Prophecy Made Simple

Leslie Lamport and Stephan Merz

22 June 2020

Abstract

Prophecy variables were introduced in the paper *The Existence of Refinement Mappings* by Abadi and Lamport. They were difficult to use in practice. We describe a new kind of prophecy variable that we find much easier to use. We also reformulate ideas from that paper in a more mathematical way.

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 States, Behaviors, and Specifications 3
2.2 State Machines 4
2.3 Internal Variables 5

3 Implementation and Refinement Mappings **6**
3.1 Specification $\mathcal{A}$ 6
3.2 Specification $\mathcal{B}$ 6
3.3 Implementation and a Refinement Mapping 7
3.4 Finding the Refinement Mapping 10
3.5 Generalization 11

4 Auxiliary Variables **12**
4.1 History Variables 14
4.2 Simple Prophecy Variables 15
4.3 Predicting the Impossible 16
4.4 A Sequence of Prophecies 18
4.5 A Set of Prophecies 20
4.6 Further Generalizations of Prophecy Variables 21
4.7 Stuttering Variables 22

5 Verifying Linearizability **24**

6 Prophecy Constants **27**

7 The Existence of Refinement Mappings **29**

References **30**

1 Introduction

Refinement mappings are used to verify that one specification implements another. They generalize to systems the concept of abstraction function, introduced by Hoare to define what it means for one input/output relation to implement another [9]. Refinement mappings are a central concept in extending Floyd-Hoare state-based reasoning to concurrent systems. They are crucial to making verification of those systems tractable, whether verification is by rigorous proof or model checking.

The Existence of Refinement Mappings by Abadi and Lamport [2] has become a standard reference for verifying implementation with refinement mappings in state-based formalisms. That paper, henceforth called ER, was mostly a synthesis of work that had been done in the preceding decade or so. It was well known that being able to construct a refinement mapping often requires adding to a specification a history variable that remembers information from previous states. The major new concept ER introduced was prophecy variables that predict future states, which may also be required to define a refinement mapping. ER showed that refinement mappings can always be found by adding history and prophecy variables for specifications satisfying certain conditions.

The prophecy variables defined by ER were elegant, looking like history variables with time running backwards. In practice, they turned out to be difficult to use. Defining the prophecy variable needed to verify implementation was challenging even in simple examples. We were never able to do it for realistic examples. Here, we describe a new kind of prophecy variable that we find easier to understand and to use. It makes simple examples simple and realistic examples not too hard.

We were motivated to take a fresh look at prophecy variables by a relatively recent paper of Abadi [1]. It describes techniques to make ER's prophecy variables easier to use, but we found those techniques hard to understand and prophecy variables still too hard to use. Our experience writing specifications with TLA, which was developed after ER was written, gave us a powerful new way to think about prophecy.

TLA is a linear-time temporal logic. A formula in such a logic is a predicate on sequences of states. In other temporal logics, formulas are built from predicates on states. TLA formulas are built from actions, which are predicates on pairs of states. This makes it easy to write as TLA formulas the state-machine specifications on which ER is based. Earlier temporal logics could also express actions, but not as conveniently as TLA. They therefore did not lead one to think in terms of actions.

Thinking in terms of actions led us quickly to the simple idea of letting the value of a prophecy variable predict which one of a set of actions will be the next one satisfied by a pair of successive states. For example, if each of the actions describes the sending of a different message, the value of the prophecy variable predicts which message is the next one to be sent. It was easy to generalize this idea to a prophecy variable that makes multiple predictions—even infinitely many.

In addition to explaining our new prophecy variables, we recast the concepts from ER in terms of temporal logic formulas. TLA is an obvious logic to use since it was devised for representing state machines, but the concepts should be applicable to any state-based formalism. We assume no prior knowledge of TLA or of ER. For readers who are familiar with ER, we point out the correspondence between our definitions and those of ER.

Sections 2, 3, and 4.1 explain how specifications are written, what it means for one specification to implement another, refinement mappings, and history variables. They correspond to Sections 2, 3, and 5.1 of ER. The rest of Section 4 explains prophecy variables and stuttering variables, which provide part of the functionality of ER's prophecy variables. Section 5 shows how a prophecy variable can be used to verify that a concurrent algorithm implements the specification of a linearizable object [7]. Our method should be useful for verifying linearizable specifications of other systems.

Section 6 shows how a very simple case of our prophecy variables are present in TLA and other temporal logics. Section 7 sketches a proof that the refinement mapping required to verify an implementation can always, in theory, be obtained by adding history and stuttering variables and those preexisting simple prophecy variables. However, our more general prophecy variables are usually more convenient in practice.

Our exposition is as informal as we can make it while trying to be rigorous. TLA$^+$ is a complete specification language based on TLA [11]. Most of what we describe here has been explained elsewhere in excruciating detail for TLA$^+$ users [12]. It is easy to write our examples in TLA$^+$, and their correctness has been checked with the TLA$^+$ tools. Since the examples are written somewhat informally here, we cannot be sure that they have no errors.

2 Preliminaries

2.1 States, Behaviors, and Specifications

Following Turing and ER, we model the execution of a discrete system as a sequence of states, which we call a *behavior*. For mathematical simplicity, we define a state to be an assignment of values to all possible variables. Think of a behavior as representing a history of the entire universe. We specify a system as a predicate on behaviors, which is satisfied by those behaviors that represent a history in which the system executes the way it should. Traditional verification methods consider only behaviors that represent possible executions of a system. We consider *all* behaviors, where a behavior is *any* sequence of states, and a state is *any* assignment of *any* values to variables.

Only a finite number of variables are relevant to a system; the system's specification allows behaviors in which other variables can have any values. For example, if we represent its display with the variable hr, a 12-hour clock that displays the hour is satisfied by behaviors of the form

(1) $[hr : 12],\ [hr : 1],\ [hr : 2],\ \ldots$

where $[hr : i]$ can be any state that assigns the value i to hr. We call each pair of successive states in a behavior a *step* of the behavior. A state of ER corresponds to an assignment of values to only the variables of the specification.

Common sense dictates that a specification of an hour clock should not say that the clock has no alarm, or no radio, or no display showing minutes. However, between any two steps that change the value of hr, a behavior representing a universe in which our hour clock also displays minutes must contain 59 steps in which the minute display changes and the value of hr remains the same. Therefore, in addition to allowing behaviors of the form (1), a specification of an hour clock must allow steps in which the value of hr does not change.

We define a *stuttering* step of a specification to be one in which both states assign the same values to the specification's variables. Two behaviors are said to be *stuttering-equivalent* for a specification iff (if and only if) they both have the same sequence of non-stuttering steps. We often don't mention the specification when it is clear from context. We write only specifications that are *stuttering-insensitive*, meaning that if two behaviors are stuttering-equivalent, then one satisfies the specification iff the other does. All behaviors are infinite. An execution in which a system stops is

represented by a behavior ending in an infinite sequence of stuttering steps of its specification. (The rest of the universe needn't also stop.)

An event e in an event-based formalism corresponds to a step that satisfies some predicate E on pairs of states. If the events are generated by transitions in an underlying state machine, then transitions that produce no event correspond to stuttering steps. In a purely event-based formalism, special "nothing happened" events correspond to stuttering steps.

Writing stuttering-insensitive specifications allows a simple definition of implementation (also called refinement). We say that a specification $\mathcal{S}_1$ *implements* a specification $\mathcal{S}_2$ iff every behavior satisfying $\mathcal{S}_1$ also satisfies $\mathcal{S}_2$. When predicates on behaviors are formulas in a temporal logic, $\mathcal{S}_1$ implements $\mathcal{S}_2$ means that the formula $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$ is valid (satisfied by all behaviors).

2.2 State Machines

Following Turing, ER, and common programming languages, we write our specifications in terms of state machines. A state machine is specified with two formulas: a predicate *Init* on states that describes the possible initial states and a predicate *Next* on pairs of states that describes how the state can change. We call a predicate A on pairs of states an *action*, and we call a step satisfying A an *A step*. Let $\mathbf{x}$ be the list $x_1, \ldots, x_n$ of all variables of the specification, and let $\mathrm{UC}\,\langle \mathbf{x} \rangle$ be the action satisfied only by stuttering steps—that is, steps leaving the variables $\mathbf{x}$ unchanged. The state machine specified by *Init* and *Next* is satisfied by a behavior $s_1, s_2, \ldots$ iff

SM1. s_1 satisfies *Init*, and

SM2. For all i, the step s_i, s_{i+1} satisfies $\mathit{Next} \vee \mathrm{UC}\,\langle \mathbf{x} \rangle$.

The disjunct $\mathrm{UC}\,\langle \mathbf{x} \rangle$ in SM2 ensures that the specification is stuttering-insensitive. The predicate on behaviors described by SM1 and SM2 is written in TLA as this formula:

(2) $\quad \mathit{Init} \wedge \Box[\mathit{Next}]_{\langle \mathbf{x} \rangle}$

In TLA, an action is written as an ordinary mathematical formula that may contain primed and unprimed variables. Unprimed variables refer to the values of the variables in the first state of a pair of states, and primed variables refer to their values in the second state. (An action with no primed variables is a predicate on states.) Thus, $\mathrm{UC}\,\langle \mathbf{x} \rangle$ equals $(x'_1 = x_1) \wedge \ldots \wedge (x'_n = x_n)$. Our hour-clock specification can be written in TLA as

$$(hr = 12) \wedge \Box[hr' = \textbf{if } hr = 12 \textbf{ then } 1 \textbf{ else } hr + 1)]_{\langle hr \rangle}$$

This specification allows behaviors in which, at some point, the values of the variables $\mathbf{x}$ never again change—that is, behaviors in which the clock halts. Allowing halting is a feature, not a problem. Formula (2) expresses a safety property. If we want the system also to satisfy a liveness property[1] L, we specify it as

(3) $\quad Init \wedge \Box[Next]_{\langle \mathbf{x} \rangle} \wedge L$

Letting L be the TLA formula $\mathrm{WF}_{\langle hr \rangle}(Next)$ makes (3) assert that the state machine never halts in a state in which a non-stuttering step is possible. For the hour clock, this implies that the clock never stops.

Safety and liveness properties are verified differently, so it is best to keep them separate in a specification. We will be concerned only with the safety part, so we don't care how L is written. We don't even require L to be a liveness property. Following ER, we call it a *supplementary* property.

2.3 Internal Variables

Specifying a system with a state machine often requires the use of variables that do not represent the actual state of the system, but serve to describe how that state changes. We call the variables describing the system's state *external* variables, and we call the additional variables *internal* variables. In our specifications, we want to hide the internal variables, leaving only the external variables visible.

