \land\ vars' = vars \\ & & \qquad\quad \land\ s' = \text{IF } s = \top \text{ THEN } initVal \text{ ELSE } decr(s) \end{array} \tag{5.5}$$

We could generalize (5.4) and (5.5) to allow putting stuttering steps both before and after an A step. We won't bother to do this because it is probably seldom needed, and it wouldn't be significantly simpler than adding two separate stuttering variables.

[6] This means that $\bot \prec \sigma$ for all $\sigma \in \Sigma$, and any decreasing chain $\sigma_1 \succ \sigma_2 \succ \ldots$ of elements in Σ must be finite.

5.2 Adding Stuttering Steps to Multiple Actions

To generalize what we did in the preceding section, we first consider how to add stuttering steps before or after an action A that may have a nonempty context. We assume that the set Σ, its $\bot$ element, and the operator *decr* do not depend on the values of the context variables. (This should be true in most real examples.) However, *initVal* may depend on them. We therefore must make sure that *initVal* is evaluated with the values of the context variables for which the action A is "executed". This is no problem for stuttering steps added after the A step, where A^s is defined by (5.4). (Since the stuttering steps do nothing but decrement the value of s, it makes no difference for which value of the context variables A^s is "executed" when $s \neq \top$.) However, it is a problem for stuttering steps added before an A step, where A^s is defined by (5.5).

To solve the problem for A^s defined by (5.5), we let the non-$\top$ values of s be records with a *val* component that equals the value of s described in (5.5), and a *ctxt* component that equals the tuple of values of the context when *initVal* is evaluated. In other words, *ctxt* is set by the ELSE clause in the second conjunct of (5.5). The condition that $s.ctxt$ equals the values of the context variables is added as a conjunct to the THEN clause to make sure that A is executed only in that context. The precise definition is given below.

We often need to add stuttering steps before or after more than one subaction of the next-state action. We could do that by adding separate stuttering variables, or we could introduce a stuttering array variable. However, because the stuttering steps we add to an action all occur immediately before or after that subaction, we can add them all with the same stuttering variable s. Stuttering steps can be added to each such subaction with its own set Σ and hence its own values of *initVal* and $\bot$ and of the operator *decr*. We just let the value of s indicate the action to which the stuttering steps are being added. We do this by adding to the non-$\top$ values of s an additional *id* component that identifies the action for which the stuttering steps are being added. The component is set when s is first set to a non-$\top$ value, and execution of the new subaction A^s is enabled when $s \neq \top$ only if $s.id$ equals the identifier of A.

We write the three definitions (5.3), (5.4), and (5.5) in TLA$^+$ using the three operators *NoStutter*, *PostStutter*, and *PreStutter*, respectively, shown in Figure 15. The module should be instantiated by substituting the new stuttering variable for s and the tuple of variables of the original specification for *vars*. It defines $\top$, which is written *top*, to be a value that is different from the values assigned to s by *PostStutter* and *PreStutter* actions (and that TLC knows is different from those values). The other values in (5.4) and (5.5) are provided by the following arguments to *PostStutter* and *PreStutter*.

A	The action to which stuttering steps are being added.
id	An identifier to distinguish that action A from other actions to which stuttering steps are added. We like to let it be the name of

$top \triangleq [top \mapsto \text{“top”}]$

VARIABLES s, $vars$

$NoStutter(A) \triangleq (s = top) \land A \land (s' = s)$

$PostStutter(A, actionId, context, bot, initVal, decr(\_)) \triangleq$
IF $s = top$ THEN $\land A$
$\land s' = [id \mapsto actionId, \ ctxt \mapsto context, val \mapsto initVal]$
ELSE $\land s.id = actionId$
$\land$ UNCHANGED $vars$
$\land s' =$ IF $s.val = bot$ THEN top
ELSE $[s$ EXCEPT $!.val = decr(s.val)]$

$PreStutter(A, enabled, actionId, context, bot, initVal, decr(\_)) \triangleq$
IF $s = top$
THEN $\land enabled$
$\land$ UNCHANGED $vars$
$\land s' = [id \mapsto actionId, ctxt \mapsto context, val \mapsto initVal]$
ELSE $\land s.id = actionId$
$\land$ IF $s.val = bot$ THEN $\land s.ctxt = context$
$\land A$
$\land s' = top$
ELSE $\land$ UNCHANGED $vars$
$\land s' = [s$ EXCEPT $!.val = decr(s.val)]$

Figure 15: The beginning of the *Stuttering* module.

the action (which is a string).

bot The $\bot$ (smallest) element of Σ.

initVal The value we have been calling by that name.

decr The operator we have been calling by that name. It must take a single argument.

enabled A formula that should be equivalent to ENABLED A. We can often find such a formula that TLC can evaluate much more efficiently than ENABLED A. You can use TLC to check that *enabled* is equivalent to ENABLED A by checking that *enabled* $\equiv$ ENABLED A is an invariant of the original specification.

context The tuple of context identifiers of A. (You can use i instead of a 1-tuple $\langle i \rangle$.) The *context* argument is used in the *PostStutter*

```
MayPostStutter(A, actionId, context, bot, initVal, decr(_)) ≜
  IF s = top THEN ∧ A
                  ∧ s′ = IF initVal = bot
                         THEN s
                         ELSE [id ↦ actionId, ctxt ↦ context,
                               val ↦ initVal]
             ELSE ∧ s.id = actionId
                  ∧ UNCHANGED vars
                  ∧ s′ = IF decr(s.val) = bot
                         THEN top
                         ELSE [s EXCEPT !.val = decr(s.val)]

MayPreStutter(A, enabled, actionId, context, bot, initVal, decr(_)) ≜
  IF s = top
    THEN ∧ enabled
         ∧ IF initVal = bot
             THEN A ∧ (s′ = s)
             ELSE ∧ UNCHANGED vars
                  ∧ s′ = [id ↦ actionId, ctxt ↦ context,
                          val ↦ decr(initVal)]
    ELSE ∧ s.id = actionId
         ∧ IF s.val = bot THEN ∧ s.ctxt = context
                               ∧ A
                               ∧ s′ = top
                          ELSE ∧ UNCHANGED vars
                               ∧ s′ = [s EXCEPT !.val = decr(s.val)]
```

Figure 16: The end of the *Stuttering* module.

action only to set the *ctxt* component of s. This component may be used in defining refinement mappings.

Note that *PostStutter* and *PreStutter* add at least one stuttering step, adding exactly one such step if $initVal = bot$. It is often more convenient to use operators that add one fewer stuttering step. These are the operators *MayPostStutter* and *MayPreStutter* defined in the *Stuttering* module as shown in Figure 16. Unlike the original definitions, the actions they define do not execute any stuttering step when *initVal* equals *bot*.

As a simple example, let formula *Spec* be defined as in Figure 17 to equal the hour clock specification of (5.1). Let us suppose that a module *HourMin* defines a formula *Spec* to equal the hour-minute clock specification $Spec_2$ of (5.2). To construct a refinement mapping under which the hour clock specification implements the hour-minute clock specification, we add 59 stuttering steps before

MODULE *Hour*

EXTENDS *Integers*

VARIABLE h

$Init \triangleq h = 0$

$Next \triangleq h' = (h + 1) \ \% \ 24$

$Spec \triangleq Init \wedge \Box[Next]_h$

Figure 17: The hour clock specification.

each *Next* step of the hour-clock specification. The obvious way to do that is to let Σ be the set $1 \,..\, 59$ ordered by $<$, with $\bot$ equal to 1 and *initVal* equal to 59. However, our refinement mapping becomes simpler if we use the reverse ordering $>$ of $1 \,..\, 59$, with $\bot$ equal to 59 and *initVal* equal to 1. The refinement mapping can then define $\overline{m}$ to equal 0 when $s = \top$ and $s.val$ when $s \neq \top$.

We use the operator *PreStutter* of the *Stuttering* module to define $Next^s$. We instantiate that module with *vars* equal to h and with s equal to the stuttering variable, which we also call s. For the arguments of *PreStutter*, observe that:

- *Next* is always enabled, so ENABLED *Next* equals TRUE.
- Since we are adding stuttering steps to only one action, it doesn't matter what constant we choose for the *actionId* argument.
- *Next*, which is the only subaction in the trivial disjunctive representation of *Next*, has a null context. We can therefore let the *context* argument be any constant.

We therefore add the following to the end of module *Hour*.

