Prophecy Made Simple

Leslie Lamport, Stephan Merz

Prophecy Made Simple

LESLIE LAMPORT, Microsoft Research, USA
STEPHAN MERZ, University of Lorraine, CNRS, Inria, LORIA, France

Prophecy variables were introduced in the article “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 article in a more mathematical way.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Modal and temporal logics**; **Logic and verification**; **Program verification;**

Additional Key Words and Phrases: Formal specification, state machine, refinement, auxiliary variable

ACM Reference format:
Leslie Lamport and Stephan Merz. 2022. Prophecy Made Simple. *ACM Trans. Program. Lang. Syst.* 44, 2, Article 6 (April 2022), 27 pages.
https://doi.org/10.1145/3492545

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 [13]. 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 article, 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 backward. In practice, they turned out to be difficult to use because reasoning backward from the future is hard. 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 because it involves

Authors’ address: L. Lamport, Microsoft Research, 14820 NE 36th St, Redmond, WA 98052; S. Merz, University of Lorraine, CNRS, Inria Nancy Grand Est, 615 rue du Jardin Botanique, 54600 Villers-lès-Nancy, France; email: stephan.merz@inria.fr.

0164-0925/2022/04-ART6 $15.00
https://doi.org/10.1145/3492545

reasoning forward from the present. It makes simple examples simple and realistic examples not too hard.

We were motivated to take a fresh look at prophecy variables by a 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. Soon after ER was written, TLA was developed. It allowed us to express the concepts developed in ER mathematically, so a specification is a formula and implementation is implication. It also gave us a 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 sketches how a prophecy variable can be used to verify that a concurrent algorithm implements the specification of a linearizable object [10]. Our method should be useful for verifying linearizable specifications of other systems.

Section 6 shows that existential quantification over constants, an operation present in TLA and some other temporal logics, can be used to predict the future. When used in this way, we call it a prophecy constant. Section 7 presents completeness results stating that the refinement mapping required to verify an implementation can always, in theory, be obtained by adding history and stuttering variables and either prophecy constants or our prophecy variables—without assuming the conditions required by ER's prophecy variables. In practice, prophecy constants and prophecy variables complement each other. A concluding section compares our prophecy variables to others inspired by ER.

Our exposition is as informal as we can make it while trying to be rigorous. TLA$^+$ is a complete specification language based on TLA [17]. Most of what we describe here has been explained in excruciating detail for TLA$^+$ users [18]. 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

$$[hr\,:\,12],\ [hr\,:\,1],\ [hr\,:\,2],\ \ldots, \tag{1}$$

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.* For a function f on states, we define UC f to be the action satisfied by a step iff it leaves the value of f unchanged. We enclose tuples in angle brackets $\langle\ \rangle$.

Let $\mathbf{x}$ be the list $x_1, \ldots, x_n$ of all variables of the specification. Then UC $\langle \mathbf{x} \rangle$ is the action satisfied only by steps that leave all the variables $\mathbf{x}$ unchanged—that is, stuttering steps. 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 $Next \lor \text{UC}\,\langle \mathbf{x} \rangle$.

The disjunct 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:

$$Init \ \wedge \ \Box[Next]_{\langle \mathbf{x} \rangle}, \tag{2}$$

where $\Box$ is the temporal logic *forever* operator and $[Next]_f$ is an abbreviation for $Next \vee (\text{UC}\ f)$.

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, $\text{UC}\langle \mathbf{x} \rangle$ equals $\langle \mathbf{x} \rangle' = \langle \mathbf{x} \rangle$, which is equivalent to $(x_1' = x_1) \wedge \ldots \wedge (x_n' = x_n)$. (Priming an expression means priming all its variables.) Our hour-clock specification can be written in TLA as

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

(The angle brackets in the subscript $\langle hr \rangle$ can be omitted.) A specification of the form (2) allows behaviors in which, at some point, the values of the variables $\mathbf{x}$ never again change—that is, in our example, 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

$$Init \ \wedge \ \Box[Next]_{\langle \mathbf{x} \rangle} \ \wedge \ L. \tag{3}$$

Letting L be the TLA weak fairness formula $\text{WF}_{\langle \mathbf{x} \rangle}(Next)$ makes Formula (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. The precise meaning of WF is irrelevant.

Safety and liveness properties are verified differently, so it is best to keep them separate in a specification. The liveness property L plays no part in defining our prophecy variables, so we don't care how L is written. We don't even require it to be a liveness property. Following ER, we call L 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 $\boldsymbol{\exists}$. The approximate definition is that $\boldsymbol{\exists}\, y : \mathcal{F}$ 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 $\boldsymbol{\exists}\, y : \mathcal{F}$ is stuttering insensitive. The correct definition is that σ satisfies $\boldsymbol{\exists}\, y : \mathcal{F}$ iff there is a behavior τ stuttering equivalent for $\mathcal{F}$ to σ and assignments of values to y that make τ satisfy $\mathcal{F}$. For a list $\mathbf{y}$ of variables $y_1, \ldots\ y_m$, we define $\boldsymbol{\exists}\, \mathbf{y} : \mathcal{F}$ to equal $\boldsymbol{\exists}\, y_1 : \ldots \boldsymbol{\exists}\, y_m : \mathcal{F}$.

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

$$\mathcal{IS} \ \stackrel{\Delta}{=} \ Init \ \wedge \ \Box[Next]_{\langle \mathbf{x}, \mathbf{y} \rangle} \ \wedge \ L, \tag{4}$$

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

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

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

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 $\mathcal{S}$.