In a linear-time temporal logic, we hide a variable y in a formula $\mathcal{F}$ with the temporal existential quantifier $\exists$. The approximate definition is that $\exists\, y : \mathcal{F}$ is true of a behavior σ iff there exist assignments of values to y in the states of σ (a separate assignment for each state of σ) that make the resulting behavior satisfy $\mathcal{F}$. This definition is wrong because it doesn't ensure that that $\exists\, y : \mathcal{F}$ is stuttering-insensitive. The correct definition is that σ satisfies $\exists\, y : \mathcal{F}$ iff there is a behavior τ stuttering-equivalent for $\mathcal{F}$ to σ and assignments of values to y that makes τ satisfy $\mathcal{F}$. For a list $\mathbf{y}$ of variables $y_1, \ldots y_m$, we define $\exists\, \mathbf{y} : \mathcal{F}$ to equal $\exists\, y_1 : \ldots \exists\, y_m : \mathcal{F}$.

We generalize the form (3) of a specification $\mathcal{S}$ to $\exists\, \mathbf{y} : \mathcal{IS}$, where

(4) $\quad \mathcal{IS} \triangleq Init \wedge \Box[Next]_{\langle \mathbf{x}, \mathbf{y} \rangle} \wedge L$

and $\mathbf{x}$ and $\mathbf{y}$ are lists of variables that may appear in *Init*, *Next*, and L. The external variables $\mathbf{x}$ are assumed to be different from the internal variables $\mathbf{y}$. We call $\mathcal{IS}$ the internal specification of $\mathcal{S}$.

[1] The definitions of safety and liveness can be found elsewhere [4]; they are not needed here.

3 Implementation and Refinement Mappings

We explain refinement mappings with an example consisting of a specification $\mathcal{A}$, a specification $\mathcal{B}$ that implements $\mathcal{A}$, and a refinement mapping that can be used to verify $\mathcal{B} \Rightarrow \mathcal{A}$.

3.1 Specification $\mathcal{A}$

Specification $\mathcal{A}$ describes a system that receives as input a sequence of integers and, after receipt of each integer, outputs the average of all the integers received thus far. Receipt of an integer i is represented by the value of the variable *in* changing from the special value `rdy` to i, where we assume `rdy` is not a number. Producing an output is represented by the value of *in* changing back to `rdy` and the value of *out* being set to the output. Initially, $in = \texttt{rdy}$ and $out = 0$. Here is the beginning of a behavior that satisfies $\mathcal{A}$:

(5) $[in : \texttt{rdy},\ out : 0],\ [in : 3,\ out : 0],\ [in : \texttt{rdy},\ out : 3],$
$[in : -2,\ out : 3],\ [in : \texttt{rdy},\ out : \frac{1}{2}],\ \ldots$

$\mathcal{A}$ is defined to equal $\exists\, sum, num : \mathcal{IA}$, where *num* is the number of outputs that have been produced and *sum* is the sum of the inputs that produced the most recent output. Here is a behavior satisfying $\mathcal{IA}$ which shows that behavior (5) satisfies $\mathcal{A}$:

(6) $[in : \texttt{rdy},\ out : 0,\ num : 0,\ sum : 0],$
$[in : 3,\ out : 0\ num : 0,\ sum : 0],$
$[in : \texttt{rdy},\ out : 3,\ num : 1,\ sum : 3],$
$[in : -2,\ out : 3,\ num : 1,\ sum : 3],$
$[in : \texttt{rdy},\ out : \frac{1}{2},\ num : 2,\ sum : 1],\ \ldots$

The complete specification $\mathcal{A}$ is defined in Figure 1, where *Int* is the set of all integers. A step satisfies the action $Next_{\mathcal{A}}$ iff it is an $Input_{\mathcal{A}}$ step or an $Output_{\mathcal{A}}$ step. An $Input_{\mathcal{A}}$ step represents the receipt of an input and an $Output_{\mathcal{A}}$ step represents the production of an output.

3.2 Specification $\mathcal{B}$

Specification $\mathcal{B}$, is a different way of writing the same specification as $\mathcal{A}$. Instead of variables that record the number of inputs and their sum, the internal specification $\mathcal{IB}$ has a single internal variable *seq* that records the entire sequence of inputs received so far. Specification $\mathcal{B}$ has the same form

$$
\begin{array}{ll}
\mathcal{A} & \triangleq \ \mathbf{\exists}\, num, sum \ : \mathcal{IA} \\
\mathcal{IA} & \triangleq \ Init_{\mathcal{A}} \ \wedge \ \Box[Next_{\mathcal{A}}]_{\langle in, out, num, sum \rangle} \\
Init_{\mathcal{A}} & \triangleq \ (in = \texttt{rdy}) \ \wedge \ (out = num = sum = 0) \\
Next_{\mathcal{A}} & \triangleq \ Input_{\mathcal{A}} \ \vee \ Output_{\mathcal{A}} \\
Input_{\mathcal{A}} & \triangleq \ (in = \texttt{rdy}) \ \wedge \ (in' \in Int) \ \wedge \ \mathrm{UC}\,\langle out, num, sum \rangle \\
Output_{\mathcal{A}} & \triangleq \quad (in \neq \texttt{rdy}) \ \wedge \ (in' = \texttt{rdy}) \\
 & \quad \wedge \ (sum' = sum + in) \ \wedge \ (num' = num + 1) \\
 & \quad \wedge \ (out' = sum' \,/\, num')
\end{array}
$$

Figure 1: The definition of specification $\mathcal{A}$.

$$
\begin{array}{ll}
\mathcal{B} & \triangleq \ \mathbf{\exists}\, seq \ : \mathcal{IB} \\
\mathcal{IB} & \triangleq \ Init_{\mathcal{B}} \ \wedge \ \Box[Next_{\mathcal{B}}]_{\langle in, out, seq \rangle} \\
Init_{\mathcal{B}} & \triangleq \ (in = \texttt{rdy}) \ \wedge \ (out = 0) \ \wedge \ (seq = \langle\,\rangle) \\
Next_{\mathcal{B}} & \triangleq \ Input_{\mathcal{B}} \ \vee \ Output_{\mathcal{B}} \\
Input_{\mathcal{B}} & \triangleq \quad (in = \texttt{rdy}) \ \wedge \ (in' \in Int) \\
 & \quad \wedge \ (seq' = Append(seq, in')) \ \wedge \ (out' = out) \\
Output_{\mathcal{B}} & \triangleq \quad (in \neq \texttt{rdy}) \ \wedge \ (in' = \texttt{rdy}) \\
 & \quad \wedge \ (out' = Sum(seq)/Len(seq)) \ \wedge \ (seq' = seq)
\end{array}
$$

Figure 2: The definition of specification $\mathcal{B}$.

as $\mathcal{A}$ except its action $Input_{\mathcal{B}}$ appends the value being input to seq, and its $Output_{\mathcal{B}}$ action outputs the average of the numbers in the sequence seq.

To write $\mathcal{B}$, we introduce some notation for sequences. We enclose sequences in angle brackets $\langle$ and $\rangle$, so $\langle\,\rangle$ is the empty sequence. We define $Len(sq)$ to equal the length of sequence sq and $Append(sq, e)$ to be the sequence obtained by appending e to the end of sequence sq, so $Len(\langle 3, 1\rangle)$ equals 2 and $Append(\langle 3, 1\rangle, 42)$ equals $\langle 3, 1, 42\rangle$. We also define $Sum(sq)$ to be the sum of the elements of sq, so $Sum(\langle 3, 1, 42\rangle$ equals 46 (which equals $3 + 1 + 42$) and $Sum(\langle\,\rangle)$ equals 0. Specification $\mathcal{B}$ is defined in Figure 2.

3.3 Implementation and a Refinement Mapping

To show $\mathcal{B} \Rightarrow \mathcal{A}$, we must show $(\mathbf{\exists}\, seq : \mathcal{IB}) \Rightarrow \mathcal{A}$. The quantifier $\mathbf{\exists}$ obeys the same rules as the quantifier $\exists$ of ordinary math. By those rules, since

seq is not a variable of $\mathcal{A}$, to show $(\exists\, seq : \mathcal{IB}) \Rightarrow \mathcal{A}$ it suffices to show $\mathcal{IB} \Rightarrow \mathcal{A}$. (This is also easy to see from the definition of $\exists$.)

For any state s, let $s[\![num \leftarrow u, sum \leftarrow v]\!]$ be the state that is the same as s except that it assigns the value u to variable num and the value v to variable sum. Since $\mathcal{A}$ equals $\exists\, num, sum : \mathcal{IA}$, to show $\mathcal{IB} \Rightarrow \mathcal{A}$, it suffices to assume that a behavior $s_1, s_2, \ldots$ satisfies $\mathcal{IB}$ and find sequences of values $\overline{num}_1, \overline{num}_2, \ldots$ and $\overline{sum}_1, \overline{sum}_2, \ldots$ such that the behavior

$$s_1[\![num \leftarrow \overline{num}_1, sum \leftarrow \overline{sum}_1]\!],\ s_2[\![num \leftarrow \overline{num}_2, sum \leftarrow \overline{sum}_2]\!],\ \ldots$$

satisfies $\mathcal{IA}$. We are free to let each $\overline{num}_i$ and $\overline{sum}_i$ depend on the entire behavior $s_1, s_2, \ldots$. However, we are going to make them depend only on the state s_i. We do that by finding expressions $\overline{num}$ and $\overline{sum}$, containing only the variables in, out, and seq of $\mathcal{IB}$, and let $\overline{num}_i$ and $\overline{sum}_i$ be the values of these expressions in state s_i.

More precisely, if u and v are expressions (formulas that need not be Boolean-valued), then let $s[\![num \leftarrow u, sum \leftarrow v]\!]$ be the state that is the same as s except that it assigns to the variables num and sum the values of u and v in state s, respectively. To show $\mathcal{IB} \Rightarrow \exists\, num, sum : \mathcal{IA}$, it suffices to find expressions $\overline{num}$ and $\overline{sum}$, containing only the (unprimed) variables of $\mathcal{IB}$, such that:

RM. If a behavior $s_1, s_2, \ldots$ satisfies $\mathcal{IB}$, then the behavior

$$s_1[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!],\ s_2[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!],\ \ldots$$

satisfies $\mathcal{IA}$.

From conditions SM1 and SM2 of Section 2.2 and the definitions of $\mathcal{IA}$ and $\mathcal{IB}$, we see that RM is implied by:

RM1. For any state s, if s satisfies $Init_{\mathcal{B}}$, then $s[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!]$ satisfies $Init_{\mathcal{A}}$.

RM2. For any states s and t, if step s, t satisfies $Next_{\mathcal{B}} \vee \mathrm{UC}\,\langle in, out, seq\rangle$, then the pair of states

$$s[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!],\ t[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!]$$

satisfies $Next_{\mathcal{A}} \vee \mathrm{UC}\,\langle in, out, num, sum\rangle$.