$vars \triangleq h$

VARIABLE s

INSTANCE *Stuttering*

$InitS \triangleq Init \wedge (s = top)$

$NextS \triangleq PreStutter(Next, \text{TRUE}, \text{“Next”}, \text{“”}, 59, 1, \text{LAMBDA } j : j + 1)$

$SpecS \triangleq InitS \wedge \Box[NextS]_{\langle vars, s \rangle}$

$HM \triangleq \text{INSTANCE } HourMin \text{ WITH } m \leftarrow \text{IF } s = top \text{ THEN } 0 \text{ ELSE } s.val$

THEOREM $SpecS \Rightarrow HM\,!\,Spec$

TLC can easily check this theorem.

5.3 Correctness of Adding a Stuttering Variable

How do we check that adding a stuttering variable using the operators of the *Stuttering* module produces a specification $Spec^s$ such that $\exists\, s : Spec^s$ is equivalent to the original specification *Spec*? The construction ensures that each behavior of $Spec^s$ is obtained by adding stuttering steps to a behavior of *Spec*, so $\exists\, s : Spec^s$ implies *Spec*. It can fail to be equivalent to *Spec* only if it either adds an infinite sequence of stuttering steps, or if we have used an incorrect *enabled* argument for *PreStutter*. It will be equivalent to *Spec* if the following conditions are satisfied for every use of the *PostStutter* and *PreStutter* operators, with arguments named as above, for some constant set Σ:

1. For every σ in Σ, the sequence of values σ, $decr(\sigma)$, $decr(decr(\sigma))$, ... is contained in Σ and eventually reaches *bot*.

2. *initVal* is in Σ.

3. *enabled* is equivalent to ENABLED A [for *PreStutter* only]

Condition 1 is a condition only on the constants Σ, *bot*, and *decr*. It can be written as $StutterConstantCondition(\Sigma, bot, decr)$ using the following definition from the *Stuttering* module:

```
StutterConstantCondition(Sigma, bot, decr(_)) ≜
  LET InverseDecr(S) ≜ {sig ∈ Sigma \ S : decr(sig) ∈ S}
      R[n ∈ Nat] ≜ IF n = 0 THEN {bot}
                            ELSE LET T ≜ R[n − 1]
                                 IN  T ∪ InverseDecr(T)
  IN  Sigma = UNION {R[n] : n ∈ Nat}
```

This condition can be checked by TLC by putting it into an ASSUME statement or else putting it in the Evaluate Constant Expression field of a model's *Model Checking Results* page. In either case, the model must replace *Nat* by a $0\,..\,n$ for a (sufficiently large) integer n, and Σ must also be replaced with a finite set if it is infinite. The *Stuttering* module defines *AltStutterConstantCondition* to be equivalent to *StutterConstantCondition* if Σ is finite, and it doesn't require redefining *Nat*.

The last two conditions are ones that need only hold for behaviors of *Spec*. They can be stated formally as follows, where A is a subaction with context $\langle \mathbf{k}; \mathbf{K} \rangle$:

$$(5.6) \quad Spec \;\Rightarrow\; \Box[\forall \langle \mathbf{k}; \mathbf{K} \rangle \,:\, A \Rightarrow (initVal \in \Sigma)]_{vars}$$

$$(5.7) \quad Spec \;\Rightarrow\; \Box(\forall \langle \mathbf{k}; \mathbf{K} \rangle \,:\, enabled \equiv \text{ENABLED } A)$$

TLC can check them in the obvious way.

5.4 Adding Infinite Stuttering

The type of stuttering variable we have been describing adds a finite number of stuttering steps before or after a step of a subaction. There is another type of stuttering variable that adds an infinite number of stuttering steps not associated with an action. You are unlikely ever to have to use one, but we include it for completeness.

Suppose we want to find a refinement mapping under which a spec $Spec_1$ that allows only halting behaviors implements a spec $Spec_2$ that allows behaviors in which externally visible variables stop changing, but internal variables keep changing forever. We obviously can't do that, because if all the variables of $Spec_1$ stop changing, then no expression defined in terms of those variables can keep changing forever. None of the methods we have described thus far for adding an auxiliary variable a to $Spec_1$ can help us, because they all have the property that if every behavior allowed by $Spec_1$ halts, then so does every behavior allowed by $Spec_1^a$.

It's hard to devise a practical example in which this problem would arise. One possibility is for $Spec_2$ to have a server perform internal actions looking for user input that may never arrive, while in $Spec_1$ the server just waits for input. Although unlikely to arise, the problem is easy to solve, so we briefly sketch a solution.

The solution is to add a stuttering variable that is required to stutter forever. Let $Spec$ equal $Init \land \Box[Next]_{vars}$ and let UC be the stuttering action UNCHANGED $vars$. Since $[Next]_{vars}$ equals $Next \lor UC$, we can write $Spec$ as $Init \land \Box[Next \lor UC]_{vars}$. We can therefore add the subaction UC to any disjunctive representation of $Next$. We add a history variable s as described in Section 3, using a disjunctive representation containing the subaction UC. This defines an action A^s for every subaction A and produces the specification $Init^s \land \Box[Next^s]_{\langle vars,s\rangle}$. (We choose UC^s so it implies $s' \neq s$.) We then define $Spec^s$ to equal

$$Init^s \land \Box[Next^s]_{\langle vars,s\rangle} \land \mathrm{WF}_{\langle vars,s\rangle}(UC^s)$$

Since ENABLED UC^s equals TRUE, the fairness requirement $\mathrm{WF}_{\langle vars,s\rangle}(UC^s)$ implies that infinitely many UC^s steps occur. These are steps that leave the variables in $vars$ unchanged, changing only s. Since we have added s as a history variable, $\exists\, s : Init^s \land \Box[Next^s]_{\langle vars,s\rangle}$ is equivalent to $Spec$. Since any TLA$^+$ spec allows stuttering steps, this implies that $\exists\, s : Spec^s$ is equivalent to $Spec$.

5.5 Liveness

Liveness poses no problem when adding a stuttering variable. As with other auxiliary variables, we obtain $Spec^s$ by adding the stuttering variable to the safety part of $Spec$ and then conjoining to it the liveness conjunct of $Spec$. (This

is true as well for the kind of stuttering variable described in Section 5.4, where $Spec^s$ contains a liveness conjunct.)

Although $Spec^s$ may have an unusual form, it isn't weird. If *Spec* is machine closed then $Spec^s$ is also machine closed. However, putting it into a standard form with fairness conditions only on subactions of $Next^s$ is not as simple as it is for history variables.

6 The Snapshot Problem

We now consider an example of using auxiliary variables to show that an algorithm satisfies its specification. Our example is based on an algorithm of Afek et al. [3]. Their algorithm implements what they call a *single-writer atomic snapshot memory*, which will call simply a *snapshot object*. Their algorithm implements a snapshot object using an unbounded amount of storage. They also present a second algorithm that uses a bounded amount of storage and implements a more general type of object, but we restrict ourselves to their first, simpler algorithm. Moreover, we consider only a simplified version of this simpler algorithm; their algorithm can be checked by adding the same auxiliary variables used for the simplified version.

6.1 Linearizability

A snapshot algorithm is used to implement an atomic read of an array of memory registers, each of which can be written by a different process. Its specification is a special case of a linearizable specification of a data object—a concept introduced by Herlihy and Wing [4].

A data object, also called a state machine, executes commands from user processes. It is described by an initial state of the object and an operator *Apply*, where $Apply(i, cmd, st)$ describes the output and new state of the object that results from process i executing command *cmd* when the object has state *st*. It is specified formally by these declared constants:

CONSTANTS *Procs*, $Commands(\_)$, $Outputs(\_)$, *InitObj*, $Apply(\_, \_, \_)$

They have the following meanings:

Procs	The set of processes.
$Commands(i)$	The set of commands that process i can issue.
$Outputs(i)$	The set of outputs the commands issued by process i can produce.
InitObj	The initial state of the object.

Apply(i, cmd, st) A record with *output* and *newState* fields describing the result of process i executing command *cmd* when the object is in state *st*.

A linearizable implementation of the data object is one in which the state of the object is internal, the only externally visible actions being the issuing of the command and the return of its output. More precisely, a process i executes a command *cmd* with a *BeginOp*(i, cmd) step, followed by a *DoOp*(i) step that modifies the state of the object, followed by an *EndOp*(i) step. The *BeginOp* and *EndOp* steps are externally visible, meaning that they modify externally visible variables (and perhaps internal variables), while the *DoOp* step modifies only internal variables—including an internal variable describing the state of the object.