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 $\mathcal{A}$:

$$
\begin{array}{l}
[in : \texttt{rdy},\, out : 0],\; [in : 3,\, out : 0],\; [in : \texttt{rdy},\, out : 3], \\
{[in : -2,\, out : 3]},\; [in : \texttt{rdy},\, out : \tfrac{1}{2}],\; \ldots .
\end{array} \tag{5}
$$

$\mathcal{A}$ 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 $\mathcal{A}$:

$$
\begin{array}{l}
[in : \texttt{rdy},\, out : 0,\, num : 0,\, sum : 0], \\
{[in : 3,\, out : 0,\, num : 0,\, sum : 0]}, \\
{[in : \texttt{rdy},\, out : 3,\, num : 1,\, sum : 3]}, \\
{[in : -2,\, out : 3,\, num : 1,\, sum : 3]}, \\
{[in : \texttt{rdy},\, out : \tfrac{1}{2},\, num : 2,\, sum : 1]},\; \ldots .
\end{array} \tag{6}
$$

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

$$
\begin{aligned}
\mathcal{B} &\triangleq \exists\!\!\exists\, seq : \mathcal{IB} \\
\mathcal{IB} &\triangleq Init_{\mathcal{B}} \land \Box[Next_{\mathcal{B}}]_{\langle in, out, seq \rangle} \\
Init_{\mathcal{B}} &\triangleq (in = \mathsf{rdy}) \land (out = 0) \land (seq = \langle\, \rangle) \\
Next_{\mathcal{B}} &\triangleq Input_{\mathcal{B}} \lor Output_{\mathcal{B}} \\
Input_{\mathcal{B}} &\triangleq \quad (in = \mathsf{rdy}) \land (in' \in Int) \\
&\quad \land\ (seq' = Append(seq, in')) \land (out' = out) \\
Output_{\mathcal{B}} &\triangleq \quad (in \neq \mathsf{rdy}) \land (in' = \mathsf{rdy}) \\
&\quad \land\ (out' = Sum(seq)/Len(seq)) \land (seq' = seq)
\end{aligned}
$$

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

variable seq that records the entire sequence of inputs received so far. Specification $\mathcal{B}$ has the same form 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. As mentioned above, we enclose sequences in angle brackets, 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 $(\exists\!\!\exists\, seq : \mathcal{IB}) \Rightarrow \mathcal{A}$. The quantifier $\exists\!\!\exists$ 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 $(\exists\!\!\exists\, seq : \mathcal{IB}) \Rightarrow \mathcal{A}$ it suffices to show $\mathcal{IB} \Rightarrow \mathcal{A}$.

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 $\exists\!\!\exists\, num, sum : \mathcal{IA}$, 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 $\mathcal{IB} \Rightarrow \exists\!\!\exists\, num, sum : \mathcal{IA}$, 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 $\boxed{Next_{\mathcal{B}} \vee \text{UC}\,\langle in, out, seq\rangle}$, then the pair of states

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

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

Because $\overline{num}$ and $\overline{sum}$ contain only the variables in, out, and seq of $\mathcal{IB}$, if the step s, t satisfies 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 UC $\langle in, out, num, sum\rangle$. Therefore, RM2 is automatically satisfied if the step s, t satisfies UC $\langle in, out, seq\rangle$. This means we can simplify RM2 to:

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

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

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

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

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

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

$$Init_{\mathcal{A}} \textbf{ with } num \leftarrow \overline{num}, sum \leftarrow \overline{sum}.$$

RM1 asserts that every state satisfying $Init_{\mathcal{B}}$ satisfies (7). Therefore, it is equivalent to

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

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 $(Init_{\mathcal{A}} \textbf{ with } \ldots)$ 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, we see that RM2 is equivalent to

RM2. $Next_{\mathcal{B}} \;\Rightarrow$

$$(Next_{\mathcal{A}} \textbf{ with } num \leftarrow \overline{num}, sum \leftarrow \overline{sum}) \vee \text{UC}\,\langle in, out, \overline{num}, \overline{sum}\rangle.$$

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 $\mathcal{IA}$ are 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 formulas:

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

$$Output_{\mathcal{B}} \;\Rightarrow\; (Output_{\mathcal{A}} \textbf{ with } num \leftarrow \overline{num},\ sum \leftarrow \overline{sum}). \tag{9}$$

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:

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

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

$$\begin{aligned}&(in = \mathsf{rdy}) \wedge (out = 0) \wedge (seq = \langle\,\rangle) \;\Rightarrow\\ &\qquad (in = \mathsf{rdy}) \wedge (out = \overline{num} = \overline{sum} = 0).\end{aligned}$$

It's not hard to verify Formula (8), since $Input_{\mathcal{B}}$ implies $Front(seq') = seq$. Equation (9) may also appear valid, but it's not. For example, there's no way to show that Formula (9) is true if $in' = 42$ and $seq = \langle \mathsf{rdy} \rangle$, since we don't know what $Sum(\langle \mathsf{rdy} \rangle)$ and $Sum(\langle \mathsf{rdy}, 42 \rangle)$ equal.