Because $\overline{num}$ and $\overline{sum}$ contain only the variables in, out, and seq of $\mathcal{IB}$, if the step s, t satisfies $\mathrm{UC}\,\langle in, out, seq\rangle$, then the step

$$s[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!],\ t[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!]$$

satisfies $\mathrm{UC}\,\langle in, out, num, sum\rangle$. Therefore, RM2 is automatically satisfied if s, t is a $\mathrm{UC}\,\langle in, out, seq\rangle$ step. This means we can simplify RM2 to:

RM2. For any states s and t, if the step s, t satisfies $Next_{\mathcal{B}}$, then the pair of states

$$s[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!], t[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!]$$

satisfies $Next_{\mathcal{A}} \vee \mathrm{UC}\,\langle in, out, num, sum\rangle$.

Let's consider RM1. Since $Init_{\mathcal{A}}$ is the formula

$$(in = \texttt{rdy}) \wedge (out = num = sum = 0)$$

the state $s[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!]$ satisfies $Init_{\mathcal{A}}$ iff state s satisfies

$$(7) \quad (in = \texttt{rdy}) \wedge (out = \overline{num} = \overline{sum} = 0)$$

This is the formula obtained by substituting the expression $\overline{num}$ for the variable num and the expression $\overline{sum}$ for the variable sum in the formula $Init_{\mathcal{A}}$. Let's call that formula $(Init_{\mathcal{A}}$ **with** $num \leftarrow \overline{num}, sum \leftarrow \overline{sum})$. RM1 asserts that every state satisfying $Init_{\mathcal{B}}$ satisfies (7). Therefore, it is equivalent to

RM1. $Init_{\mathcal{B}} \;\Rightarrow\; (Init_{\mathcal{A}}$ **with** $num \leftarrow \overline{num}, sum \leftarrow \overline{sum})$

As a sanity check on this condition, observe that because the variables in expressions $\overline{num}$ and $\overline{sum}$ are variables of $Init_{\mathcal{B}}$, and the other variables in and out of $Init_{\mathcal{A}}$ are also variables of $Init_{\mathcal{B}}$, the formula $(Init_{\mathcal{A}}$ **with** $\ldots)$ in RM1 contains only variables in $Init_{\mathcal{B}}$. Therefore, RM1 asserts that $Init_{\mathcal{B}}$ implies a formula containing only variables of $Init_{\mathcal{B}}$.

Applying the same reasoning to RM2, and performing the substitution in the expression $\mathrm{UC}\,\langle in, out, num, sum\rangle$, we see that RM2 is equivalent to

RM2. $Next_{\mathcal{B}} \;\Rightarrow$
$(Next_{\mathcal{A}}$ **with** $num \leftarrow \overline{num}, sum \leftarrow \overline{sum}) \vee \mathrm{UC}\,\langle in, out, \overline{num}, \overline{sum}\rangle$

Substituting an expression like $\overline{num}$ for num in $Next_{\mathcal{A}}$ means replacing num' by $\overline{num}'$. The expression $\overline{num}'$ represents the value of $\overline{num}$ in the second state of a step. It is the expression obtained by priming all the variables in $\overline{num}$.

The substitutions $num \leftarrow \overline{num}, sum \leftarrow \overline{sum}$ of expressions containing variables of $\mathcal{IB}$ for the internal variables of $\mathcal{IA}$ is what we call a refinement mapping. In ER, a state of $\mathcal{IA}$ or $\mathcal{IB}$ would be an assignment of values to that specification's variables. The mapping from states of $\mathcal{IB}$ to states of $\mathcal{IA}$ that maps s to $s[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!]$ is what ER calls a refinement mapping. Thinking of refinement mappings in terms of formulas instead of states is better when writing proofs, since proofs are written with formulas.

3.4 Finding the Refinement Mapping

Let's now find the expressions $\overline{num}$ and $\overline{sum}$ for the actual formulas defined in Figures 1 and 2 that satisfy RM1 and RM2. RM2 asserts that a step satisfying $Next_{\mathcal{B}}$ simulates a step satisfying $Next_{\mathcal{A}}$ or a stuttering step, where the values of num and sum are simulated by the values of $\overline{num}$ and $\overline{sum}$. In this simulation, the variables in and out are simulated by themselves. This implies that an $Input_{\mathcal{B}}$ step must simulate an $Input_{\mathcal{A}}$ step, leaving $\overline{num}$ and $\overline{sum}$ unchanged, and an $Output_{\mathcal{B}}$ step must simulate an $Output_{\mathcal{A}}$ step. So, we should verify RM2 by verifying these two formula:

(8) $Input_{\mathcal{B}} \Rightarrow (Input_{\mathcal{A}}$ **with** $num \leftarrow \overline{num},\ sum \leftarrow \overline{sum})$

(9) $Output_{\mathcal{B}} \Rightarrow (Output_{\mathcal{A}}$ **with** $num \leftarrow \overline{num},\ sum \leftarrow \overline{sum})$

It's pretty clear that, after an output step, $\overline{num}$ should equal $Len(seq)$ and $\overline{sum}$ should equal $Sum(seq)$. Since in equals `rdy` after an $Output_{\mathcal{A}}$ step, this leads to the following definitions:

$$\overline{num} \triangleq \textbf{if } in = rdy \textbf{ then } Len(seq) \textbf{ else } Len(Front(seq))$$
$$\overline{sum} \triangleq \textbf{if } in = rdy \textbf{ then } Sum(seq) \textbf{ else } Sum(Front(seq))$$

where $Front(sq)$ is defined to equal the sequence consisting of the first $Len(sq) - 1$ elements of sequence sq, and $Front(\langle\,\rangle)$ is defined to equal $\langle\,\rangle$.

It's easy to verify RM1, which asserts

$$(in = \texttt{rdy}) \wedge (out = 0) \wedge (seq = \langle\,\rangle) \Rightarrow$$
$$(in = \texttt{rdy}) \wedge (out = \overline{num} = \overline{sum} = 0)$$

It's not hard to verify (8), since $Input_{\mathcal{B}}$ implies $Front(seq') = seq$. Many readers will also be able to convince themselves that (9) is valid. Those readers are wrong. For example, there's no way to show that (9) is true if $in' = 42$ and $seq = \langle \texttt{rdy} \rangle$, since we don't know what $Sum(\langle \texttt{rdy} \rangle)$ and $Sum(\langle \texttt{rdy}, 42 \rangle)$ equal.

We expect many readers will object that seq can't equal $\langle \texttt{rdy} \rangle$. But, why can't it? Nothing in (9) or Figure 2 asserts that seq doesn't equal $\langle \texttt{rdy} \rangle$. What is true is that the value of seq can't equal $\langle \texttt{rdy} \rangle$ in any state of any behavior satisfying $\mathcal{IB}$. To show implementation, we don't have to show that RM2 is true for all pairs of states. It need only be true for *reachable* states, which are states that can occur in a behavior satisfying $\mathcal{IB}$. In fact, every reachable state of $\mathcal{IB}$ satisfies the following formula Inv:

$$Inv \triangleq (in \in Int \cup \{\texttt{rdy}\}) \wedge (out \in Int) \wedge (seq \in Int^*) \wedge$$
$$((in \neq rdy) \Rightarrow (seq \neq \langle\,\rangle) \wedge (in = Last(seq)))$$

where Int^* is the set of finite sequences of integers and $Last(sq)$ denotes the last element of a non-empty sequence sq. A formula that is true in every reachable state of a specification is called an *invariant* of the specification. In temporal logic, the formula $\Box Inv$ is satisfied by a behavior iff every state of the behavior satisfies Inv. Therefore, the assertion that Inv is an invariant of $\mathcal{IB}$ is expressed by $\mathcal{IB} \Rightarrow \Box Inv$.

Since Inv contains only variables of $\mathcal{IB}$, its value is left unchanged by steps that leave those variables unchanged. To show that Inv is an invariant of $\mathcal{IB}$, by induction it suffices to show:

I1. $Init_{\mathcal{B}} \Rightarrow Inv$

I2. $Inv \wedge Next_{\mathcal{B}} \Rightarrow Inv'$

(Remember that Inv' is the formula obtained by priming all the variables in Inv.) Because Inv is an invariant of $\mathcal{IB}$, instead of showing RM2 we need only show:

(10) $Inv \wedge Inv' \wedge Next_{\mathcal{B}} \Rightarrow$
$(Next_{\mathcal{A}}$ **with** $num \leftarrow \overline{num}, sum \leftarrow \overline{sum}) \vee \mathrm{UC}\,\langle in, out, \overline{num}, \overline{sum}\rangle$

We leave this to the reader.

Proving invariance by proving I1 and I2 underlies all state-based methods for proving correctness, including the Floyd-Hoare [6, 10] and Owicki-Gries [13] methods. ER avoids the explicit use of invariants by restricting a specification's set of states to ones that satisfy the needed invariant.

3.5 Generalization

We now generalize what we have done in this section to arbitrary specifications $\mathcal{S}_1$ and $\mathcal{S}_2$, with external variables $\mathbf{x}$, defined by

(11) $\mathcal{IS}_1 \triangleq Init_1 \wedge \Box[Next_1]_{\langle \mathbf{x},\mathbf{y}\rangle} \wedge L_1$
$\mathcal{IS}_2 \triangleq Init_2 \wedge \Box[Next_2]_{\langle \mathbf{x},\mathbf{z}\rangle} \wedge L_2$
$\mathcal{S}_1 \triangleq \exists\, \mathbf{y} : \mathcal{IS}_1 \qquad \mathcal{S}_2 \triangleq \exists\, \mathbf{z} : \mathcal{IS}_2$

where the lists $\mathbf{y}$ and $\mathbf{z}$ of internal variables of $\mathcal{S}_1$ and $\mathcal{S}_2$ contain no variables of $\mathbf{x}$. To verify $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$, we first define a state predicate Inv, with variables in $\mathbf{x}$ and $\mathbf{y}$, and show it is an invariant of $\mathcal{IS}_1$ by showing:

I1. $Init_1 \Rightarrow Inv$

I2. $Inv \wedge Next_1 \Rightarrow Inv'$

Then, if $\mathbf{z}$ is the list $z_1, \ldots, z_m$ of variables, we find expressions $\overline{z_1}, \ldots, \overline{z_m}$ with variables $\mathbf{x}$ and $\mathbf{y}$ and show the following; where $\mathbf{z} \leftarrow \overline{\mathbf{z}}$ means $z_1 \leftarrow \overline{z_1}, \ldots, z_m \leftarrow \overline{z_m}$:

RM1. $Init_1 \;\Rightarrow\; (Init_2 \;\mathbf{with}\; \mathbf{z} \leftarrow \overline{\mathbf{z}})$

RM2. $Inv \wedge Inv' \wedge Next_1 \;\Rightarrow\; ((Next_2 \;\mathbf{with}\; \mathbf{z} \leftarrow \overline{\mathbf{z}}) \vee \mathrm{UC}\,\langle \mathbf{x}, \overline{\mathbf{z}} \rangle)$

RM3. $Init_1 \wedge \Box[Next_1]_{\langle \mathbf{x},\mathbf{y} \rangle} \wedge L_1 \;\Rightarrow\; (L_2 \;\mathbf{with}\; \mathbf{z} \leftarrow \overline{\mathbf{z}})$

When RM1–RM3 hold, we say that $\mathcal{IS}_1$ *implements* $\mathcal{IS}_2$ *under the refinement mapping* $\mathbf{z} \leftarrow \overline{\mathbf{z}}$.