To simplify the specification, we assume that the sets of commands and of outputs are disjoint. We can then use a single externally visible variable *interface*, letting *BeginOp*(i, cmd) set $interface[i]$ to the command *cmd* and letting *EndOp*(i) set it to the command's output. We also introduce an internal variable *istate* to hold the internal state of the processes—needed to remember, while a process is executing a command, whether or not it has performed the *DoOp* step and, if it has, what output was produced. We do this by letting *BeginOp*(i, cmd) set $istate[i]$ to *cmd*, and letting *DoOp*(i) set it to the command's output. Here is the definition of the next-state action and its subactions.

$$
\begin{aligned}
BeginOp(i,\ cmd) \triangleq\ & \land\ interface[i] \in Outputs(i) \\
& \land\ interface' = [interface \text{ EXCEPT } ![i] = cmd] \\
& \land\ istate' = [istate \text{ EXCEPT } ![i] = cmd] \\
& \land\ object' = object
\end{aligned}
$$

$$
\begin{aligned}
DoOp(i) \triangleq\ & \land\ interface[i] \in Commands(i) \\
& \land\ istate[i] = interface[i] \\
& \land \text{ LET } result \triangleq Apply(i,\ interface[i],\ object) \\
& \quad \text{IN} \quad \land\ object' = result.newState \\
& \qquad\ \ \land\ istate' = [istate \text{ EXCEPT } ![i] = result.output] \\
& \land\ interface' = interface
\end{aligned}
$$

$$
\begin{aligned}
EndOp(i) \triangleq\ & \land\ interface[i] \in Commands(i) \\
& \land\ istate[i] \in Outputs(i) \\
& \land\ interface' = [interface \text{ EXCEPT } ![i] = istate[i]] \\
& \land \text{ UNCHANGED } \langle object,\ istate \rangle
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ & \exists\, i \in Procs : \lor \exists\, cmd \in Commands(i) : BeginOp(i,\ cmd) \\
& \qquad\qquad\ \lor DoOp(i) \\
& \qquad\qquad\ \lor EndOp(i)
\end{aligned}
$$

Initially, $interface[i]$ and $istate[i]$ equal some output, for each i. We let that

MODULE *Linearizability*

CONSTANTS $Procs$, $Commands(\_)$, $Outputs(\_)$, $InitOutput(\_)$,
$ObjValues$, $InitObj$, $Apply(\_, \_, \_)$

ASSUME $LinearAssumps \triangleq$
$\land\ InitObj \in ObjValues$
$\land\ \forall\, i \in Procs : InitOutput(i) \in Outputs(i)$
$\land\ \forall\, i \in Procs : Outputs(i) \cap Commands(i) = \{\}$
$\land\ \forall\, i \in Procs,\ obj \in ObjValues :$
$\forall\, cmd \in Commands(i) :$
$\land\ Apply(i,\ cmd,\ obj).output \in Outputs(i)$
$\land\ Apply(i,\ cmd,\ obj).newState \in ObjValues$

VARIABLES $object$, $interface$, $istate$
$vars \triangleq \langle object,\ interface,\ istate \rangle$

$Init \triangleq \land\ object = InitObj$
$\land\ interface = [i \in Procs \mapsto InitOutput(i)]$
$\land\ istate = [i \in Procs \mapsto InitOutput(i)]$

$BeginOp(i,\ cmd) \triangleq \ldots$
$DoOp(i) \triangleq \ldots$
$EndOp(i) \triangleq \ldots$
$Next \triangleq \ldots$

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

$Fairness \triangleq \forall\, i \in Procs : \mathrm{WF}_{vars}(DoOp(i)) \land \mathrm{WF}_{vars}(EndOp(i))$
$Spec \triangleq Init \land \Box[Next]_{vars} \land Fairness$

Figure 18: Module *Linearizability*.

equal $InitOutput(i)$ for some CONSTANT operator $InitOutput$. We also add a fairness requirement to imply that any command that has begun (with a *BeginOp* step) eventually completes (with an *EndOp* step). The complete specification (with the action definitions above elided) is in module *Linearizability*, shown in Figure 18. Any particular linearizable object can be specified by instantiating the module with the appropriate constants. The module includes an assumption named *LinearAssumps*, to check that the instantiated constants satisfy the properties they should for the module to specify a linearizable object. To state all those properties, the specification introduces a constant *ObjValues* to describe the set of all possible states of the object. This set could be defined to equal the following rather complicated expression. Trying to understand it provides a good lesson in set theory.

```
LET ApplyProcTo(i, S) ≜
      {Apply(i, cmd, x).newState :  x ∈ S, cmd ∈ Commands(i)}
    ApplyTo(S) ≜ UNION {ApplyProcTo(i, S) : i ∈ Procs}
    ApplyITimes[i ∈ Nat] ≜
      IF i = 0 THEN {InitObj}
               ELSE ApplyTo(ApplyITimes[i − 1])
IN  UNION {ApplyITimes[i] : i ∈ Nat}
```

6.2 The Linearizable Snapshot Specification

By a snapshot object, we mean what Afek et al. [3] called an *atomic snapshot memory*. In a snapshot object, the processes are either readers or writers. Reader and writer should be thought of as roles; the same physical process can act as both a reader and a writer. A snapshot object is an array of registers, one per writer. A write operation writes a value to the writer's register and produces as output some fixed value that is not a possible register value. A read operation has a single command that produces the object's state (an array of register values) as output and leaves that state unchanged.

The specification declares four constants: the sets *Readers* and *Writers* of reader and writer processes; the set *RegVals* of possible register values; and a value *InitRegVal* in *RegVals* that is the initial value of a register. We call the snapshot object a *memory* and use different names for some of the parameters of the *Linearizability* module, including *MemVals* and *InitMem* for *ObjValues* and *InitObj*. We define *NotMemVal* be the single reader command and *NotRegVal* to be the single write command output. The complete specification is in module *LinearSnapshot* of Figure 19. (The ASSUME is added at the end of the module so TLC will check that the assumption *LinearAssumps* of module *Linearizability* is true under the instantiation.)

6.3 The Simplified Afek et al. Snapshot Algorithm

The snapshot algorithm of Afek et al. uses an internal variable *imem* whose value is an array with $imem[i]$ a pair consisting of the value of the i^{th} register and an integer whose value is the number of times the register has been written. It assumes that the entire pair can be read and written atomically.

A write operation writes the register value *cmd* in the obvious way, the $DoOp(i)$ action setting $imem[i]$ to $\langle cmd, imem[i][2] + 1\rangle$.

A read operation first performs the following *scan* procedure:

> It reads all the elements $imem[i]$ once, in any order. It then reads them a second time, again in any order. If it reads the same values both times for all i, it outputs the array of register values it read.

```
MODULE LinearSnapshot
CONSTANTS Readers, Writers, RegVals, InitRegVal

ASSUME ∧ Readers ∩ Writers = {}
       ∧ InitRegVal ∈ RegVals

Procs ≜ Readers ∪ Writers

MemVals ≜ [Writers → RegVals]
InitMem ≜ [i ∈ Writers ↦ InitRegVal]

NotMemVal ≜ CHOOSE v : v ∉ MemVals
NotRegVal ≜ CHOOSE v : v ∉ RegVals

Commands(i) ≜ IF i ∈ Readers THEN {NotMemVal}
                             ELSE RegVals

Outputs(i) ≜ IF i ∈ Readers THEN MemVals
                            ELSE {NotRegVal}

InitOutput(i) ≜ IF i ∈ Readers THEN InitMem ELSE NotRegVal

Apply(i, cmd, obj) ≜ IF i ∈ Readers
                     THEN [newState ↦ obj, output ↦ obj]
                     ELSE [newState ↦ [obj EXCEPT ![i] = cmd],
                           output   ↦ NotRegVal]

VARIABLES mem, interface, istate

INSTANCE Linearizability WITH ObjValues ← MemVals, InitObj ← InitMem,
                              object ← mem

ASSUME LinearAssumps
```

Figure 19: Module *LinearSnapshot*.

If the values obtained for each element $imem[i]$ by the two reads are not all the same, so the scan procedure does not produce an output, then the procedure is repeated. The scan procedure is repeated again and again until it produces an output.

The actual algorithm has an alternative method of producing an output that can be used when it has read three different values for $imem[i]$, for some writer i. By using this method, termination of the read is assured. However, for simplicity, we use an algorithm that keeps performing the scan procedure until it succeeds in producing an output. Thus, a read need never terminate, so the algorithm does not satisfy the liveness requirement of a snapshot algorithm—

namely, it does not satisfy the weak fairness requirement of the $DoOp(i)$ action for a reader i. However, it does satisfy the safety requirement. The correctness of the complete algorithm (including liveness) can be verified by essentially the same method used for our simplified version; but the complete algorithm is more complicated, so the refinement mapping is more complicated and model checking takes longer. We therefore consider only the simplified algorithm.