It may seem obvious that seq can't equal $\langle \mathsf{rdy} \rangle$, but why can't it? Nothing in Formula (9) or Figure 2 asserts that seq doesn't equal $\langle \mathsf{rdy} \rangle$. What is true is that the value of seq can't equal $\langle \mathsf{rdy} \rangle$ 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:

$$\begin{aligned}Inv \;\triangleq\;& (in \in Int \cup \{\mathsf{rdy}\}) \wedge (out \in Int) \wedge (seq \in Int^*) \;\wedge\\ &\quad ((in \neq rdy) \Rightarrow (seq \neq \langle\,\rangle) \wedge (in = Last(seq))),\end{aligned}$$

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:

$$\begin{aligned} &Inv \wedge Inv' \wedge Next_{\mathcal{B}} \Rightarrow \\ &\quad (Next_{\mathcal{A}} \textbf{ with } num \leftarrow \overline{num}, sum \leftarrow \overline{sum}) \vee \text{UC}\,\langle in, out, \overline{num}, \overline{sum}\rangle. \end{aligned} \tag{10}$$

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 [9, 12] and Owicki-Gries [21] 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

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

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. $Init_1 \Rightarrow (Init_2 \textbf{ with } \mathbf{z} \leftarrow \overline{\mathbf{z}})$,
RM2. $Inv \wedge Inv' \wedge Next_1 \Rightarrow ((Next_2 \textbf{ with } \mathbf{z} \leftarrow \overline{\mathbf{z}}) \vee \text{UC}\,\langle \mathbf{x}, \overline{\mathbf{z}}\rangle)$,
RM3. $Init_1 \wedge \Box[Next_1]_{\langle \mathbf{x},\mathbf{y}\rangle} \wedge L_1 \Rightarrow (L_2 \textbf{ with } \mathbf{z} \leftarrow \overline{\mathbf{z}})$.

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 Formula (4). We define $\mathcal{S}^a$ to equal $\exists\, \mathbf{y} : \mathcal{IS}^a$ and define

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

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

$$Init^a \;\triangleq\; Init \land J, \tag{13}$$

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, if U, V, and $W(i)$ are actions, we can consider the elementary actions of

$$U \;\lor\; V \;\lor \exists\, i \in Int \;:\; W(i) \tag{14}$$

to be U, V, and all $W(i)$ with $i \in Int$. (We could also consider $U \lor V$ and $\exists\, i \in Int : W(i)$ to be the elementary actions of Formula (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

$$A^a \;\triangleq\; A \land B, \tag{15}$$

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 Formulas (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 $\mathcal{IA}^h \Rightarrow \boldsymbol{\exists}\, seq : \mathcal{IB}$, 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

$$Init^h_{\mathcal{A}} \;\triangleq\; Init_{\mathcal{A}} \land (h = \langle\,\rangle).$$

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:

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

Finally, we define

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

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:

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

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

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

$$Init^h \triangleq Init \land (h = f),$$

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

$$A^h \triangleq A \land (h' = F),$$

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 consider a specification $\mathcal{C}$ that models implementing $\mathcal{A}$ with speculative execution, where an input can either produce an output or else be "undone" by resetting in to `rdy` without changing any other variables. Such a specification cannot implement $\mathcal{A}$, which requires that every input produces an output. However, it does model a specification $\widetilde{\mathcal{A}}$ that is the same as $\mathcal{A}$ except with the variable in hidden.

We therefore let $\widetilde{\mathcal{A}}$ equal $\exists\, in : \mathcal{A}$, 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 produces 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

$$\begin{aligned}
C &\triangleq \exists\, in, num, sum \,:\, \mathcal{IC} \\
\mathcal{IC} &\triangleq Init_{\mathcal{A}} \land \Box[Next_C]_{\langle out, in, num, sum \rangle} \\
Next_C &\triangleq Next_{\mathcal{A}} \lor Undo_C \\
Undo_C &\triangleq (in \neq \mathsf{rdy}) \land (in' = \mathsf{rdy}) \land \text{UC}\,\langle out, num, sum\rangle
\end{aligned}$$

Fig. 3. The definition of specification C.

$$\begin{aligned}
C^p &\triangleq \exists\, in, num, sum \,:\, \mathcal{IC}^p \\
\mathcal{IC}^p &\triangleq Init^p_C \land \Box[Next^p_C]_{\langle out, in, num, sum, p \rangle} \\
Init^p_C &\triangleq (p \in \{\mathsf{do}, \mathsf{undo}\}) \land Init_{\mathcal{A}} \\
Next^p_C &\triangleq Input^p_{\mathcal{A}} \lor Output^p_{\mathcal{A}} \lor Undo^p_C \\
Input^p_C &\triangleq (p' = p) \land Input_{\mathcal{A}} \\
Output^p_C &\triangleq (p = \mathsf{do}) \land (p' \in \{\mathsf{do}, \mathsf{undo}\}) \land Output_{\mathcal{A}} \\
Undo^p_C &\triangleq (p = \mathsf{undo}) \land (p' \in \{\mathsf{do}, \mathsf{undo}\}) \land Undo_C
\end{aligned}$$

Fig. 4. The definition of specification C^p.

input value will be output. More precisely, its value predicts whether the next $Output_{\mathcal{A}} \lor Undo_C$ step will be an $Output_{\mathcal{A}}$ step or an $Undo_C$ step. The initial predicate makes the first prediction. The next prediction is made after the currently predicted $Output_{\mathcal{A}}$ or $Undo_C$ step occurs. The specification 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 $Undo_C$ step must be an $Output_{\mathcal{A}}$ step; it must be an $Undo_C$ 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:

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

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 $(p = i) \land (p' \in P) \land A_i$,
(3) Replacing each other elementary action B by $(p' = p) \land B$.

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

The examples we use to illustrate prophecy variables are unrealistically simple because input steps are internal. This makes it possible to define the necessary refinement mappings using the stuttering variables introduced in Section 4.7 instead of prophecy variables. We have no reason to believe stuttering variables can replace prophecy variables in any realistic example.