4 Auxiliary Variables

Sometimes, one specification implements another, but there does not exist a refinement mapping that shows it. For example, while we showed above that $\mathcal{B}$ implies $\mathcal{A}$, the two specifications are actually equivalent. However, $\mathcal{IA}$ does not implement $\mathcal{IB}$ under any refinement mapping because there is no way to define $\overline{seq}$ in terms of the variables of $\mathcal{A}$.

To show $\mathcal{A} \Rightarrow \mathcal{B}$, we construct a specification $\mathcal{A}^a$ from $\mathcal{A}$ containing an additional variable a such that $\mathcal{A}$ is equivalent to $\exists\, a : \mathcal{A}^a$, and we show $\mathcal{A}^a \Rightarrow \mathcal{B}$. This shows $\mathcal{A} \Rightarrow \mathcal{B}$, assuming that a is not an (external) variable of $\mathcal{B}$. Constructing $\mathcal{A}^a$ such that $\exists\, a : \mathcal{A}^a$ is equivalent to $\mathcal{A}$ is called adding the auxiliary variable a to $\mathcal{A}$. We define three kinds of auxiliary variables: *history*, *prophecy*, and *stuttering* variables.

Let specification $\mathcal{S}$ have internal specification $\mathcal{IS}$ defined by (4). We define $\mathcal{S}^a$ to equal $\exists\, \mathbf{y} : \mathcal{IS}^a$ and define

(12) $\mathcal{IS}^a \;\triangleq\; Init^a \wedge \Box[Next^a]_{\langle \mathbf{x},\mathbf{y},a \rangle} \wedge L$

where $Init^a$ and $Next^a$ are obtained from $Init$ and $Next$ by adding specifications of the initial value of a and how a changes. To show that $\mathcal{S}^a$ is obtained by adding a as an auxiliary variable—that is, $\exists\, a : \mathcal{S}^a$ is equivalent to $\mathcal{S}$—we show that $\exists\, a : \mathcal{IS}^a$ is equivalent to $\mathcal{IS}$. Since $\mathcal{IS}^a$ and $\mathcal{IS}$ have the same supplementary property L, it suffices to show their equivalence with L removed. That is, we only have to show that if we hide the variable a, the state machines of $\mathcal{IS}^a$ and $\mathcal{IS}$ are equivalent. This requires verifying two conditions:

AV1. Any behavior satisfying SM1 and SM2 for $\mathcal{IS}^a$ satisfies them for $\mathcal{IS}$.

AV2. From any behavior σ satisfying SM1 and SM2 for $\mathcal{IS}$, we can obtain a behavior σ^a satisfying SM1 and SM2 for $\mathcal{IS}^a$ by adding stuttering steps and assigning new values to the variable a in the states of the resulting behavior.

For all our auxiliary variables, $Init^a$ is defined by

(13) $Init^a \triangleq Init \wedge J$

where J is an expression containing the variables $\mathbf{x}$, $\mathbf{y}$, and a. To define $Next^a$, we write $Next$ as a disjunction of elementary actions, where we consider existential quantification to be a disjunction. For example, we can consider the elementary actions of

(14) $U \vee V \vee \exists\, i \in Int : W(i)$

to be U, V, and all $W(i)$ with $i \in Int$. (We could also consider $U \vee V$ and $\exists\, i \in Int : W(i)$ to be the elementary actions of (14).) We define $Next^a$ by replacing every elementary action A of $Next$ with an action A^a. For history and prophecy variables, A^a is defined by letting

(15) $A^a \triangleq A \wedge B$

where B is an action containing the variables $\mathbf{x}$, $\mathbf{y}$, and a (which may appear primed or unprimed), and letting a be left unchanged by stuttering steps of $\mathcal{IS}$. Condition AV1 is implied by (13) and (15). Condition AV2 is implied by:

AX. For any behavior $s_1, s_2, \ldots$ satisfying SM1 and SM2 for $\mathcal{IS}$, there exists a behavior $s_1^a, s_2^a, \ldots$ such that each s_i^a is the same as s_i except for the value it assigns to a, and: (1) s_1^a satisfies $Init^a$ and (2) for each elementary action A and each step s_i, s_{i+1} that satisfies A, the step s_i^a, s_{i+1}^a satisfies A^a.

We can show that history and prophecy variables satisfy AX. Stuttering variables can be shown to satisfy AV1 and AV2 directly.

Inspired by Abadi [1], we explain prophecy variables in terms of examples in which a specification with an *undo* action that reverses the effect of some other action implements the same specification without the *undo* action. However, there is nothing about *undo* that makes our prophecy variables work especially well. We find them just as easy to use on other kinds of examples.

4.1 History Variables

We use a history variable h to show $\mathcal{A} \Rightarrow \mathcal{B}$. A history variable stores information from the current and previous states. To be able to find a refinement mapping that shows $\mathcal{IA}^h \Rightarrow \exists\, seq : \mathcal{IB}$, we let h record the sequence of values input thus far. The initial value of h should obviously be the empty sequence, so we define

$$Init^h_{\mathcal{A}} \triangleq Init_{\mathcal{A}} \wedge (h = \langle\rangle)$$

The elementary actions of $Next_{\mathcal{A}}$ are $Input_{\mathcal{A}}$ and $Output_{\mathcal{A}}$. We let $Input^h_{\mathcal{A}}$ append the new input value to h and $Output_{\mathcal{A}}$ leave h unchanged:

$$\begin{aligned} Input^h_{\mathcal{A}} &\triangleq Input_{\mathcal{A}} \wedge (h' = Append(h, in')) \\ Output^h_{\mathcal{A}} &\triangleq Output_{\mathcal{A}} \wedge (h' = h) \end{aligned}$$

Finally, we define

$$\begin{aligned} Next^h_{\mathcal{A}} &\triangleq Input^h_{\mathcal{A}} \vee Output^h_{\mathcal{A}} \\ \mathcal{IA}^h &\triangleq Init^h_{\mathcal{A}} \wedge \Box[Next^h_{\mathcal{A}}]_{\langle in, out, sum, num, h\rangle} \\ \mathcal{A}^h &\triangleq \exists\, sum, num : \mathcal{IA}^h \end{aligned}$$

Condition AX is satisfied because, for any behavior $s_1, s_2, \ldots$ satisfying SM1 and SM2 for $\mathcal{IA}$, we can inductively define the required states s^h_i as follows: The value of h in s^h_1 is determined by the condition $h = \langle\rangle$. For each i, a nonstuttering step s_i, s_{i+1} is a step of one of the two elementary actions, and we let s^h_{i+1} assign to h the value of h' determined by the $h' = \ldots$ condition of that action. For a stuttering step, $h' = h$.

To show $\mathcal{A}^h \Rightarrow \mathcal{B}$, we let $\overline{seq}$ equal h; that is, we use the refinement mapping $seq \leftarrow h$. We must find an invariant Inv of $\mathcal{IA}^h$ and show:

$$\begin{aligned} &(16)\quad Init^h_{\mathcal{A}} \Rightarrow (Init_{\mathcal{B}} \textbf{ with } seq \leftarrow h) \\ &\qquad Inv \wedge Inv' \wedge Input^h_{\mathcal{A}} \Rightarrow (Input_{\mathcal{B}} \textbf{ with } seq \leftarrow h) \\ &\qquad Inv \wedge Inv' \wedge Output^h_{\mathcal{A}} \Rightarrow (Output_{\mathcal{B}} \textbf{ with } seq \leftarrow h) \end{aligned}$$

This is a standard exercise in assertional reasoning. Formulas (16) imply RM1 and RM2, which imply $\mathcal{A}^h \Rightarrow \mathcal{B}$.

The generalization to an arbitrary internal specification (4) is simple. We define

$$Init^h \triangleq Init \wedge (h = f)$$

where f is an expression that can contain the variables $\mathbf{x}$ and $\mathbf{y}$. For an elementary action A of *Next*, we define

$$A^h \triangleq A \wedge (h' = F)$$

where F is an expression that can contain the variables $\mathbf{x}$ and $\mathbf{y}$, both unprimed and primed, and the unprimed variable h. The general verification of AX is essentially the same as for our example. (If a step satisfies more than one elementary action, the value of h' determined by A^h for any of those actions can be used.)

4.2 Simple Prophecy Variables

We now define $\widetilde{\mathcal{A}}$ to be the same spec as $\mathcal{A}$ except with the variable *in* hidden. That is, $\widetilde{\mathcal{A}}$ equals $\exists\, in : \mathcal{A}$, which equals $\exists\, in, num, sum : \mathcal{IA}$. Thus, $\widetilde{\mathcal{A}}$ is the same as $\mathcal{A}$ except we consider input actions to be internal to the system.

We define $\mathcal{C}$ to be the same as $\widetilde{\mathcal{A}}$, except that after an input is received, the input action can be "undone", setting *in* to `rdy`, without producing any output for that input. The definition of $\mathcal{C}$ is in Figure 3.

Since *out* is the only external variable, it's clear that $\mathcal{C}$ allows the same externally visible behaviors as $\widetilde{\mathcal{A}}$. An $Input_{\mathcal{A}}$ step followed by an $Undo_{\mathcal{C}}$ step produce no change to *out*, so viewed externally they're just stuttering steps. It's obvious that $\widetilde{\mathcal{A}}$ implements $\mathcal{C}$ because $\mathcal{IA}$ implies $\mathcal{IC}$. (A behavior allowed by $\mathcal{IA}$ is allowed by $\mathcal{IC}$ because $\mathcal{IC}$ does not require that any $Undo_{\mathcal{C}}$ steps occur.) However, we can't show $\mathcal{C} \Rightarrow \widetilde{\mathcal{A}}$ with a refinement mapping, even by adding history variables.