To specify the algorithm in TLA$^+$, we declare the same constants *Readers*, *Writers*, *RegVals*, and *InitRegVal* and make the same definitions of *MemVals*, *InitMem*, *NotMemVal*, and *NotRegVal* as in module *LinearSnapshot* above. We also define:

$$\begin{array}{lcl} IRegVals & \triangleq & RegVals \times Nat \\ IMemVals & \triangleq & [Writers \to IRegVals] \\ InitIMem & \triangleq & [i \in Writers \mapsto \langle InitRegVal, 0\rangle] \end{array}$$

We declare five variables, with the following meanings:

interface : The same as in *LinearSnapshot*.

imem : Like *mem* in *LinearSnapshot*, except $imem[i]$ is an ordered pair in $RegVals \times Nat$, the first component representing $mem[i]$ and the second the number of times $mem[i]$ has been written. The initial value of *imem* is initially *InitIMem*.

wrNum : A function with domain *Writers*, where $wrNum[i]$ is the number of $BeginWr(i)$ steps that have been taken.

$rdVal1$, $rdVal2$: They are functions such that $rdVal1[i]$ and $rdVal2[i]$ describe the values read so far by reader i in the two reads of the *scan* procedure. Both $rdVal1[i]$ and $rdVal2[i]$ are functions whose domain is the set of writers j for which the first or second read of $imem[j]$ has been performed, mapping each such j to the value read. They are set initially to the empty function (the function with empty domain), which we write $\langle\ \rangle$.

The writer actions are straightforward. Note that because $wrNum[i]$ counts the number of $BeginWr(i, cmd)$ steps and $imem[i][2]$ is set to $wrNum[i]$ by the $DoWr(i)$, the $EndWrite(i)$ action should be enabled and $DoWr(i)$ disabled when $imem[i][2]$ equals $wrNum[i]$.

$$\begin{array}{lcl} BeginWr(i,\ cmd) & \triangleq & \land\ interface[i] = NotRegVal \\ & & \land\ wrNum' = [wrNum \text{ EXCEPT } ![i] = wrNum[i] + 1] \\ & & \land\ interface' = [interface \text{ EXCEPT } ![i] = cmd] \\ & & \land\ \text{UNCHANGED } \langle imem,\ rdVal1,\ rdVal2\rangle \end{array}$$

$$\begin{array}{lcl} DoWr(i) & \triangleq & \land\ interface[i] \in RegVals \\ & & \land\ imem[i][2] \neq wrNum[i] \\ & & \land\ imem' = [imem \text{ EXCEPT } ![i] = \langle interface[i],\ wrNum[i]\rangle] \end{array}$$

$$\wedge \text{ UNCHANGED } \langle interface,\ wrNum,\ rdVal1,\ rdVal2\rangle$$

$$\begin{aligned} EndWr(i) \triangleq\ & \wedge\ interface[i] \in RegVals \\ & \wedge\ imem[i][2] = wrNum[i] \\ & \wedge\ interface' = [interface \text{ EXCEPT } ![i] = NotRegVal] \\ & \wedge \text{ UNCHANGED } \langle imem,\ wrNum,\ rdVal1,\ rdVal2\rangle \end{aligned}$$

The $BeginRd(i)$ action is straightforward.

$$\begin{aligned} BeginRd(i) \triangleq\ & \wedge\ interface[i] \in MemVals \\ & \wedge\ interface' = [interface \text{ EXCEPT } ![i] = NotMemVal] \\ & \wedge \text{ UNCHANGED } \langle imem,\ wrNum,\ rdVal1,\ rdVal2\rangle \end{aligned}$$

The definitions of the actions that perform the *scan* procedure use the following definition. We define $AddToFcn(f, x, v)$ to be the function g obtained from the function f by adding x to its domain and defining $g[x]$ to equal v. Using operators defined in the *TLC* module, it can be defined to equal $f@@(x :> v)$. However, it's easy enough to define it directly as:

$$\begin{aligned} & AddToFcn(f,\ x,\ v) \triangleq \\ & \quad [y \in (\text{DOMAIN } f) \cup \{x\} \mapsto \text{IF } y = x \text{ THEN } v \text{ ELSE } f[y]] \end{aligned}$$

Using $AddToFcn$, we define the $Rd1$ action that performs the scan's first read of *imem* and the $Rd2$ action that performs its second read.

$$\begin{aligned} Rd1(i) \triangleq\ & \wedge\ interface[i] = NotMemVal \\ & \wedge \exists\, j \in Writers \setminus \text{DOMAIN } rdVal1[i] : \\ & \qquad rdVal1' = [rdVal1 \text{ EXCEPT} \\ & \qquad\qquad ![i] = AddToFcn(rdVal1[i],\ j,\ imem[j])] \\ & \wedge \text{ UNCHANGED } \langle interface,\ imem,\ wrNum,\ rdVal2\rangle \end{aligned}$$

$$\begin{aligned} Rd2(i) \triangleq\ & \wedge\ interface[i] = NotMemVal \\ & \wedge \text{ DOMAIN } rdVal1[i] = Writers \\ & \wedge \exists\, j \in Writers \setminus \text{DOMAIN } rdVal2[i] : \\ & \qquad rdVal2' = [rdVal2 \text{ EXCEPT} \\ & \qquad\qquad ![i] = AddToFcn(rdVal2[i],\ j,\ imem[j])] \\ & \wedge \text{ UNCHANGED } \langle interface,\ imem,\ wrNum,\ rdVal1\rangle \end{aligned}$$

Finally, we define $TryEndRd(i)$ to be an action that is enabled when the reader's scan operation has completed. It compares the values read by the two sets of reads and, if they are equal, it performs the *EndOp* for the read. Otherwise, it enables the next scan to begin.

$$\begin{aligned} TryEndRd(i) \triangleq\ & \wedge\ interface[i] = NotMemVal \\ & \wedge \text{ DOMAIN } rdVal1[i] = Writers \\ & \wedge \text{ DOMAIN } rdVal2[i] = Writers \end{aligned}$$

$$
\begin{array}{l}
\land \text{ IF } rdVal1[i] = rdVal2[i] \\
\quad \text{THEN } interface' = \\
\qquad\qquad [interface \text{ EXCEPT} \\
\qquad\qquad\quad ![i] = [j \in Writers \mapsto rdVal1[i][j][1]]] \\
\quad \text{ELSE } interface' = interface \\
\land rdVal1' = [rdVal1 \text{ EXCEPT } ![i] = \langle\rangle] \\
\land rdVal2' = [rdVal2 \text{ EXCEPT } ![i] = \langle\rangle] \\
\land \text{UNCHANGED } \langle imem, wrNum\rangle
\end{array}
$$

The complete specification is in module *AfekSimplified*, shown in Figure 20 with the action definitions above elided.

6.4 Another Snapshot Specification

The algorithm in module *AfekSimplified* satisfies the safety specification in *LinearSnapshot*, but we now show that it does not implement that safety specification under any refinement mapping. Let $Spec_A$ be the algorithm's specification and let $SSpec_L$ be the specification *SafeSpec* of *LinearSnapshot*. We assume there is a refinement mapping $mem \leftarrow \overline{mem}$ and $istate \leftarrow \overline{istate}$ under which $Spec_A$ implements $SSpec_L$ and obtain a contradiction. Let $\overline{F}$ be the formula obtained from a formula F of module *LinearSnapshot* by replacing *mem* with $\overline{mem}$ and *istate* with $\overline{istate}$. Consider a behavior satisfying $Spec_A$ that begins with the following three sequences of steps.

1. Reader i does a $BeginRd(i)$ step, completes its first scan (so DOMAIN $rdVal1[i]$ equals *Writers*) and begins its second scan by reading $imem[j] = \langle v_1, 0\rangle$ for some writer j and v_1 in *RegVals* (so DOMAIN $rdVal2[i]$ equals $\{j\}$ and $rdVal2[i][j]$ equals $\langle v_1, 0\rangle$).

2. Writer j then does a complete write operation, writing a new value v_2 different from v_1.

3. Reader i completes its second scan, executes its $TryEndRd(i)$ action, finding $rdVal2[i]$ equal to $rdVal1[i]$, and completing the read operation by setting $interface[i]$ to a value M with $M[j] = v_1$.

The behavior satisfies $\overline{SSpec_L}$, so this sequence of actions must start with a $\overline{BeginRd(i)}$ step, contain a $\overline{DoRd(i)}$ step, and end with an $\overline{EndRd(i)}$ step. The reader has not determined the value to be output by the read command until it has finished its second scan, so the $\overline{DoRd(i)}$ step must occur in sequence 3. The three steps of the write of v_2 by writer j occur in sequence 2, so the $\overline{DoWr(j)}$ step for that write must occur in that sequence, therefore preceding the $\overline{DoRd(i)}$ step. Hence, the *LinearSnapshot* spec implies that the $\overline{DoRd(i)}$ step must set $\overline{istate}[i][j]$ to v_2. However in the last step of 3, the reader sets the value of $interface[i][j]$ to v_1, which implies that the $\overline{DoRd(i)}$ step set $\overline{istate}[i][j]$ to v_1.