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? This would appear to cause unsoundness, because it seems that $\exists\, p : \mathcal{S}^p$ and $\mathcal{S}$ couldn't be equivalent. But they would be equivalent, and understanding why helps understand our prophecy variables. 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 $(x' = x + 1) \vee \textsc{false}$, we can rewrite this as

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

Following the procedure above, we add a prophecy variable p that predicts if the next non-stuttering step (i.e., the next $(x' = x + 1) \vee \textsc{false}$ step) is an $x' = x + 1$ step or a $\textsc{false}$ step:

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

If p ever becomes equal to $\texttt{stop}$, then no further Next^p step is possible (since no step can satisfy $\textsc{false}$), 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 $\mathrm{WF}_{\langle x \rangle}(x' = x + 1)$. In that case, $\mathcal{S}^p$ becomes

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

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

Readers who find this equivalence puzzling may be confusing the next-state action $\textsc{false}$ with the specification $\textsc{false}$. The specification $\textsc{false}$ is satisfied by no behavior; the specification $\Box[\textsc{false}]_{\langle x \rangle}$ allows any behavior in which the value of x never changes.

Readers who find the specification (17) weird are not confused. 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, num, 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, $\mathit{Head}(\mathit{sq})$ is the first element of sq and $\mathit{Tail}(\mathit{sq})$ is the sequence obtained from sq by removing its first element, with $\mathit{Tail}(\langle\,\rangle) = \langle\,\rangle$.

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

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

$$\begin{aligned}
\mathcal{E} &\triangleq \boldsymbol{\exists}\, inq, num, sum : \mathcal{IE} \\
\mathcal{IE} &\triangleq Init_{\mathcal{D}} \land \Box[Next_{\mathcal{E}}]_{\langle inq,out,num,sum \rangle} \\
Next_{\mathcal{E}} &\triangleq Next_{\mathcal{D}} \lor Undo_{\mathcal{E}} \\
Undo_{\mathcal{E}} &\triangleq (inq \neq \langle\,\rangle) \land (inq' = Tail(inq)) \land \text{UC}\,\langle out, sum, num \rangle
\end{aligned}$$

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

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

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

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:

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

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

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

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, and let P^n be the set of all length n sequences of elements of 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 $p \in P^n$ to the initial predicate $Init$, for some $n \geq 0$. (Note that $p \in P^0$ is equivalent to $p = \langle\,\rangle$.)
(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 $s_1^p, s_2^p, \ldots$ 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 $Output_{\mathcal{E}}^p$ and $Undo_{\mathcal{E}}^p$ because $\mathcal{IE}^p$ implies that p is always the same length as inq, and $Output_{\mathcal{D}}$ and $Undo_{\mathcal{E}}$ both imply $inq \neq \langle\,\rangle$.

In condition 1, n can even equal ∞, where P^∞ is the set of all infinite sequences of elements of P. In that case, condition 3 requires replacing B by $(p' = p) \wedge B$, since one cannot append an element to an infinite sequence. Such a prophecy variable, which makes infinitely many initial predictions that can unfold forever, is used in the completeness proof of Section 7.

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:

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

Fig. 9. The definition of specification $\mathcal{G}$.

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

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

$EmptyFcn$ 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 $s^p_1, s^p_2, \ldots$ a behavior having all the right predictions. Specification $\mathcal{IG}^p$ implements $\mathcal{IF}$ under this refinement mapping:

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

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

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. The generalization is described in detail in [18]. 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 predictions and/or make new predictions. An action that fulfills a prediction must remove that prediction. Except for that requirement, actions may leave all predictions unchanged.

Whether 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 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. A none value is handled like the value $\langle\,\rangle$ of the prophecy sequence variables of Section 4.4.

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. As an example, let $\mathcal{S}_2$ have an internal variable q, whose value is a sequence, and an action A that sets q to $Tail(q)$. Let A equal

$$E \wedge (q' = Tail(q)) \wedge \mathrm{UC}\,\langle \mathbf{w} \rangle,$$

where E is an enabling condition that implies $q \neq \langle\,\rangle$, and $\mathbf{w}$ is the list of all internal and external variables of $\mathcal{S}_2$ except q.

Let $\mathcal{S}_1$ implement $\mathcal{S}_2$ by representing q with an array variable a, where the value of q is the sequence of elements $a[0], \ldots, a[Len(q)-1]$, and $a[j]$ equals a special value $null$ for $j \geq Len(q)$. (We assume that a is a large enough array—perhaps infinite.) Let an A step of $\mathcal{S}_2$ be implemented with a variable i, initially equal to 0, by steps setting $a[i]$ to $a[i+1]$ for i equal to 0 through $Len(q)-1$, the last step resetting i to 0. To show that $\mathcal{S}_1$ implements $\mathcal{S}_2$, we might use a refinement mapping in which an A step of $\mathcal{S}_2$ is implemented by the first of those $Len(q)$ steps of $\mathcal{S}_1$.

If a and i are internal variables of $\mathcal{S}_1$, then $\mathcal{S}_2$ should implement $\mathcal{S}_1$. Since $\mathcal{S}_1$ takes more steps than $\mathcal{S}_2$ to implement an A step, defining a refinement mapping to show that $\mathcal{S}_2$ implements $\mathcal{S}_1$ requires an auxiliary variable that adds stuttering steps to $\mathcal{S}_2$ that implement the additional steps taken by $\mathcal{S}_1$. That variable must add $Len(q)-1$ stuttering steps for each A step.