We can verify that $\mathcal{C}$ implements $\widetilde{\mathcal{A}}$ by adding a prophecy variable p to $\mathcal{C}$ and showing that $\mathcal{IC}^p$ implements $\mathcal{IA}$ under a refinement mapping. The variable p predicts whether or not an input value will be output. More precisely, its value predicts whether the next $Output_{\mathcal{A}} \vee Undo_{\mathcal{C}}$ step will be an $Output_{\mathcal{A}}$ step or an $Undo_{\mathcal{C}}$ step. The initial predicate makes the first prediction. The next prediction is made after the currently predicted $Output_{\mathcal{A}}$ or $Undo_{\mathcal{C}}$ step occurs.

$$\begin{aligned}
\mathcal{C} &\triangleq \exists\, in, num, sum : \mathcal{IC} \\
\mathcal{IC} &\triangleq Init_{\mathcal{A}} \wedge \Box[Next_{\mathcal{C}}]_{\langle out, in, num, sum\rangle} \\
Next_{\mathcal{C}} &\triangleq Next_{\mathcal{A}} \vee Undo_{\mathcal{C}} \\
Undo_{\mathcal{C}} &\triangleq (in \neq \texttt{rdy}) \wedge (in' = \texttt{rdy}) \wedge \mathrm{UC}\,\langle out, num, sum\rangle
\end{aligned}$$

Figure 3: The definition of specification $\mathcal{C}$.

$$
\begin{array}{ll}
\mathcal{C}^p & \triangleq \; \exists\, in, num, sum \,:\, \mathcal{IC}^p \\
\mathcal{IC}^p & \triangleq \; Init^p_{\mathcal{C}} \,\wedge\, \Box[Next^p_{\mathcal{C}}]_{\langle out, in, num, sum, p\rangle} \\
Init^p_{\mathcal{C}} & \triangleq \; (p \in \{\texttt{do}, \texttt{undo}\}) \,\wedge\, Init_{\mathcal{A}} \\
Next^p_{\mathcal{C}} & \triangleq \; Input^p_{\mathcal{A}} \,\vee\, Output^p_{\mathcal{A}} \,\vee\, Undo^p_{\mathcal{C}} \\
Input^p_{\mathcal{C}} & \triangleq \; (p' = p) \,\wedge\, Input_{\mathcal{A}} \\
Output^p_{\mathcal{C}} & \triangleq \; (p = \texttt{do}) \,\wedge\, (p' \in \{\texttt{do}, \texttt{undo}\}) \wedge Output_{\mathcal{A}} \\
Undo^p_{\mathcal{C}} & \triangleq \; (p = \texttt{undo}) \,\wedge\, (p' \in \{\texttt{do}, \texttt{undo}\}) \wedge Undo_{\mathcal{C}}
\end{array}
$$

Figure 4: The definition of specification $\mathcal{C}^p$.

The specification $\mathcal{C}^p$ is defined in Figure 4. The value of the prophecy variable p is always either `do` or `undo`. Initially, p can have either of those values. If p equals `do`, then the next $Output_{\mathcal{A}}$ or $Undo_{\mathcal{C}}$ step must be an $Output_{\mathcal{A}}$ step; it must be an $Undo_{\mathcal{C}}$ step if p equals `undo`. In either case, after that step is taken, p is set to either `do` or `undo`. Condition AX is satisfied because for any behavior $s_1, s_2, \ldots$ satisfying $\mathcal{IC}$, there is a corresponding behavior $s^p_1, s^p_2, \ldots$ satisfying $\mathcal{IC}^p$ in which p always makes the correct prediction.

It's not hard to see that $\mathcal{IC}^p$ implements $\mathcal{IA}$ under this refinement mapping:

$$in \leftarrow \textbf{if } p = \texttt{undo} \textbf{ then } \texttt{rdy} \textbf{ else } in, \quad num \leftarrow num, \quad sum \leftarrow sum$$

The generalization from this example is straightforward. Suppose the next-state action $Next$ is the disjunction of elementary actions that include a set of actions A_i for i in some set P. A simple prophecy variable p that predicts for which i the next A_i step occurs is obtained by:

1. Conjoining $p \in P$ to the initial predicate $Init$.
2. Replacing each A_i by $(p = i) \,\wedge\, (p' \in P) \,\wedge\, A_i$.
3. Replacing each other elementary action B by $(p' = p) \,\wedge\, B$.

Generalizations of simple prophecy variables and of the prophecy variables described in Sections 4.4 and 4.5 are discussed in Section 4.6.

4.3 Predicting the Impossible

What if we obtain $\mathcal{S}^p$ by adding a prophecy variable p in this way to a specification $\mathcal{S}$, and p makes a prediction that cannot be fulfilled? It may

seem impossible for $\exists\, p : \mathcal{S}^p$ to be equivalent to $\mathcal{S}$ if this can happen. To see why this doesn't affect the equivalence of the two specifications, let's consider an especially egregious example. Define $\mathcal{S}$ by:

$$\mathcal{S} \triangleq (x = 0) \wedge \Box[x' = x + 1]_{\langle x \rangle}$$

Since $x' = x + 1$ equals $(x' = x + 1) \vee \text{FALSE}$, we can rewrite this as:

$$\mathcal{S} \triangleq (x = 0) \wedge \Box[(x' = x + 1) \vee \text{FALSE}]_{\langle x \rangle}$$

Following the procedure above, we add a prophecy variable p that predicts if the next $(x' = x + 1) \vee \text{FALSE}$ step is an $x' = x + 1$ step or a FALSE step.

$$\begin{aligned}
\mathcal{S}^p &\triangleq \mathit{Init}^p \wedge \Box[\mathit{Next}^p]_{\langle x, p \rangle} \\
\mathit{Init}^p &\triangleq (p \in \{\texttt{go}, \texttt{stop}\}) \wedge \mathit{Init} \\
\mathit{Next}^p &\triangleq \quad ((p = \texttt{go}) \wedge (x' = x + 1) \wedge (p' \in \{\texttt{go}, \texttt{stop}\})) \\
&\quad \vee ((p = \texttt{stop}) \wedge \text{FALSE} \wedge (p' \in \{\texttt{go}, \texttt{stop}\}))
\end{aligned}$$

If p ever becomes equal to `stop`, then no further Next^p step is possible (since no step can satisfy FALSE), at which point the behavior must consist entirely of stuttering steps. In other words, the behavior describes a system that has stopped. But that's fine because $\mathcal{S}$ allows such behaviors. If we don't want $\mathcal{S}$ to allow such halting behaviors, we must conjoin to it a supplementary property such as $\text{WF}_{\langle x \rangle}(x' = x + 1)$. In that case, $\mathcal{S}^p$ becomes

$$\mathit{Init}^p \wedge \Box[\mathit{Next}^p]_{\langle \mathbf{x}, p \rangle} \wedge \text{WF}_{\langle x \rangle}(x' = x + 1) \tag{17}$$

The conjunct $\text{WF}_{\mathbf{x}}(x' = x + 1)$ implies that a behavior must keep taking steps that increment x. Formula (17) thus rules out any behavior in which p ever equals `stop`, so $\exists\, p : \mathcal{S}^p \wedge \text{WF}_{\langle x \rangle}(x' = x + 1)$ is equivalent to $\mathcal{S} \wedge \text{WF}_{\langle x \rangle}(x' = x + 1)$.

A reader who finds this hard to understand is making the mistake of thinking of a specification like (17) as a rule for generating behaviors. It's not. It's a predicate on behaviors—a formula that is either satisfied or not satisfied by a behavior.

A reader who finds the specification (17) weird is making no mistake. It is weird. In the terminology introduced by ER, it is weird because it is not machine closed. (Machine closure is explained in ER; it originally appeared under the name *feasibility* [5].) Except in rare cases, system specifications should be machine closed. However, a specification obtained by adding a prophecy variable is not meant to specify a system. It is used only to verify the system. Its weirdness is harmless.

4.4 A Sequence of Prophecies

We generalize a simple prophecy variable that makes a single prediction to one that makes a sequence of consecutive predictions. As an example, let $\mathcal{D}$ be the specification that is the same as $\widetilde{\mathcal{A}}$ except instead of alternating between input and output actions, it maintains a queue *inq* of unprocessed input values. An input action appends a value to the end of *inq*, and an output action removes the value at the head of the queue and changes *sum* and *out* as in our previous specifications. An input action can be performed anytime, but an output action can occur only when *inq* is not empty. The definition of $\mathcal{D}$ is in Figure 5, where for any nonempty sequence *sq* of values, $Head(sq)$ is the first element of *sq* and $Tail(sq)$ is the sequence obtained from *sq* by removing its first element, with $Tail(\langle\,\rangle) = \langle\,\rangle$.

As for our previous example, we implement $\mathcal{D}$ with a specification $\mathcal{E}$ which also contains an undo action that throws away the first input in *inq* instead of processing it. It is specified in Figure 6.

To define a refinement mapping under which $\mathcal{E}$ implements $\mathcal{D}$, we add a prophecy variable whose value is a sequence of predictions, each one predicting whether the corresponding value of *inq* will be processed by an output action or thrown away by an undo action. Each prediction is made when the value is added to *inq* by an input action. The prediction is forgotten when the predicted action occurs. The definition of $\mathcal{E}^p$ is in Figure 7.

For sequences *vsq* and *dsq* of the same length, let $OnlyDo(vsq, dsq)$ be the subsequence of *vsq* consisting of all the elements for which the corresponding element of *dsq* equals `do`. For example:

$$OnlyDo(\langle 3, 2, 1, 4, 7\rangle, \langle \texttt{do}, \texttt{undo}, \texttt{undo}, \texttt{do}, \texttt{undo}\rangle) = \langle 3, 4\rangle$$

$$\begin{array}{lcl}
\mathcal{D} & \triangleq & \exists\, inq, num, sum \,:\, \mathcal{ID} \\
\mathcal{ID} & \triangleq & Init_{\mathcal{D}} \,\wedge\, \Box[Next_{\mathcal{D}}]_{\langle inq, out, num, sum\rangle} \\
Init_{\mathcal{D}} & \triangleq & (inq = \langle\,\rangle) \,\wedge\, (out = num = sum = 0) \\
Next_{\mathcal{D}} & \triangleq & Input_{\mathcal{D}} \,\vee\, Output_{\mathcal{D}} \\
Input_{\mathcal{D}} & \triangleq & \exists\, n \in Int \,:\, (inq' = Append(inq, n)) \,\wedge\, \mathrm{UC}\,\langle out, num, sum\rangle \\
Output_{\mathcal{D}} & \triangleq & \quad (inq \neq \langle\,\rangle) \,\wedge\, (inq' = Tail(inq)) \\
 & & \wedge\; (sum' = sum + Head(inq)) \,\wedge\, (num' = num + 1) \\
 & & \wedge\; (out' = sum' \,/\, num')
\end{array}$$

Figure 5: The definition of specification $\mathcal{D}$.

$$
\begin{array}{lcl}
\mathcal{E} & \triangleq & \exists\, inq, num, sum \,:\, \mathcal{IE} \\
\mathcal{IE} & \triangleq & Init_{\mathcal{D}} \,\wedge\, \Box[Next_{\mathcal{E}}]_{\langle inq, out, num, sum\rangle} \\
Next_{\mathcal{E}} & \triangleq & Next_{\mathcal{D}} \,\vee\, Undo_{\mathcal{E}} \\
Undo_{\mathcal{E}} & \triangleq & (inq \neq \langle\,\rangle) \,\wedge\, (inq' = Tail(inq)) \,\wedge\, \mathrm{UC}\,\langle out, sum, num\rangle
\end{array}
$$

Figure 6: The definition of specification $\mathcal{E}$.

$$
\begin{array}{lcl}
\mathcal{E}^p & \triangleq & \exists\, inq, num, sum \,:\, \mathcal{IE}^p \\
\mathcal{IE}^p & \triangleq & Init^p_{\mathcal{E}} \,\wedge\, \Box[Next^p_{\mathcal{E}}]_{\langle inq, out, num, sum, p\rangle} \\
Init^p_{\mathcal{E}} & \triangleq & (p = \langle\,\rangle) \,\wedge\, Init_{\mathcal{D}} \\
Next^p_{\mathcal{E}} & \triangleq & Input^p_{\mathcal{E}} \,\vee\, Output^p_{\mathcal{E}} \,\vee\, Undo^p_{\mathcal{E}} \\
Input^p_{\mathcal{E}} & \triangleq & (\exists\, d \in \{\mathtt{do}, \mathtt{undo}\} \,:\, p' = Append(p, d)) \,\wedge\, Input_{\mathcal{D}} \\
Output^p_{\mathcal{E}} & \triangleq & (Head(p) = \mathtt{do}) \,\wedge\, (p' = Tail(p)) \wedge Output_{\mathcal{D}} \\
Undo^p_{\mathcal{E}} & \triangleq & (Head(p) = \mathtt{undo}) \,\wedge\, (p' = Tail(p)) \wedge Undo_{\mathcal{E}}
\end{array}
$$

Figure 7: The definition of specification $\mathcal{E}^p$.