MODULE *AfekSimplified*

EXTENDS *Integers*

CONSTANTS $Readers$, $Writers$, $RegVals$, $InitRegVal$

$MemVals \triangleq [Writers \to RegVals]$
$InitMem \triangleq [i \in Writers \mapsto InitRegVal]$
$NotMemVal \triangleq \text{CHOOSE } v : v \notin MemVals$
$NotRegVal \triangleq \text{CHOOSE } v : v \notin RegVals$

$IRegVals \triangleq RegVals \times Nat$
$IMemVals \triangleq [Writers \to IRegVals]$
$InitIMem \triangleq [i \in Writers \mapsto \langle InitRegVal, 0\rangle]$

VARIABLES $imem$, $interface$, $wrNum$, $rdVal1$, $rdVal2$
$vars \triangleq \langle imem, interface, wrNum, rdVal1, rdVal2\rangle$

$$\begin{aligned} Init \triangleq\ & \land imem = InitIMem \\ & \land interface = [i \in Readers \cup Writers \mapsto \\ & \qquad \text{IF } i \in Readers \text{ THEN } InitMem \text{ ELSE } NotRegVal] \\ & \land wrNum = [i \in Writers \mapsto 0] \\ & \land rdVal1 = [i \in Readers \mapsto \langle\rangle] \\ & \land rdVal2 = [i \in Readers \mapsto \langle\rangle] \end{aligned}$$

$BeginWr(i, cmd) \triangleq \ldots$
$DoWr(i) \triangleq \ldots$
$EndWr(i) \triangleq \ldots$
$BeginRd(i) \triangleq \ldots$
$AddToFcn(f, x, v) \triangleq \ldots$
$Rd1(i) \triangleq \ldots$
$Rd2(i) \triangleq \ldots$
$TryEndRd(i) \triangleq \ldots$

$$\begin{aligned} Next \triangleq\ & \lor \exists\, i \in Readers : BeginRd(i) \lor Rd1(i) \lor Rd2(i) \lor TryEndRd(i) \\ & \lor \exists\, i \in Writers : \lor \exists\, cmd \in RegVals : BeginWr(i, cmd) \\ & \qquad\qquad \lor DoWr(i) \lor EndWr(i) \end{aligned}$$

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

Figure 20: Module *AfekSimplified*

Since $v_1 \neq v_2$, this is a contradiction, showing that the refinement mapping cannot exist.

This behavior of $Spec_A$ is allowed by $\exists\, mem, istate : SSpec_L$, since we can choose values of mem and $istate$ for which $SSpec_L$ is satisfied—namely, values for which the $\overline{DoRd(i)}$ step occurs before the $\overline{DoWr(j)}$ step. However, choosing those values requires knowing what steps occur after the $\overline{DoWr(j)}$ step. The linearizability specification $SSpec_L$ chooses the value returned by a read sooner than it has to. This tells us that to find a refinement mapping that shows $Spec_A$ implements $\exists\, mem, istate : SSpec_L$, we must add a prophecy variable to $Spec_A$.

Instead of adding a prophecy variable to $Spec_A$, we write a new snapshot specification $Spec_{NL}$ that allows the same externally visible behaviors as specification $Spec_L$ of *LinearSnapshot*; and whose safety specification $SSpec_{NL}$ allows the same visible behaviors as $SSpec_L$. However, in $Spec_{NL}$ we make a reader wait as long as possible before choosing its output value. We can then find a refinement mapping to show that $Spec_A$ implements $SSpec_{NL}$ without using a prophecy variable.

We will still need a prophecy variable to show that $SSpec_{NL}$ allows the same externally visible behavior as $SSpec_L$. The advantage of introducing $Spec_{NL}$ is that the specification of what an algorithm is supposed to do is generally much simpler than the algorithm. Prophecy variables are the most complicated kind of auxiliary variables, and it is easier to add one to a high-level specification than to a lower-level algorithm. (This same idea of modifying the high-level specification to avoid adding a prophecy variable to the algorithm can be applied to the queue example of Herlihy and Wing [4].)

Specification $Spec_{NL}$ records in its internal state all values of the memory mem that a read operation is allowed to return. The *EndRd* operation nondeterministically chooses one of those values as its output. Its internal state therefore remembers much more about what happened in the past than a reasonable implementation would. This means that defining a refinement mapping under which an algorithm implements $Spec_{NL}$ will require adding a history variable to the algorithm's spec. Adding a history variable is much easier than adding a prophecy variable.

We write specification $Spec_{NL}$ (and $SSpec_{NL}$) in module *NewLinearSnapshot*. It has the same declarations of *Readers*, *Writers*, *RegVals*, and *InitRegVal* and the same definitions of *MemVals*, *InitMem*, *NotMemVal*, and *NotRegVal* as in module *LinearSnapshot*. It has the same variables *interface* and *mem* as module *LinearSnapshot*, plus these two internal variables:

wstate A function with domain *Writers* such that the value $wstate[i]$ is the same as the value of $istate[i]$ in *LinearSnapshot*, for each writer i.

rstate A function with domain *Readers* so that, for each reader i currently executing a read operation, $rstate[i]$ is the sequence of values that *mem* has assumed thus far while the operation has been executing.

The first element of $rstate[i]$ is therefore the value mem had when the $BeginRd(i)$ step occurred. The value of $rstate[i]$ is the empty sequence $\langle\,\rangle$ when i is not executing a read operation.

The $BeginWr$ command is essentially the same as in $LinearSnapshot$.

$$\begin{aligned} BeginWr(i,\ cmd) \triangleq\ & \land\ interface[i] = NotRegVal \\ & \land\ interface' = [interface \text{ EXCEPT } ![i] = cmd] \\ & \land\ wstate' = [wstate \text{ EXCEPT } ![i] = cmd] \\ & \land \text{ UNCHANGED } \langle mem,\ rstate\rangle \end{aligned}$$

The $BeginRd(i)$ command, which sets $rstate[i]$ to a one-element sequence containing the current value of mem, is:

$$\begin{aligned} BeginRd(i) \triangleq\ & \land\ interface[i] \in MemVals \\ & \land\ interface' = [interface \text{ EXCEPT } ![i] = NotMemVal] \\ & \land\ rstate' = [rstate \text{ EXCEPT } ![i] = \langle mem\rangle] \\ & \land \text{ UNCHANGED } \langle mem,\ wstate\rangle \end{aligned}$$

The writer executes a $DoWr$ that is the same as in $LinearSnapshot$, except that it also appends the new value of mem to the end of $rstate[j]$ for every reader j currently executing a read operation.

$$\begin{aligned} DoWr(i) \triangleq\ & \land\ interface[i] \in RegVals \\ & \land\ wstate[i] = interface[i] \\ & \land\ mem' = [mem \text{ EXCEPT } ![i] = interface[i]] \\ & \land\ wstate' = [wstate \text{ EXCEPT } ![i] = NotRegVal] \\ & \land\ rstate' = [j \in Readers \mapsto \\ & \qquad\qquad \text{IF } rstate[j] = \langle\rangle \\ & \qquad\qquad\quad \text{THEN } \langle\rangle \\ & \qquad\qquad\quad \text{ELSE } Append(rstate[j],\ mem')] \\ & \land\ interface' = interface \end{aligned}$$

A reader i has no internal actions, only the externally visible $BeginRd(i)$ and $EndRd(i)$ actions. Its $EndRd(i)$ action outputs an arbitrarily chosen element of $rstate[i]$.

$$\begin{aligned} EndRd(i) \triangleq\ & \land\ interface[i] = NotMemVal \\ & \land\ \exists\, j \in 1\,..\,Len(rstate[i]) : \\ & \qquad interface' = [interface \text{ EXCEPT } ![i] = rstate[i][j]] \\ & \land\ rstate' = [rstate \text{ EXCEPT } ![i] = \langle\rangle] \\ & \land \text{ UNCHANGED } \langle mem,\ wstate\rangle \end{aligned}$$

The writer's $EndWr$ action is essentially the same as in $LinearSnapshot$.

$$\begin{aligned} EndWr(i) \triangleq\ & \land\ interface[i] \in RegVals \\ & \land\ wstate[i] = NotRegVal \\ & \land\ interface' = [interface \text{ EXCEPT } ![i] = wstate[i]] \\ & \land \text{ UNCHANGED } \langle mem,\ rstate,\ wstate\rangle \end{aligned}$$

The complete module, minus the action definitions above, is in Figure 21.