ER made their prophecy variables more complicated so they could add stuttering steps; we have long felt it was easier instead to use a new kind of auxiliary variable. We introduce a variable s that can add stuttering steps before and/or after an action, in this case the action A. An easy way to do it is to let the value of 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 the number of steps remaining. Let *Init* and *Next* be the initial predicate and next-state actions of $\mathcal{S}_2$. We assume *Next* is written $A \vee B_1 \vee \ldots \vee B_n$, where for each j, no step is both an A and B_j step. We add the stuttering variable s to $\mathcal{S}_2$ as follows to obtain the specification $\mathcal{S}_2^s$ that adds $Len(q)-1$ steps after each A step. The initial predicate $Init^s$ and next-state action $Next^s$ of $\mathcal{S}_2^s$ are

$$\begin{aligned} Init^s &\triangleq Init \wedge (s = 0) \qquad Next^s \triangleq A^s \vee B_1^s \vee \ldots \vee B_n^s, \\ A^s &\triangleq \quad ((s = 0) \wedge (s' = Len(q) - 1) \wedge A) \\ &\quad\ \vee ((s > 0) \wedge (s' = s - 1) \wedge \mathrm{UC}\,\langle \mathbf{w}, q \rangle), \\ B_j^s &\triangleq (s = s' = 0) \wedge B_j, \text{ for } j = 1, \ldots, n. \end{aligned}$$

Defining the refinement mapping under which $\mathcal{S}_2^s$ implements $\mathcal{S}_1$ is tricky. The reader can verify that if A were the only action that changed q, then the correct refinement mapping would include these substitutions, where $q[k]$ equals the kth element of q:

$$\begin{array}{l} i \leftarrow iBar \\ a[j] \leftarrow \textbf{if } s = 0 \\ \qquad\qquad \textbf{then if } j < Len(q) \textbf{ then } q[j+1] \textbf{ else } null \\ \qquad\qquad \textbf{else if } j < iBar \textbf{ then } q[j+1] \\ \qquad\qquad\qquad\qquad \textbf{else if } j \leq Len(q) \textbf{ then } q[j] \textbf{ else } null \\ \quad \text{where } iBar \triangleq \textbf{ if } s = 0 \textbf{ then } 0 \textbf{ else } Len(q) + 1 - s. \end{array}$$

To add the $Len(q) - 1$ stuttering steps *before* every A step, we just change the definition of A^s to:

$$\begin{array}{rl} A^s \triangleq & ((s = 0) \land E \land (s' = Len(q) - 1) \land \text{UC}\langle \mathbf{w}, q\rangle) \\ & \lor\ ((s > 1) \land (s' = s - 1) \land \text{UC}\langle \mathbf{w}, q\rangle) \\ & \lor\ ((s = 1) \land (s' = 0) \land (q' = Tail(q)) \land \text{UC}\langle \mathbf{w}\rangle). \end{array} \tag{18}$$

Finding a refinement mapping with this stuttering variable is another tricky exercise.

The generalizations that add K stuttering steps before or after each A step, for an arbitrary state function K and action A, are straightforward. To add the steps after A, we modify the definition above by replacing $Len(q) - 1$ with K and $\mathbf{w}, q$ with the list of all variables of $\mathcal{S}_2$. To add the stuttering steps before the A step, we make one additional change to the corresponding definition of A^s: replacing $(q' = Tail(q)) \land \text{UC}\langle \mathbf{w}\rangle$ with F, where F is an action such that $A \equiv E \land F$ and F is enabled in every state in which E is true.

We don't have to use natural numbers for counting stuttering states. 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—e.g., for an additional J stuttering steps after action A, we can replace the last disjunct of Formula (18) by

$$\begin{array}{l} \lor\ ((s = 1) \land (s' = -J) \land (q' = Tail(q)) \land \text{UC}\langle \mathbf{w}\rangle) \\ \lor\ ((s < 0) \land (s' = s + 1) \land \text{UC}\langle \mathbf{w}, q\rangle). \end{array}$$

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 $s' = Tail(s)$.

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 $\langle \text{“}A\text{”}, i\rangle$ for a number i, and add stuttering steps to an action B by letting s assume values of the form $\langle \text{“}B\text{”}, q\rangle$ for q a sequence of processes.

To handle the rare 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. The formula $\text{WF}_{\langle s\rangle}(s' \neq s)$ asserts that there are infinitely many steps in which the value of s changes.[2] We add the auxiliary variable s by conjoining to the supplementary property of $\mathcal{S}_1$ the temporal logic tautology $\exists\, s : \text{WF}_{\langle s\rangle}(s' \neq s)$. An infinite stuttering variable is our only auxiliary variable that is added by modifying the supplementary property rather than the initial predicate and next-state action.

[2]We assume that variables can take more than one value.

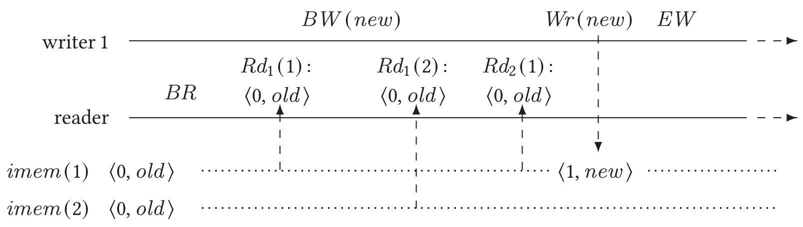

writer 1
$BW(new)$
$Wr(new)$
EW
reader
BR
$Rd_1(1)$: $\langle 0, old\rangle$
$Rd_1(2)$: $\langle 0, old\rangle$
$Rd_2(1)$: $\langle 0, old\rangle$
$imem(1)$ $\langle 0, old\rangle$
$\langle 1, new\rangle$
$imem(2)$ $\langle 0, old\rangle$

Fig. 11. The common prefix of behaviors σ and τ.