Specification $\mathcal{IE}$ implements $\mathcal{ID}$ under this refinement mapping:

$$inq \leftarrow OnlyDo(inq, p),\ \ sum \leftarrow sum,\ \ num \leftarrow num$$

The generalization from this example is straightforward, if we take $p = \langle\,\rangle$ to mean that there is no prediction being made. Let the next-state action *Next* be the disjunction of elementary actions that include a set of actions A_i for i in a set P. Here is how we add a prophecy variable p that makes a sequence of predictions of the i for which the next A_i step occurs:

1. Conjoin $p = \langle\,\rangle$ to the initial predicate *Init*.
2. Replace each A_i by $(p = \langle\,\rangle \,\vee\, Head(p) = i) \,\wedge\, (p' = Tail(p)) \,\wedge\, A_i$.
3. Replace each other elementary action B by either $(p' = p) \,\wedge\, B$ or $(\exists\, i \in P : p' = Append(p, i)) \,\wedge\, B$.

As with simple prophecy variables, AX is satisfied with the required behavior $s^p_1, s^p_2, \ldots$ being one in which all the right predictions are made.

In our definition of $\mathcal{E}^p$, we could eliminate the $p = \langle\,\rangle$ of condition 2 from the definitions of $Output^p_{\mathcal{E}}$ and $Undo^p_{\mathcal{E}}$ because $\mathcal{IE}^p$ implies that p is always the same length as *inq*, and $Output_{\mathcal{D}}$ and $Undo_{\mathcal{E}}$ both imply $inq \neq \langle\,\rangle$.

4.5 A Set of Prophecies

Our next type of prophecy variable is one that makes a set of concurrent predictions. Our example specification $\mathcal{F}$ is similar to $\mathcal{D}$, except that instead of a queue *inq* of inputs, it has an unordered set *inset* of inputs. An output action can process any element of *inset*. Formula $\mathcal{F}$ is defined in Figure 8, where $\setminus$ is the set difference operator, so $Int \setminus inset$ is the set of all integers not in *inset*.

As before, we add an undo action that can throw away an element in *inset* so it is not processed by an output action. The resulting specification $\mathcal{G}$ is defined in Figure 9.

To show that $\mathcal{G}$ implements $\mathcal{F}$, we add a prophecy variable p whose value is always a function with domain *inset*. For any element n of *inset*, $p(n)$ predicts whether that element will be undone or produce an output. To write the resulting specification $\mathcal{G}^p$, we need some notation for describing functions:

EmptyFcn The (unique) function whose domain is the empty set.

$Extend(f, v, w)$ The function $\widehat{f}$ obtained from function f by adding v to its domain and defining $\widehat{f}(v)$ to equal w.

$Remove(f, v)$ The function obtained from function f by removing v from its domain.

The specification $\mathcal{G}^p$ is defined in Figure 10. As before, AX holds with $s_1^p, s_2^p, \ldots$ a behavior having all the right predictions. Specification $\mathcal{IG}^p$ implements $\mathcal{IF}$ under this refinement mapping:

$$inset \leftarrow \{n \in inset : p(n) = \texttt{do}\},\ sum \leftarrow sum,\ num \leftarrow num$$

$$\begin{array}{lcl}
\mathcal{F} & \triangleq & \exists\, inset, num, sum : \mathcal{IF} \\
\mathcal{IF} & \triangleq & Init_{\mathcal{F}} \wedge \Box[Next_{\mathcal{F}}]_{\langle inset, out, num, sum\rangle} \\
Init_{\mathcal{F}} & \triangleq & (inset = \{\}) \wedge (out = num = sum = 0) \\
Next_{\mathcal{F}} & \triangleq & (\exists\, n \in Int \setminus inset : Input_{\mathcal{F}}(n)) \vee (\exists\, n \in inset : Output_{\mathcal{F}}(n)) \\
Input_{\mathcal{F}}(n) & \triangleq & (inset' = inset \cup \{n\}) \wedge \mathrm{UC}\,\langle out, num, sum\rangle \\
Output_{\mathcal{F}}(n) & \triangleq & \begin{array}{l} (inset' = inset \setminus \{n\}) \\ \wedge\ (sum' = sum + n) \wedge (num' = num + 1) \\ \wedge\ (out' = sum' \,/\, num') \end{array}
\end{array}$$

Figure 8: The definition of specification $\mathcal{F}$.

$$
\begin{array}{lcl}
\mathcal{G} & \triangleq & \exists\, inset, num, sum \,:\, \mathcal{IG} \\
\mathcal{IG} & \triangleq & Init_{\mathcal{F}} \,\wedge\, \Box[Next_{\mathcal{G}}]_{\langle inset, out, num, sum\rangle} \\
Next_{\mathcal{G}} & \triangleq & Next_{\mathcal{F}} \,\vee\, (\exists\, n \in inset \,:\, Undo_{\mathcal{G}}(n)) \\
Undo_{\mathcal{G}}(n) & \triangleq & (inset' = inset \setminus \{n\}) \,\wedge\, \mathrm{UC}\,\langle out, sum, num\rangle
\end{array}
$$

Figure 9: The definition of specification $\mathcal{G}$

$$
\begin{array}{lcl}
\mathcal{G}^p & \triangleq & \exists\, inset, num, sum \,:\, \mathcal{IG}^p \\
\mathcal{IG}^p & \triangleq & Init^p_{\mathcal{G}} \,\wedge\, \Box[Next^p_{\mathcal{G}}]_{\langle inset, out, num, sum, p\rangle} \\
Init^p_{\mathcal{G}} & \triangleq & (p = EmptyFcn) \,\wedge\, Init_{\mathcal{F}} \\
Next^p_{\mathcal{G}} & \triangleq & \quad (\exists\, n \in Int \setminus inset \,:\, Input^p_{\mathcal{G}}(n)) \\
 & & \vee\ (\exists\, n \in inset \,:\, Output^p_{\mathcal{G}}(n) \,\vee\, Undo^p_{\mathcal{G}}(n)) \\
Input^p_{\mathcal{G}}(n) & \triangleq & (\exists\, d \in \{\texttt{do}, \texttt{undo}\} \,:\, p' = Extend(p, n, d)) \,\wedge\, Input_{\mathcal{F}}(n) \\
Output^p_{\mathcal{G}}(n) & \triangleq & (p(n) = \texttt{do}) \,\wedge\, (p' = Remove(p, n)) \wedge Output_{\mathcal{F}}(n) \\
Undo^p_{\mathcal{G}} & \triangleq & (p(n) = \texttt{undo}) \,\wedge\, (p' = Remove(p, n)) \wedge Undo_{\mathcal{G}}(n)
\end{array}
$$

Figure 10: The definition of specification $\mathcal{G}^p$.

which assigns to the variable *inset* of $\mathcal{IF}$ the subset of *inset* consisting of all elements n with $p(n) = \texttt{do}$.

The only nontrivial part of the generalization from this example to an arbitrary set of prophecies is that p should make no prediction for a value not in its domain. Usually, as in our example, the actions to which the prediction apply are not enabled for a value not in the domain of p. If that's not the case, then the condition conjoined to an action to enforce the prediction should equal TRUE if the prediction is being made for a value not in the domain of p.

4.6 Further Generalizations of Prophecy Variables

Prophecy variables making sequences and sets of predictions can be generalized to prophecy variables whose predictions are organized in any data structure—even an infinite one. A data structure can be represented as a function. For example, a sequence of length n is naturally represented as a function with domain the set $\{1, 2, \ldots, n\}$. The generalization is described in detail in [12]. The basic ideas are:

- A prediction predicts a value i for which the next step satisfying an action $\exists\, i \in P : A_i$ satisfies A_i. To add the prophecy variable, each A_i is modified to enforce this prediction.

- An action or an initial condition that makes a prediction must allow any value i in P to be predicted.

- Any action may remove a prediction and/or make a new prediction. An action that fulfills a prediction must remove that prediction and may replace it with a new prediction. Any other action may leave the prediction unchanged.

Whether or not a particular prophecy is made is often indicated by the data structure containing the prophecies. In the example of Section 4.5, whether a prediction is made for an integer n depends on whether or not n is in the domain of p. Sometimes it is convenient to indicate the absence of a prophecy by a special value `none` that is not an element of the set P of possible predictions. In the example of a simple prophecy variable in Section 4.2, we could let the *Output* and *Undo* actions remove the prophecy by setting p to `none`, and have the *Input* action make the prophecy by setting p to `do` or `undo`. Handling `none` values is straightforward.

4.7 Stuttering Variables

Usually, when $\mathcal{S}_1$ implements $\mathcal{S}_2$, specification $\mathcal{S}_1$ takes more steps than $\mathcal{S}_2$. Those extra steps simulate stuttering steps of $\mathcal{S}_2$ under a refinement mapping. If $\mathcal{S}_2$ takes more steps than $\mathcal{S}_1$ to perform some operation, then defining a refinement mapping requires an auxiliary variable that adds stuttering steps to $\mathcal{S}_1$. For example, our specification of an hour clock implements the specification of an hour-minute clock with the variable describing the minute display hidden. Defining a refinement mapping to show this requires an auxiliary variable that adds to the hour-clock specification 59 stuttering steps between every change to the variable hr. ER used prophecy variables for this. We find it more convenient to use another type of auxiliary variable, which we obviously call a stuttering variable.