MODULE *NewLinearSnapshot*

EXTENDS *Integers*, *Sequences*

CONSTANTS *Readers*, *Writers*, *RegVals*, *InitRegVal*

ASSUME $\land Readers \cap Writers = \{\}$
$\land InitRegVal \in RegVals$

$InitMem \triangleq [i \in Writers \mapsto InitRegVal]$
$MemVals \triangleq [Writers \to RegVals]$
$NotMemVal \triangleq$ CHOOSE $v : v \notin MemVals$
$NotRegVal \triangleq$ CHOOSE $v : v \notin RegVals$

VARIABLES *mem*, *interface*, *rstate*, *wstate*
$vars \triangleq \langle mem, interface, rstate, wstate \rangle$

$Init \triangleq \land mem = InitMem$
$\land interface = [i \in Readers \cup Writers \mapsto$
IF $i \in Readers$ THEN $InitMem$ ELSE $NotRegVal]$
$\land rstate = [i \in Readers \mapsto \langle\rangle]$
$\land wstate = [i \in Writers \mapsto NotRegVal]$

$BeginRd(i) \triangleq \ldots$
$BeginWr(i, cmd) \triangleq \ldots$
$DoWr(i) \triangleq \ldots$
$EndRd(i) \triangleq \ldots$
$EndWr(i) \triangleq \ldots$

$Next \triangleq \lor \exists\, i \in Readers : BeginRd(i) \lor EndRd(i)$
$\lor \exists\, i \in Writers : \lor \exists\, cmd \in RegVals : BeginWr(i, cmd)$
$\lor DoWr(i) \lor EndWr(i)$

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

$Fairness \triangleq \land \forall\, i \in Readers : \mathrm{WF}_{vars}(EndRd(i))$
$\land \forall\, i \in Writers : \mathrm{WF}_{vars}(DoWr(i)) \land \mathrm{WF}_{vars}(EndWr(i))$

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

Figure 21: Module *NewLinearSnapshot*

6.5 *NewLinearSnapshot* Implements *LinearSnapshot*

For compactness, in the following discussion we let:

$$\mathcal{S}_L \triangleq \exists\, mem, istate : Spec_L$$
$$\mathcal{S}_{NL} \triangleq \exists\, mem, rstate, wstate : Spec_{NL}$$

Specifications $\mathcal{S}_L$ and $\mathcal{S}_{NL}$ are equivalent. However, our goal is to prove that $Spec_A$ implements $\mathcal{S}_L$, for which it suffices to show that it implements specification $\mathcal{S}_{NL}$ and that $\mathcal{S}_{NL}$ implements $\mathcal{S}_L$. So, we won't bother showing equivalence of the two specs; we just show here that $\mathcal{S}_{NL}$ implements $\mathcal{S}_L$. We show in Section 6.6 below that $Spec_A$ implements $\mathcal{S}_{NL}$.

To show that $\mathcal{S}_{NL}$ implements $\mathcal{S}_L$, we add to $Spec_{NL}$ a prophecy variable p then a stuttering variable s to obtain a specification $Spec^{ps}_{NL}$ such that $\exists\, s, p : Spec^{ps}_{NL}$ is equivalent to $Spec_{NL}$. We then show that $\exists\, s, p : Spec^{ps}_{NL}$ implements $\mathcal{S}_L$ by showing that $Spec^{ps}_{NL}$ implements $Spec_L$ under a suitable refinement mapping $mem \leftarrow \overline{mem}$, $istate \leftarrow \overline{istate}$.

The two auxiliary variables we add to $Spec_{NL}$ have the following functions:

p A prophecy variable that predicts for each reader i which element of the sequence of memory values $rstate[i]$ will be chosen as the output.

s A stuttering variable that adds:

- A single stuttering step after a $BeginRd(i)$ step if $p[i]$ predicts that the read will return the current value of memory. The refinement mapping will be defined so that stuttering step will be a $\overline{DoRd(i)}$.
- Stuttering steps after a $DoWr(i)$ step that will implement the $\overline{DoRd(j)}$ step of every current read operation that returns the value of mem immediately after the $DoWr(i)$ step.

Both these variables are added in a single module named *NewLinearSnapshotPS*.

6.5.1 Adding the Prophecy Variable

The prophecy variable p is a prophecy data structure variable as described in Section 4.4. Its domain Dom is the set of readers that are currently executing a read, which can be described as the set of readers i such that $rstate[i]$ is a nonempty sequence. The value of $p[i]$ is a positive integer that predicts which element of the list $rstate[i]$ will be chosen as the output. This value can be arbitrarily large, since arbitrarily many writes can occur during a read operation, so Π is the set $Nat \setminus \{0\}$. Module *NewLinearSnapshotPS* therefore begins

EXTENDS *NewLinearSnapshot*

$Pi \triangleq Nat \setminus \{0\}$
$Dom \triangleq \{r \in Readers : rstate[r] \neq \langle\rangle\}$
INSTANCE *Prophecy* WITH $DomPrime \leftarrow Dom'$

It is most convenient to define p in terms of a disjunctive representation in which $EndRd(i)$ is decomposed into $\exists\, j \in 1\,..\,Len(rstate[i]) : IEndRd(i, j)$, where $IEndRd$ can be defined by:

$$\begin{aligned} IEndRd(i,\, j) \;\triangleq\; & \wedge\, interface[i] = NotMemVal \\ & \wedge\, interface' = [interface \text{ EXCEPT } ![i] = rstate[i][j]] \\ & \wedge\, rstate' = [rstate \text{ EXCEPT } ![i] = \langle\rangle] \\ & \wedge \text{ UNCHANGED } \langle mem,\, wstate\rangle \end{aligned}$$

We could make the change in our original specification *NewLinearSnapshot*, but instead we define a new next-state action Nxt that is equivalent to $Next$:

$$\begin{aligned} Nxt \;\triangleq\; & \vee \exists\, i \in Readers : \vee\, BeginRd(i) \\ & \qquad\qquad\qquad\quad \vee \exists\, j \in 1\,..\,Len(rstate[i]) : IEndRd(i,\, j) \\ & \vee \exists\, i \in Writers : \vee \exists\, cmd \in RegVals : BeginWr(i,\, cmd) \\ & \qquad\qquad\qquad\quad \vee\, DoWr(i) \vee EndWr(i) \end{aligned}$$

It's easy to see that Nxt is equivalent to formula $Next$ of *NewLinearSnapShot*, and TLAPS can easily check this proof.

```
THEOREM Next = Nxt
BY  DEF Next, Nxt, EndRd, IEndRd
```

A prediction is made for reader i when the element i is added to Dom, which is done by a $BeginRd(i)$ step. The prediction is used by the $IEndRd(i, j)$ action, allowing it to be performed only if $p[i]$ has predicted that the j^{th} item in the sequence $rstate[i]$ will be output. The definitions of $Pred_A$, $PredDom_A$, and $DomInj_A$ for the subactions A are given along with the beginning of the module in Figure 22. The module next defines the temporal formula $Condition$, which should be implied by $Spec$. There follows the definition of the specification $SpecP$ obtained by adding the prophecy variable p to $Spec$. TLC can check that $Condition$ is implied by $Spec$, which implies that $\exists\, p : SpecP$ is equivalent to $Spec$. These definitions appear in Figure 23.

6.5.2 Adding the Stuttering Variable

The module next adds the stuttering variable s to $SpecS$. We need to add a single stuttering step after a $BeginRdP(i)$ step iff the reader will output the current value of mem, which is the case iff the step sets $p[i]$ to 1. We also need to add a stuttering step after $DoWrP(i)$ for every currently reading reader j for which $p[j]$ predicts that the value of mem that the step appends to $rstate[j]$ is the one that the read will output. These steps are added by letting the values of $s.val$ be subsets of readers, ordered by the subset relation, with the decrement operation removing an element from the set chosen with the CHOOSE operator.

The specification $SpecPS$ obtained by adding the stuttering variable s to $SpecP$ is defined in the part of module *NewLinearSnapshotPS* shown in Figure 24.