5 VERIFYING LINEARIZABILITY

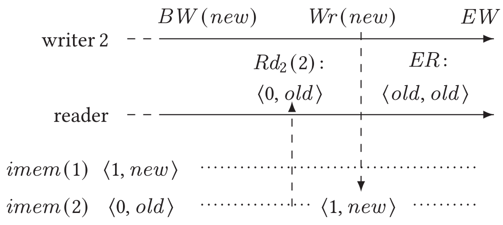
Linearizability has become a standard way of specifying an object shared by multiple processes [10]. A process's operation Op is described by a sequence of three steps: a $BeginOp$ step that provides the operation's input, a $DoOp$ step that performs the operation by reading and/or modifying the object, 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; a precise exposition including formal TLA$^+$ specifications is in [18]. 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.

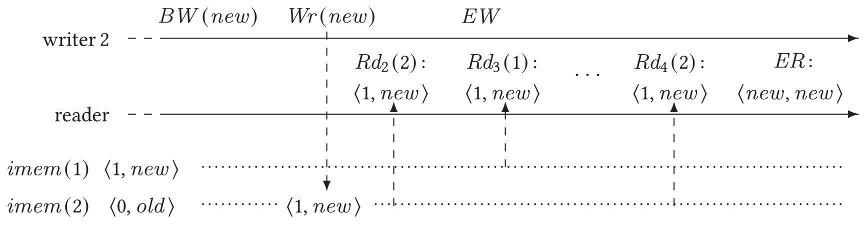
We consider a simplified version of the Afek et al. snapshot algorithm we call $SimpleAfek$. It maintains an internal variable $imem$. A writer w writes a value v on its i^{th} write by setting $imem(w)$ to the pair $\langle i, v\rangle$. A reader does a sequence of reads of $imem$, each of those reads 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 algorithm adds 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. The more complicated algorithm 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. We show why with the following example, illustrated in Figures 11 to 13. There are two writers, named 1 and 2, and one reader. We describe two behaviors σ and τ of $SimpleAfek$ and consider how a refinement mapping maps steps of those behaviors to $DoRead$ and $DoWrite$ steps of $LinearSnap$. As shown in Figure 11, both behaviors begin in a state having $imem(1) = imem(2) = \langle 0, old\rangle$ for some value old and with the reader performing a $BeginRead$ step (labeled BR) and beginning its sequence of reads of the complete array $imem$ with individual reads obtaining the value $\langle 0, old\rangle$ for the first reads $Rd_1(1)$ of $imem(1)$ and $Rd_1(2)$ of $imem(2)$, as well as for the second read of $imem(1)$. Meanwhile, writer 1 performs a



$BW(new)$
$Wr(new)$
EW
writer 2
$Rd_2(2)$: $\langle 0, old \rangle$
ER: $\langle old, old \rangle$
reader
$imem(1)$ $\langle 1, new \rangle$
$imem(2)$ $\langle 0, old \rangle$
$\langle 1, new \rangle$

Fig. 12. Suffix of behavior σ.



$BW(new)$
$Wr(new)$
EW
writer 2
$Rd_2(2)$: $\langle 1, new \rangle$
$Rd_3(1)$: $\langle 1, new \rangle$
...
$Rd_4(2)$: $\langle 1, new \rangle$
ER: $\langle new, new \rangle$
reader
$imem(1)$ $\langle 1, new \rangle$
$imem(2)$ $\langle 0, old \rangle$
$\langle 1, new \rangle$

Fig. 13. Suffix of behavior τ.

BeginWrite step $BW(new)$ of the value *new*. It then writes value *new* to $imem(1)$, updating it to $\langle 1, new \rangle$, followed by an *EndWrite* step.

As illustrated in Figure 12, in behavior σ the reader then completes its second read, finding $imem(2) = \langle 0, old \rangle$. Concurrently, writer 2 writes *new* to its register, the write of $imem(2)$ occurring after the second read of $imem(2)$. Since the first two reads see the same values of *imem*, the read completes and returns the snapshot with $mem(1) = mem(2) = old$. Because the reader obtained the snapshot before the writes, we have:

A. For behavior σ, the step of *SimpleAfek* that implements the *DoRead* step must precede the step that implements writer 1's *DoWrite* step.

As shown in Figure 13, in behavior τ the second read of $imem(2)$ is preceded by writer 2 updating $imem(2)$ to $\langle 1, new \rangle$. The reader then continues its operation, finding $imem(2) = \langle 1, new \rangle$. Because the first two reads obtained different values of $imem(2)$, the reader continues reading. Its third and fourth reads find $imem(1) = imem(2) = \langle 1, new \rangle$, so the read completes and returns the snapshot with $mem(1) = mem(2) = new$. Since that snapshot contains the value written by writer 1, we have:

B. For behavior τ, the step of *SimpleAfek* that implements the *DoRead* step must follow the steps that implement the *DoWrite* steps of both writers.

Observations A and B imply that to handle the behavior σ correctly, the refinement mapping must know that the current behavior is σ and not τ before the two behaviors have diverged. This is possible only if something in the state predicts the future—i.e., only if we add a prophecy variable.

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 during a read operation by reader r, the value of $isnap(r)$ is the sequence of values that mem had since the operation began. These values are the snapshots that $LinearSnap$ allows the read to 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 of reader r. 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.

If $p(r)$ is set to 1, predicting that r will return the value mem had when the $BeginRead$ step occurred, then the refinement mapping will cause the $DoRead$ step of $LinearSnap$ to occur right after the $BeginRead$ step. If $p(r)$ is set to a number greater than 1, predicting that r will return the value of mem after the $(p(r) - 1)^{\text{st}}$ $DoWrite$ action since the read began, then the refinement mapping will cause the $DoRead$ step to occur immediately after that $DoWrite$ step. The step that implements the $DoRead$ step of $LinearSnap$ is added as follows as a stuttering step of $NewLinearSnap$.