It's easy to make up examples like the hour clock implementing the hour-minute clock where a stuttering variable is clearly required. In practice, stuttering variables are often used in more subtle ways. A realistic use appears in Section 5 below. A more surprising use is that in the three examples of prophecy variables in Sections 4.2, 4.4, and 4.5, we can use stuttering variables instead of the prophecy variables. We simply add a

stuttering step before each output-action step, and we define the refinement mapping to make that stuttering step implement the input step. For the last two of those examples, the refinement mapping maps each behavior of the specification with undo to a behavior in which there is never more than one value in the internal queue or set. We can use stuttering variables instead of prophecy variables in these examples only because they unrealistically make input steps internal while output steps are externally visible.

We add stuttering steps before and/or after elementary actions of the next-state action. An easy way to do it is to let the value of the stuttering variable s be a natural number. Normally s equals 0; it is set to a positive integer to take stuttering steps, the value of s being used to count the number of steps remaining. For example, consider the specification $\mathit{Init} \land \Box[\mathit{Next}]_{\mathbf{x}}$, where $\mathbf{x}$ is the tuple of all the specification's variables (internal and external); and let *Next* equal $A \lor B \lor C$. A stuttering variable s that adds 4 stuttering steps after each C step can be defined by:

$$\begin{array}{lll} \mathit{Init}^s & \triangleq & \mathit{Init} \land (s = 0) \qquad \mathit{Next}^s \triangleq A^s \lor B^s \lor C^s \\ A^s & \triangleq & (s = s' = 0) \land A \qquad B^s \triangleq (s = s' = 0) \land B \\ C^s & \triangleq & ((s = 0) \land (s' = 4) \land C) \ \lor \ ((s > 0) \land (s' = s - 1) \land \mathrm{UC}\,\langle \mathbf{x} \rangle) \end{array}$$

To add 4 stuttering steps *before* each C step, we have to write C in the form $E \land D$, where E is a state predicate and D is an action that is enabled in every state satisfying E—which means that for every state s satisfying E there is a state t such that the step s, t is a D step. Most elementary actions in specifications can easily be written in this form. (In TLA$^+$, we can always let E equal ENABLED C and let D equal C.) We can then define

$$\begin{array}{lll} C^s & \triangleq & ((s = 0) \ \land \ E \ \land \ (s' = 4) \ \land \ \mathrm{UC}\,\langle \mathbf{x} \rangle) \\ & & \lor \ ((s > 1) \ \land \ (s' = s - 1) \ \land \ \mathrm{UC}\,\langle \mathbf{x} \rangle) \\ & & \lor \ ((s = 1) \ \land \ (s' = 0) \ \land \ D) \end{array}$$

It is not hard to see that both of these constructions satisfy AV1 and AV2.

We don't have to use natural numbers for counting stuttering states. For example, we can add stuttering steps both before and after an action by using negative integers to count the steps after the action, counting up to 0. Often, we let s take values that help define the refinement mapping. For example, suppose we want to take stuttering steps so the refinement mapping can implement an action by each process satisfying some condition. We can let s always be a sequence of processes, where the empty sequence is the normal value of s, and counting down is done by $s' = \mathit{Tail}(s)$.

A single variable s can be used to add stuttering steps before and/or after multiple actions. For example, we can let the normal value of s be $\langle\,\rangle$, add stuttering steps to an action A by letting s assume values of the form $\langle\,\text{“A”}, i\,\rangle$ for a number i, and add stuttering steps to an action B by letting s assume values of the form $\langle\,\text{“B”}, q\,\rangle$ for q a sequence of processes.

To handle the unusual case when $\mathcal{S}_1$ implements $\mathcal{S}_2$ but it has internal behaviors that halt while the corresponding internal behaviors of $\mathcal{S}_2$ must take additional steps, we add an *infinite stuttering variable* s to $\mathcal{S}_1$ that simply keeps changing forever. We do this by conjoining $\text{WF}_{\langle s\rangle}(s' \neq s)$ to the internal specification of $\mathcal{S}_1$.

5 Verifying Linearizability

Linearizability has become a standard way of specifying an object shared by multiple processes [7]. A process's operation *Op* is described by a sequence of three steps: a *BeginOp* step that provides the operation's input, a *DoOp* step that performs the operation by reading and/or modifying the object, and an *EndOp* step that reports the operation's output. The *BeginOp* and *EndOp* steps are externally visible, meaning that they change external variables. The *DoOp* step is internal, meaning it modifies only internal variables.

We illustrate our use of auxiliary variables for verifying a linearizability specification with the atomic snapshot algorithm of Afek et al. [3]. Our discussion is informal; formal TLA$^+$ specifications are in [12]. The algorithm implements an array of memory registers accessed by a set of writer processes and a set of reader processes, with one register for each writer. A writer can perform *write* operations to its register. A reader can perform *read* operations that return a “snapshot” of the memory—that is, the values of all the registers.

We let *LinearSnap* be a linearizable specification of what a snapshot algorithm should do. It uses an internal variable *mem*, where $mem(w)$ equals the value of writer w's register. A *DoWrite* step modifies $mem(w)$ for a single writer w. A single *DoRead* step reads the value of *mem*. Another internal variable maintains a process's state while it is performing an operation, including whether the *DoOp* action has been performed and, for a reader, what value of *mem* was read by *DoRead* and will be returned by *EndRead*. An external variable describes the *BeginOp* and *EndOp* actions.

We consider a simplified version of the Afek et al. snapshot algorithm we call *SimpleAfek*. It maintains an internal variable *imem*. A writer w writes a value v on its i^{th} write by setting $imem(w)$ to the pair $\langle i, v\rangle$. A

reader does a sequence of reads of *imem*, each of those reads reading the values of $imem(w)$ for all writers w in separate actions, executed in any order. If the reader obtains the same value of *imem* on two successive reads, it returns the obvious snapshot contained in that value of *imem*. If not, it keeps reading. *SimpleAfek* does not guarantee termination. The actual algorithms add a way to have reading terminate after at most three reads, and a way to replace the unbounded write numbers by a bounded set of values. These more complicated versions can be handled in the same way as *SimpleAfek*.

SimpleAfek implements *LinearSnap*, but constructing a refinement mapping to show that it does requires predicting the future. To see why, assume a refinement mapping under which *SimpleAfek* implements *LinearSnap*. Since *BeginOp* and *EndOp* actions of *LinearSnap* modify external variables, there is no choice of which *SimpleAfek* actions implement them under a refinement mapping. Only the choice of which *SimpleAfek* action implements *DoOp* depends on the refinement mapping. Consider the following scenario, in which we conflate actions of *LinearSnap* with the actions of *SimpleAfek* that simulate them.

The scenario begins with no operation in progress. A reader performs a *BeginRead* action, completes one read of *imem*, and then begins its second read by reading $imem(w)$, obtaining the same value as in its first read. Writer w then performs its *BeginWrite* action, writes a new value in $imem(w)$, and is about to perform its *EndWrite* action. If no other writer performs a *DoWrite*, then the reader will complete its second read of *imem*, obtaining a snapshot containing the old value of $mem(w)$. This requires that its *DoRead* must occur before the *DoWrite* of w. However, suppose another writer u *does* perform *BeginWrite* after w performs the *DoWrite* and writes $imem(u)$ before the reader reads it, and no more write operations are performed. In that case, the reader will read *imem* two more times and then return a snapshot containing the value of $imem(w)$ just written by w, so the *DoRead* must occur after the *DoWrite* by w. Thus, knowing whether the *DoRead* occurs before or after the *DoWrite* of w requires knowing what writes occur in the future. Constructing the refinement mapping requires predicting the future.

Linearizability provides a simple, uniform way of specifying data objects; but it provides little insight into what state must be maintained by an implementation. Whether this is a feature or a flaw depends on what the specification is used for. We present an equivalent snapshot specification *NewLinearSnap* that can make verifying correctness of an implementation easier. We verify that *SimpleAfek* implements *LinearSnap* by verifying

that it implements *NewLinearSnap* and that *NewLinearSnap* implements *LinearSnap*.

In addition to the internal variable *mem* of *LinearSnap*, *NewLinearSnap* uses an internal variable *isnap* such that $isnap(r)$ is the sequence of snapshots (values of *mem*) that a read by r can return. The *BeginRead* action sets $isnap(r)$ to a one-element sequence containing the current value of *mem*. The writer actions are the same as in *LinearSnap*, except that a *DoWrite* action appends the new value of *mem* to $isnap(r)$ for all readers r that have executed a *BeginRead* action but not the corresponding *EndRead*. The *EndRead* action of reader r returns a nondeterministically chosen element of the sequence $isnap(r)$. There is no *DoRead* action.

To verify that *SimpleAfek* implements *NewLinearSnap*, we add to it a history variable that has the same value as variable *isnap* of *NewLinearSnap*. Translating an understanding of why the algorithm is correct into an invariant of *SimpleAfek* and a refinement mapping under which it implements *NewLinearSnap* is then a typical exercise in assertional reasoning about concurrent algorithms, requiring no prophecy variable.

Although *NewLinearSnap* is equivalent to *LinearSnap*, to verify *SimpleAfek* we need only verify that it implements *LinearSnap*. This is done by first adding to it a prophecy variable p so that $p(r)$ predicts which element of the sequence $isnap(r)$ of snapshots will be chosen by the *EndRead* action. The value of $p(r)$ is set to an arbitrary positive integer by r's *BeginRead* action and is reset to `none` by its *EndRead* action. We then add a stuttering variable that adds a single stuttering step after r's *BeginRead* action if $p(r) = 1$ and adds stuttering steps after a *DoWrite* action—one stuttering step for every read r for which the write adds the $p(r)^{\text{th}}$ element to $isnap(r)$. Each of those stuttering steps will simulate a *DoRead* step for one reader. To add the stuttering step after a *BeginRead* step, the stuttering variable simply counts down from 1. To add the stuttering steps after a *DoWrite* step, it counts down using the set of readers whose *DoRead* the steps will simulate. Requiring the stuttering steps to simulate those *DoRead* actions makes it clear how to define the refinement mapping.

This technique of verifying linearizability by verifying an equivalent specification should often be applicable. In their paper defining linearizability [7], Herlihy and Wing specify a linearizable FIFO queue and show an implementation in which defining a refinement mapping requires predicting the future. As with the snapshot example, we can write a new specification of the queue that is equivalent to the linearizable specification, and then verify that the Herlihy-Wing algorithm implements that new specification without needing

a prophecy variable. Instead of maintaining a totally ordered queue of elements, the new specification maintains a partially ordered set, where the partial order describes constraints on the order in which items may be dequeued. To define a refinement mapping showing that the new specification implements the original one, we add a prophecy variable that predicts the order in which items will be dequeued.