MODULE *NewLinearSnapshotPS*

EXTENDS *NewLinearSnapshot*

$Pi \triangleq Nat \setminus \{0\}$
$Dom \triangleq \{r \in Readers : rstate[r] \neq \langle\rangle\}$
INSTANCE *Prophecy* WITH $DomPrime \leftarrow Dom'$

$$\begin{aligned} IEndRd(i, j) \triangleq\ & \land interface[i] = NotMemVal \\ & \land interface' = [interface \text{ EXCEPT } ![i] = rstate[i][j]] \\ & \land rstate' = [rstate \text{ EXCEPT } ![i] = \langle\rangle] \\ & \land \text{UNCHANGED } \langle mem, wstate\rangle \end{aligned}$$

$$\begin{aligned} Nxt \triangleq\ & \lor \exists\, i \in Readers : \lor BeginRd(i) \\ & \qquad\qquad\qquad\quad \lor \exists\, j \in 1 \,..\, Len(rstate[i]) : IEndRd(i, j) \\ & \lor \exists\, i \in Writers : \lor \exists\, cmd \in RegVals : BeginWr(i, cmd) \\ & \qquad\qquad\qquad\quad \lor DoWr(i) \lor EndWr(i) \end{aligned}$$

THEOREM $Next = Nxt$
BY DEF *Next*, *Nxt*, *EndRd*, *IEndRd*

$PredBeginRd(p) \triangleq$ TRUE
$PredDomBeginRd \triangleq \{\}$
$DomInjBeginRd \triangleq IdFcn(Dom)$

$PredIEndRd(p, i, j) \triangleq j = p[i]$
$PredDomIEndRd(i) \triangleq \{i\}$
$DomInjIEndRd \triangleq IdFcn(Dom')$

$PredBeginWr(p) \triangleq$ TRUE
$PredDomBeginWr \triangleq \{\}$
$DomInjBeginWr \triangleq IdFcn(Dom)$

$PredDoWr(p) \triangleq$ TRUE
$PredDomDoWr \triangleq \{\}$
$DomInjDoWr \triangleq IdFcn(Dom)$

$PredEndWr(p) \triangleq$ TRUE
$PredDomEndWr \triangleq \{\}$
$DomInjEndWr \triangleq IdFcn(Dom)$

Figure 22: Module *NewLinearSnapshotPS*, part 1.

$$
\begin{array}{l}
Condition \triangleq \\
\Box[\land \forall\, i \in Readers : \\
\qquad \land\, ProphCondition(BeginRd(i),\ DomInjBeginRd, \\
\qquad\qquad\qquad PredDomBeginRd,\ PredBeginRd) \\
\qquad \land \forall\, j \in 1\,..\,Len(rstate[i]) : \\
\qquad\quad ProphCondition(IEndRd(i,\ j),\ DomInjIEndRd, \\
\qquad\qquad\qquad PredDomIEndRd(i), \\
\qquad\qquad\qquad \text{LAMBDA } p : PredIEndRd(p,\ i,\ j)) \\
\quad \land \forall\, i \in Writers : \\
\qquad \land \forall\, cmd \in RegVals : \\
\qquad\quad ProphCondition(BeginWr(i,\ cmd),\ DomInjBeginWr, \\
\qquad\qquad\qquad PredDomBeginWr,\ PredBeginWr) \\
\qquad \land\, ProphCondition(DoWr(i),\ DomInjDoWr,\ PredDomDoWr, \\
\qquad\qquad\qquad PredDoWr) \\
\qquad \land\, ProphCondition(EndWr(i),\ DomInjEndWr,\ PredDomEndWr, \\
\qquad\qquad\qquad PredEndWr) \\
]_{vars}
\end{array}
$$

VARIABLE p

$$varsP \triangleq \langle vars,\ p \rangle$$

$$InitP \triangleq Init \land (p = EmptyFcn)$$

$$BeginRdP(i) \triangleq ProphAction(BeginRd(i),\ p,\ p',\ DomInjBeginRd,\ PredDomBeginRd,\ PredBeginRd)$$

$$BeginWrP(i,\ cmd) \triangleq ProphAction(BeginWr(i,\ cmd),\ p,\ p',\ DomInjBeginWr,\ PredDomBeginWr,\ PredBeginWr)$$

$$DoWrP(i) \triangleq ProphAction(DoWr(i),\ p,\ p',\ DomInjDoWr,\ PredDomDoWr,\ PredDoWr)$$

$$
\begin{array}{l}
IEndRdP(i,\ j) \triangleq ProphAction(IEndRd(i,\ j),\ \ p,\ p',\ DomInjIEndRd, \\
\qquad\qquad\qquad\qquad\qquad\qquad PredDomIEndRd(i), \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{LAMBDA } q : PredIEndRd(q,\ i,\ j))
\end{array}
$$

$$EndWrP(i) \triangleq ProphAction(EndWr(i),\ p,\ p',\ DomInjEndWr,\ PredDomEndWr,\ PredEndWr)$$

$$
\begin{array}{lll}
NextP \triangleq & \lor \exists\, i \in Readers : & \lor\, BeginRdP(i) \\
 & & \lor \exists\, j \in 1\,..\,Len(rstate[i]) : IEndRdP(i,\ j) \\
 & \lor \exists\, i \in Writers : & \lor \exists\, cmd \in RegVals : BeginWrP(i,\ cmd) \\
 & & \lor\, DoWrP(i) \lor EndWrP(i)
\end{array}
$$

$$SpecP \triangleq InitP \land \Box[NextP]_{varsP} \land Fairness$$

Figure 23: Module *NewLinearSnapshotPS*, part 2.

The two theorems at the beginning are conditions (5.6) for adding the stuttering steps to $BeginRdP(i)$ and $DoWrP(i)$ steps. They can be checked by temporarily ending the module immediately after those theorems and running TLC on a model having *SpecP* as its specification. Two ASSUME statements have been added to check the constant conditions on the arguments of the *MayPostStutter* operators used to add those stuttering steps.

6.5.3 The Refinement Mapping

Let's again use the abbreviations $Spec_L$ for formula *Spec* of *LinearSnapshot* and $Spec_{PS}$ for formula *SpecPS* of module *NewLinearSnapshot*. We now define the state functions $\overline{mem}$ and $\overline{istate}$ such that $Spec_{PS}$ implements $Spec_L$ under the refinement mapping $mem \leftarrow \overline{mem}$, $istate \leftarrow \overline{istate}$. We want writer actions of $Spec_L$ to be simulated by the corresponding writer actions of $Spec_{PS}$. Hence, we let $\overline{mem}$ equal mem and we let $\overline{istate}[i]$ equal $wstate[i]$ for every writer i. The problem is defining $\overline{istate}[i]$ for readers i.

In $Spec_L$, for any process i not executing a read or write, $istate[i]$ equals $interface[i]$. Hence, we can define $\overline{istate}[i]$ to equal $interface[i]$ for any reader i not currently reading. We now consider the case when i is currently reading, which is true iff $rstate[i] \neq \langle\,\rangle$, which implies $p[i]$ is a positive integer. There are two possibilities:

$p[i] = 1$ In this case, the $DoRd(i)$ step of $Spec_L$ is simulated by the stuttering step added to $BeginRd(i)$. The $DoRd(i)$ step changes $\overline{istate}[i]$ from *NotMemVal* to the memory value to be output, so $\overline{istate}[i]$ should equal $rstate[i][1]$ when $rstate[i] \neq \langle\,\rangle$, except after the $BeginRd(i)$ step and before the stuttering step added immediately after it.

$p[i] > 1$ In this case, the $DoRd(i)$ step of $Spec_L$ is simulated by one of the stuttering steps added to the $DoWr(j)$ step for the writer that appends the $p[i]^{\text{th}}$ element to $rstate[i]$. We let it be the stuttering step that removes i from $s.val$, so $\overline{istate}[i]$ equals *NotMemVal* until $p[i] \leq Len(rstate[i])$ and it's not the case that i is an element of $s.val$ while some writer is performing a stuttering step added after its *DoWr* step.

The definition of $\overline{istate}$, under the name *istateBar*, appears near the end of the module, shown in Figure 25. The theorems at the end of the module can be checked with TLC. In fact, TLC checks that *SpecPS* satisfies property *LS*!*Spec* by checking that the safety part of *SpecPS* satisfies both (a) the safety part of *LS*!*Spec* and (b) the property that the liveness part of *SpecPS* implies the liveness part of *LS*!*Spec*. Therefore, having TLC check that *SpecPS* satisfies *LS*!*Spec* checks both theorems.