We introduce a stuttering variable that adds a single stuttering step after r's $BeginRead$ action if $p(r) = 1$, and that adds stuttering steps after each $DoWrite$ action—one stuttering step for every read r for which the write adds the $p(r)^{\text{th}}$ element to $isnap(r)$. To add the stuttering step after a $BeginRead$ step, the stuttering variable simply counts down from 1. To add any necessary 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 $DoRead$ steps of $LinearSnap$ makes it clear how to define the refinement mapping.

The verification that $SimpleAfek$ implements $LinearSnap$, which requires a prophecy variable, has been split into two steps: verifying that the intermediate specification $NewLinearSnap$ implements $LinearSnap$ and is implemented by $SimpleAfek$. Only the first step requires a prophecy variable. This approach can also be applied to an example in Herlihy and Wing's article defining linearizability [10]: an algorithm that implements a linearizable specification of a queue and requires predicting the future to construct a refinement mapping. The intermediate specification replaces the totally ordered queue of the original specification with a partially ordered set, where the partial order constrains which items may be dequeued. A refinement mapping showing that the new specification implements the original one can be defined using a prophecy variable that predicts the order in which items will be dequeued.

Verifying linearizability is an important problem. Liang and Feng devised a method for doing it [19], and Chakraborty et al. developed an elaborate theory just for proving linearizability of queue algorithms [6]. Our method of decomposing the verification has two obvious advantages. Adding a prophecy variable is not trivial—especially for complex algorithms. It is easier to add it to the simpler intermediate specification. Second, the same intermediate specification can be used for

$$
\begin{array}{lll}
Init_1 \triangleq x = 0 & & Next_1 \triangleq x' = x + 1 \\
Stops \triangleq \Diamond\Box[x' = x]_{\langle x \rangle} & & \\
\mathcal{S}_1 \triangleq Init_1 \land \Box[Next_1]_{\langle x \rangle} \land Stops & &
\end{array}
$$

$$
\begin{array}{lcl}
Init_2 & \triangleq & (x = 0) \land (y \in Nat) \\
Next_2 & \triangleq & (y > 0) \land (x' = x + 1) \land (y' = y - 1) \\
\mathcal{IS}_2 & \triangleq & Init_2 \land \Box[Next_2]_{\langle x,y \rangle} \\
\mathcal{S}_2 & \triangleq & \exists\!\!\exists\, y : \mathcal{IS}_2
\end{array}
$$

Fig. 14. The definitions of specification $\mathcal{S}_1$ and $\mathcal{S}_2$.

multiple implementations. There is a third advantage that suggests the approach should be widely applicable: the intermediate specification is likely to be useful.

A specification might be written for implementors of a system or for its users. A specification written for implementors should not contain in its state information that is not needed by an implementation. The need for auxiliary variables to define a refinement mapping means that its state does contain unnecessary information. However, it could be a better specification for users. It's easier for a user to think of a queue as being totally ordered; an implementor should know that a partial order suffices. Writing an intermediate specification with no unnecessary state can be a useful part of the system-design process.

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 $SimpleAfek$ specification are examples of constant parameters. Constant parameters can also be used to make predictions. Moreover, no new rules are required to use them in this way. The ordinary rules of logic suffice.

What we call a variable is called a *flexible variable* by logicians because its value can vary during the course of a behavior. They call a constant parameter a *rigid variable* because its value remains the same throughout a behavior. 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 $\exists\, n : (n \in P) \land \mathcal{F}$, 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 14, 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 that increment x a finite number of times (possibly 0 times) and then 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:

$$\mathcal{S}_1 \;\Rightarrow\; \exists\, n \in Nat \,:\, \Box(x \leq n). \tag{19}$$

Define $\mathcal{T}_1(n)$ to equal $\mathcal{S}_1 \wedge \Box(x \leq n)$. Formula (19) implies that $\mathcal{S}_1$ equals $\exists\, n \in Nat \,:\, \mathcal{T}_1(n)$. By the $\exists$ Elimination rule, this implies that to prove $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$, it suffices to assume $n \in Nat$ and prove $\mathcal{T}_1(n) \Rightarrow \mathcal{S}_2$. This 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 $\mathcal{T}_1(n) \Rightarrow \mathcal{S}_2$. 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 n is a constant that predicts the value x will have when the execution stops (stutters forever). We take the latter view and call n a *prophecy constant.*

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 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.

The completeness results of the next section show that, in principle, we can use prophecy constants instead of prophecy variables, and vice versa. In practice, it seems that we should use prophecy variables to predict safety and prophecy constants to predict liveness. A prophecy variable is good for predicting which of multiple possibilities will be allowed to happen; a prophecy constant is good for predicting what will eventually happen.

"Prophecy constants" is just a new name for the bound rigid variables of a temporal logic. The observation that those bound variables can be used for defining refinement mappings appears not to have been published before. Hesselink's eternity variables [11] are essentially a special case of prophecy constants, except based on reasoning directly about sequences of states using history variables rather than on temporal logic. Temporal logic was introduced to computer science by Pnueli to abstract that kind of reasoning [24].