6 Prophecy Constants

In addition to variables and constants like 0, a temporal logic formula can contain constant parameters. The sets of readers and writers in the *Simple-Afek* specification are examples of constant parameters. While the value of a variable can be different in different states of a behavior, the value of a constant parameter is the same throughout any behavior. (Logicians call constant parameters *rigid variables*, and what we call variables they call *flexible variables*.)

In addition to quantifiers over variables, temporal logic has quantifiers $\exists$ and $\forall$ over constant parameters. A behavior σ satisfies the formula $\exists\, n : \mathcal{F}$ iff there is a value of the constant parameter n (the same value in every state of σ) for which σ satisfies $\mathcal{F}$. We let $\exists\, n \in P : \mathcal{F}$ equal $\exists\, n : (n \in P) \wedge \mathcal{F}$, where P is a constant expression (one containing only constants and constant parameters) not containing n. The following simple rule of ordinary logic holds for any temporal logic formulas $\mathcal{F}$ and $\mathcal{G}$ and constant expression P.

> $\exists$ **Elimination** To prove $(\exists\, n \in P : \mathcal{F}) \Rightarrow \mathcal{G}$, it suffices to assume $n \in P$ and prove $\mathcal{F} \Rightarrow \mathcal{G}$.

The following example from Section 5.2 of ER shows how this rule can be used to construct refinement mappings that require predicting the future, without adding a prophecy variable.

Specification $\mathcal{S}_1$ is satisfied by behaviors that begin with $x = 0$, repeatedly increment x by 1, and eventually stop (take only stuttering steps). It has no internal variables. Specification $\mathcal{S}_2$ has external variable x and internal variable y. Its internal specification is satisfied by behaviors that begin with $x = 0$ and y any element of the set *Nat* of natural numbers, take steps that increment x by 1 and decrement y by 1, and stop when $y = 0$. The TLA specifications of $\mathcal{S}_1$ and $\mathcal{S}_2$ are in Figure 11, where formula *Stops* asserts that the value of x eventually stops changing.

Clearly $\mathcal{S}_1$ and $\mathcal{S}_2$ are equivalent, since both are satisfied by behaviors in which x is incremented a finite number of times (possibly 0 times) and then

$$
\begin{array}{l}
\mathit{Init}_1 \triangleq x = 0 \qquad \mathit{Next}_1 \triangleq x' = x + 1 \\
\mathit{Stops} \triangleq \Diamond\Box[x' = x]_{\langle x \rangle} \\
\mathcal{S}_1 \triangleq \mathit{Init}_1 \wedge \Box[\mathit{Next}_1]_{\langle x \rangle} \wedge \mathit{Stops}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Init}_2 \triangleq (x = 0) \wedge (y \in \mathit{Nat}) \\
\mathit{Next}_2 \triangleq (y > 0) \wedge (x' = x + 1) \wedge (y' = y - 1) \\
\mathcal{IS}_2 \triangleq \mathit{Init}_2 \wedge \Box[\mathit{Next}_2]_{\langle x, y \rangle} \\
\mathcal{S}_2 \triangleq \mathbf{\exists}\, y : \mathcal{IS}_2
\end{array}
$$

Figure 11: The definitions of specification $\mathcal{S}_1$ and $\mathcal{S}_2$.

stop. ER observes that $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$ cannot be verified using their prophecy variables because $\mathcal{S}_1$ doesn't satisfy a condition they call finite internal nondeterminism. We can prove it using the $\exists$ Elimination rule.

Specification $\mathcal{S}_1$ implies that the value of x is bounded, which means that there is some natural number n for which $x \leq n$ is an invariant. This means that the following theorem is true:

(18) $\mathcal{S}_1 \Rightarrow \exists\, n \in \mathit{Nat} : \Box(x \leq n)$

Define $\mathcal{T}_1(n)$ to equal $\mathcal{S}_1 \wedge \Box(x \leq n)$. Formula (18) implies that $\mathcal{S}_1$ equals $\exists\, n \in \mathit{Nat} : \mathcal{T}_1(n)$. By the $\exists$ Elimination rule, this implies that to prove $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$, it suffices to assume $n \in \mathit{Nat}$ and prove $T_1(n) \Rightarrow \mathcal{S}_2$, which can be done with the refinement mapping $\overline{y} \leftarrow n - x$. The proof of $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$ can be made completely rigorous in TLA and presumably in other temporal logics.

In general, we prove $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$ by finding a formula $\mathcal{T}_1(n)$ such that $\mathcal{S}_1$ implies $\exists\, n \in P : \mathcal{T}_1(n)$ for some constant set P, and we then prove $n \in P$ implies $T_1(n) \Rightarrow \mathcal{S}_2$. We can view this method in two ways. The first is that instead of proving $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$ with a single refinement mapping, we prove $\mathcal{T}_1(n) \Rightarrow \mathcal{S}_2$ by using a separate refinement mapping for each value of n. The second is that the constant parameter n is equivalent to a simple prophecy variable that predicts an action that never occurs, so its value never changes. This is because $\exists\, n \in P : \mathcal{T}_1(n)$ is equivalent to

(19) $\mathbf{\exists}\, p : (p \in P) \wedge \Box[p' = p]_p \wedge \mathcal{T}_1(p)$

For our example, this formula is equivalent to

$$\mathbf{\exists}\, p : ((p \in \mathit{Nat}) \wedge \mathit{Init}_1) \wedge \Box[(p' = p) \wedge \mathit{Next}_1]_{\langle x, p \rangle} \wedge \mathit{Stops}$$

This is the formula we get by observing that $Next_1$ is equivalent to

$$Next_1 \lor (\exists\, n \in Nat \,:\, (x = n) \land \text{FALSE})$$

and adding a simple prophecy variable p to predict for which value of n the next $(x = n) \land \text{FALSE}$ step occurs.

When a constant parameter n is used in this way, we call it a *prophecy constant*. The equivalence of (19) and $\exists\, n \in P : \mathcal{T}_1(p)$ means that a verification using a prophecy constant can be done using a simple prophecy variable, but there's no reason to do so.

Prophecy constants are useful for predicting the infinite future—that is, making predictions that depend on the entire behavior. Section 6 of ER provides an example in which they cannot prove $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$ with a refinement mapping because the supplementary property of $\mathcal{S}_2$ implies that the initial value of an internal variable depends on whether or not the behavior terminates, violating a condition they call internal continuity. It is easy to find the refinement mapping by adding a prophecy constant that predicts if the behavior terminates—a prediction about the entire behavior.

7 The Existence of Refinement Mappings

There is a completeness result stating that for any specification $\mathcal{S}_1$ of the form $\exists\, \mathbf{y} : Init \land \Box[Next]_{\langle \mathbf{x},\mathbf{y} \rangle} \land L$, if $\mathcal{S}_1$ implements $\mathcal{S}_2$, then we can add history, prophecy, and stuttering variables to $\mathcal{S}_1$ to obtain an equivalent specification $\mathcal{S}_1^{\mathbf{a}}$ for which there exists a refinement mapping showing that $\mathcal{S}_1^{\mathbf{a}}$ implements $\mathcal{S}_2$. In fact, we can use prophecy constants instead of prophecy variables. We need only assume that the language for defining auxiliary variables and writing proofs is sufficiently expressive. (TLA$^+$ is such a language.)

This result has been known for almost two decades. Our prophecy constants are essentially what Hesselink called *eternity variables* [8]. His proof of completeness for eternity variables can be translated directly to the following proof for prophecy constants.

Let $\mathcal{IS}_1$ and $\mathcal{IS}_2$ be the internal specifications of $\mathcal{S}_1$ and $\mathcal{S}_2$. To simplify the proof, we assume that the next-state action $Next$ of $\mathcal{IS}_1$ allows stuttering steps, replacing it by $Next \lor \text{UC}\,\langle \mathbf{x}, \mathbf{y} \rangle$ if necessary; and we assume $\mathcal{IS}_1$ never halts, adding an infinite stuttering variable if it may halt.[2] Let $\mathcal{IS}_1^h$ be obtained from $\mathcal{IS}_1$ by adding a history variable h that initially equals 1

[2] This also allows us to avoid Hesselink's "preservation of quiescence" assumption.

and is incremented by 1 with every *Next* step. Letting $\sigma_{[i]}$ be the i^{th} state of a behavior σ, specification $\mathcal{IS}_1^h$ equals

$$\exists\, \sigma \in P \,:\, \mathcal{IS}_1^h \wedge \Box(\langle \mathbf{x}, \mathbf{y} \rangle = \sigma_{[h]})$$

where P is the set of all behaviors satisfying $\mathcal{IS}_1^h$. We define a refinement mapping that depends on the specific behavior σ.

Since $\mathcal{S}_1$ implements $\mathcal{S}_2$, for each σ in P there exists a behavior $f(\sigma)$ of $\mathcal{IS}_2$ that σ simulates. We define the refinement mapping for σ so that it maps the state $\sigma_{[h]}$ in the behavior of $\mathcal{IS}_1^h$ to the corresponding state $f(\sigma)_{[g]}$ of $\mathcal{IS}_2$, for some g. In the absence of stuttering steps, g would equal h. To define g in general, we first make the externally visible steps of σ and $f(\sigma)$ match up by adding stuttering steps to σ and/or $f(\sigma)$. Since $\mathcal{IS}_2$ is stuttering insensitive, we can assume that $f(\sigma)$ already has the necessary stuttering steps. We define $\mathcal{IS}_1^{hs}$ by adding a stuttering variable s to $\mathcal{IS}_1^h$ that adds those stuttering steps needed to make the externally visible steps of σ match those of $f(\sigma)$. We can then define g to be a function of h, s, σ, and $f(\sigma)$.

What this proof shows is that prophecy constants allow embedding behavioral reasoning about a specification into state-based reasoning about another specification. That just places a state-based veneer over a behavioral proof, and presents a state-based tool like a model checker with a specification whose states are impossibly complex. It defeats the purpose of refinement mappings, which is to extend the Floyd-Hoare state-based approach to systems.

A prophecy constant makes a single prediction. When prophecy is needed in practice, as with the Afek et al. algorithm, repeated predictions are almost always required. Making multiple predictions with a constant parameter requires encoding some aspect of the system's behavior in the value of that constant. Our prophecy variables allow that to be avoided.

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] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[5] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.

[6] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Math., Vol. 19*, pages 19–32. American Mathematical Society, 1967.

[7] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, January 1990.

[8] Wim H. Hesselink. Eternity variables to prove simulation of specifications. *ACM Trans. Comput. Log.*, 6(1):175–201, 2005.

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

[10] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[11] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003. Also available on the Web via a link at http://lamport.org.

[12] Leslie Lamport and Stephan Merz. Auxiliary variables in TLA+. arXiv:1703.05121 (https://arxiv.org/abs/1703.05121) Also available, together with TLA$^+$ specifications, at http://lamport.azurewebsites.net/tla/auxiliary/auxiliary.html.

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