THEOREM $SpecP \Rightarrow \Box[\forall\, i \in Readers : BeginRdP(i) \Rightarrow$
$(\text{IF } p'[i] = 1 \text{ THEN } 1 \text{ ELSE } 0) \in \{0, 1\}]_{varsP}$

THEOREM $SpecP \Rightarrow \Box[\forall\, i \in Writers,\ cmd \in RegVals :$
$DoWrP(i) \Rightarrow$
$\{j \in Readers : (rstate[j] \neq \langle\rangle)$
$\wedge \quad (p[j] = Len(rstate'[j]))\}$
$\in (\text{SUBSET } Readers)]_{varsP}$

VARIABLE s
$varsPS \triangleq \langle vars,\ p,\ s\rangle$

INSTANCE $Stuttering$ WITH $vars \leftarrow varsP$

$InitPS \triangleq InitP \wedge (s = top)$

$BeginRdPS(i) \triangleq MayPostStutter(BeginRdP(i),$ "BeginRd", $i,\ 0,$
$\text{IF } p'[i] = 1 \text{ THEN } 1 \text{ ELSE } 0,$
$\text{LAMBDA } j : j - 1)$
ASSUME $StutterConstantCondition(\{0, 1\},\ 0,\ \text{LAMBDA } j : j - 1)$

$BeginWrPS(i,\ cmd) \triangleq NoStutter(BeginWrP(i,\ cmd))$

$DoWrPS(i) \triangleq MayPostStutter(DoWrP(i),$ "DoWr", $i,\ \{\},$
$\{j \in Readers :$
$(rstate[j] \neq \langle\rangle) \wedge (p[j] = Len(rstate'[j]))\},$
$\text{LAMBDA } S : S \setminus \{\text{CHOOSE } x \in S : \text{TRUE}\})$
ASSUME $StutterConstantCondition(\text{SUBSET } Readers,\ \{\},$
$\text{LAMBDA } S : S \setminus \{\text{CHOOSE } x \in S : \text{TRUE}\})$

$IEndRdPS(i,\ j) \triangleq NoStutter(IEndRdP(i,\ j))$

$EndWrPS(i) \triangleq NoStutter(EndWrP(i))$

$NextPS \triangleq \lor \exists\, i \in Readers : \lor BeginRdPS(i)$
$\lor \exists\, j \in 1\,..\,Len(rstate[i]) : IEndRdPS(i,\ j)$
$\lor \exists\, i \in Writers : \lor \exists\, cmd \in RegVals : BeginWrPS(i,\ cmd)$
$\lor DoWrPS(i) \lor EndWrPS(i)$

$SafeSpecPS \triangleq InitPS \wedge \Box[NextPS]_{varsPS}$
$SpecPS \triangleq SafeSpecPS \wedge Fairness$

Figure 24: Module *NewLinearSnapshotPS*, part 3.

```
istateBar ≜ [i ∈ Readers ∪ Writers ↦
              IF i ∈ Writers
                THEN wstate[i]
                ELSE IF rstate[i] = ⟨⟩
                       THEN interface[i]
                       ELSE IF p[i] = 1
                              THEN IF ∧ s ≠ top
                                      ∧ s.id  = "BeginRd"
                                      ∧ s.ctxt = i
                                   THEN NotMemVal
                                   ELSE rstate[i][1]
                              ELSE IF ∨ p[i] > Len(rstate[i])
                                      ∨ ∧ s ≠ top
                                        ∧ s.id = "DoWr"
                                        ∧ i ∈ s.val
                                   THEN NotMemVal
                                   ELSE rstate[i][p[i]]]

LS ≜ INSTANCE LinearSnapshot WITH istate ← istateBar

THEOREM SafeSpecPS ⇒ LS!SafeSpec

THEOREM SpecPS ⇒ LS!Spec
```

Figure 25: Module *NewLinearSnapshotPS*, part 4.

6.6 *AfekSimplified* Implements *NewLinearSnapshot*

We now finish checking the correctness of the algorithm $Spec_A$ of *AfekSimplified* by showing that it implements $\exists\, mem, rstate, wstate : SSpec_{NL}$, where $SSpec_{NL}$ is the safety specification of *NewLinearSnapshot*. As we suggested in Section 6.4, finding a refinement mapping to show this requires adding a history variable to $Spec_A$ that captures the information remembered by $SSpec_{NL}$ in the variable *rstate*. This is straightforward. We just add a history variable h such that $h[i]$ is changed by $BeginRd(i)$, $DoWr(i)$, and an ending $TryEndRdH(i)$ action (one executed with $rdVal1[i] = rdVal2[i]$) the same way $rstate[i]$ is changed by the corresponding $BeginRd(i)$, $DoWr(i)$, and $EndRd(i)$ action of $SSpec_{NL}$.

The specification *SpecH*, obtained by adding the history variable h to specification *Spec* of module *AfekSimplified*, is defined in module *AfekSimplified* as shown in Figure 26. The definition should be easy to understand by comparing the definition of the initial predicate *InitH* and of the actions in the module to the corresponding definitions in module *NewLinearSnapshot*. Note that the action definitions use *memBar* where the corresponding action definitions in *NewLinearSnapshot* use *mem*. The module defines *memBar* to equal the mem-

MODULE *AfekSimplifiedH*

EXTENDS *AfekSimplified*, *Sequences*

VARIABLE h

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

$InitH \triangleq Init \land (h = [i \in Readers \mapsto \langle\rangle])$

$memBar \triangleq [i \in Writers \mapsto imem[i][1]]$

$BeginWrH(i,\ cmd) \triangleq BeginWr(i,\ cmd) \land (h' = h)$

$$\begin{aligned} DoWrH(i) \triangleq\ & \land DoWr(i) \\ & \land h' = [j \in Readers \mapsto \\ & \qquad \text{IF } h[j] = \langle\rangle \\ & \qquad\ \text{THEN } \langle\rangle \\ & \qquad\ \text{ELSE } Append(h[j],\ memBar')] \end{aligned}$$

$EndWrH(i) \triangleq EndWr(i) \land (h' = h)$

$$\begin{aligned} BeginRdH(i) \triangleq\ & \land BeginRd(i) \\ & \land h' = [h \text{ EXCEPT } ![i] = \langle memBar\rangle] \end{aligned}$$

$Rd1H(i) \triangleq Rd1(i) \land (h' = h)$

$Rd2H(i) \triangleq Rd2(i) \land (h' = h)$

$$\begin{aligned} TryEndRdH(i) \triangleq\ & \land TryEndRd(i) \\ & \land h' = \text{IF } rdVal1[i] = rdVal2[i] \\ & \qquad\ \text{THEN } [h \text{ EXCEPT } ![i] = \langle\rangle] \\ & \qquad\ \text{ELSE } h \end{aligned}$$

$$\begin{aligned} NextH \triangleq\ & \\ & \lor \exists\, i \in Readers : BeginRdH(i) \lor Rd1H(i) \lor Rd2H(i) \lor TryEndRdH(i) \\ & \lor \exists\, i \in Writers : \lor \exists\, cmd \in RegVals : BeginWrH(i,\ cmd) \\ & \qquad\qquad\ \lor DoWrH(i) \lor EndWrH(i) \end{aligned}$$

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

Figure 26: Beginning of module *AfekSimplifiedH*.

```
wstateBar ≜ [i ∈ Writers ↦
               IF (interface[i] = NotRegVal) ∨ (wrNum[i] = imem[i][2])
               THEN NotRegVal
               ELSE interface[i]]

NLS ≜ INSTANCE NewLinearSnapshot
        WITH mem ← memBar, rstate ← h, wstate ← wstateBar

THEOREM SpecH ⇒ NLS!SafeSpec
```

Figure 27: End of module *AfekSimplifiedH*.

ory value obtained from *imem* in the obvious way, by letting $memBar[i]$ equal the first element of $imem[i]$. The expression *memBar* is, of course, the value substituted for *mem* by the refinement mapping.

The rest of the refinement mapping is defined at the end of the module, shown in Figure 27. It substitutes *h* for *rstate*. The expression *wstateBar* is substituted for *wstate*. To understand it, remember that in $SSpec_{NL}$, the value of $wstate[i]$ for a writer *i* is *NotRegVal* except between a $BeginWr(i, cmd)$ action and a $DoWr(i)$ action, when it equals $interface[i]$ (which equals *cmd*).

The theorem was checked by TLC in about 10 hours on a circa 2012 laptop, using a model with: two readers, two writers, and two register values; symmetry of readers and writers (register values aren't symmetric because *IntRegVal* equals one of them); a state constraint limiting each writer to at most three writes; and two worker threads.

References

[1] Martín Abadi. The prophecy of undo. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering*, volume 9033 of *Lecture Notes in Computer Science*, pages 347–361, Berlin Heidelberg, 2015. Springer.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[3] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[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] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[6] Leslie Lamport. Auxiliary variables in TLA+. Web page at URL `http://research.microsoft.com/en-us/um/people/lamport/tla/auxiliary/aux`

[7] Leslie Lamport. TLA—temporal logic of actions. A web page, a link to which can be found at URL `http://lamport.org`. The page can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.

[8] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003. A link to an electronic copy can be found at `http://lamport.org`.

[9] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–284, May 1976.