7 THE EXISTENCE OF REFINEMENT MAPPINGS

We now present two completeness results. The first states that for any specification $\mathcal{S}_1$ of the form $\boldsymbol{\exists}\,\mathbf{y} : Init \wedge \Box[Next]_{\langle \mathbf{x},\mathbf{y} \rangle} \wedge L$, if $\mathcal{S}_1$ implements $\mathcal{S}_2$, then we can add history, stuttering, and prophecy variables to $\mathcal{S}_1$ to obtain an equivalent specification $\mathcal{S}_1^{\mathrm{a}}$ for which there exists a refinement mapping showing that $\mathcal{S}_1^{\mathrm{a}}$ implements $\mathcal{S}_2$. The second result is the same except for prophecy constants rather than prophecy variables.

These results require only the assumption that the language for defining auxiliary variables and writing proofs is sufficiently expressive. (TLA$^+$ is such a language.) We first sketch the proof for prophecy constants. It is almost the same as Hesselink's proof of completeness for eternity variables [11]. We then indicate how the proof is modified for prophecy variables. In the proofs, we let a specification's state be an assignment of values to that specification's variables, and we let a behavior of a specification be the sequence of specification states of a behavior satisfying the specification.

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.[3] Let $\mathcal{IS}_1^h$ be obtained from $\mathcal{IS}_1$ by adding a history variable h that initially equals 1 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

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

where P is the set[4] of all behaviors of $\mathcal{IS}_1^h$. We define a refinement mapping using σ as a prophecy constant.

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 σ and $f(\sigma)$ match up by adding stuttering steps to σ and/or $f(\sigma)$. Since our specifications are stuttering insensitive, we can choose f so that $f(\sigma)$ already has the necessary stuttering steps. Since σ is an arbitrary behavior of $\mathcal{IS}_1$, we may have to add stuttering steps to it to make the externally visible steps of σ match those of $f(\sigma)$. We do that by adding a stuttering variable s to $\mathcal{IS}_1^h$. We can then define g to be a function of h, s, σ, and $f(\sigma)$.

We now show how to use a prophecy variable p instead of the prophecy constant σ to construct the refinement mapping. Let $\mathcal{IS}_1^h$ be as above. Let $\mathcal{IS}_1^{hj}$ be the specification obtained by adding a history variable j to $\mathcal{IS}_1^h$ whose value is the sequence of $\mathcal{IS}_1$ states reached thus far during the current behavior of $\mathcal{IS}_1^h$. Initially, j contains a single element that equals the initial state of $\mathcal{IS}_1$; and each $Next^h$ step appends the new $\mathcal{IS}_1$ state to j. At any point during an execution of $\mathcal{IS}_1^{hj}$, the value of j is the part of the behavior of $\mathcal{IS}_1$ executed thus far, and its length always equals h. We will define p to be a prophecy variable whose value is an infinite sequence of $\mathcal{IS}_1$ states, predicted to be the rest of the behavior whose prefix is the current value of j. Thus, the complete behavior σ is a function of the current state. We can then define the necessary refinement mapping g the same way we did for the prophecy constant σ.

To complete our proof sketch, we show how the prophecy variable p is defined and how to define the behavior σ of $\mathcal{IS}_1$ as a function of the current values of j and p. Let Σ be any set of $\mathcal{IS}_1$ states containing all its reachable states. We then define the specification $\mathcal{IS}_1^{hjp}$ obtained by adding the prophecy variable p to $\mathcal{IS}_1^{hj}$ as follows: the initial value of p is any infinite sequence of elements of Σ, and the next-state action of $\mathcal{IS}_1^{hjp}$ is

$$Next^{hj} \land (Head(p) = \langle \mathbf{x}', \mathbf{y}'\rangle) \land (p' = Tail(p)).$$

The behavior σ is defined to equal the concatenation of j and p, which equals its initial value throughout a behavior of $\mathcal{IS}_1^{hjp}$. Define a step of σ to be a correct prediction if it satisfies the next-state relation of $\mathcal{IS}_1$. If all the steps of σ are correct predictions, then σ is a behavior of $\mathcal{IS}_1$. We can therefore define the required refinement mapping the same way we did for the prophecy constant σ. If σ contains an incorrect prediction, then $\mathcal{IS}_1^{hjp}$ halts (stutters forever) at the first incorrect prediction. Since $\mathcal{IS}_1$ is assumed not to halt, this means σ does not satisfy $\mathcal{IS}_1$ and it doesn't

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

[4]In TLA$^+$, it is easy to write a specification $\mathcal{IS}_1^h$ in which P is a collection that is "too large" to be a set—for example, a version of the Afek et al. algorithm in which a memory value can be any set, Russell's paradox implying that the collection of all sets is not a set. We could remove the assumption that P is a set by generalizing prophecy variables and constants to predict one among an arbitrary collection of possibilities, but there is no practical reason to do so. This assumption is built into many formalisms, including the one used by ER.

matter how the refinement mapping is defined, since a proof that $\mathcal{S}_1$ implements $\mathcal{S}_2$ considers only behaviors satisfying $\mathcal{IS}_1$.

These completeness proofs are based on embedding behavioral reasoning in state-based reasoning by recording behaviors in auxiliary variables. This is the same idea used in the first completeness result for assertional reasoning about concurrent specifications [22]. Such a completeness result is important because it shows that there are no inherent limitations to a proof method. However, it does not tell us anything about how to use the method in practice. For example, prophecy constants and prophecy variables are used differently, even though their completeness proofs are similar. Trying to mimic the reasoning used in these completeness proofs would simply place a state-based veneer over a behavioral proof. It would defeat the purpose of refinement mappings, which is to extend the Floyd-Hoare state-based assertional approach to concurrent systems.

8 CONCLUSION

Refinement means implementing a higher-level description of a program or system with a lower-level one. We know of two general methods of verifying refinement: refinement mappings, which we have described, and reduction. Reduction means obtaining a coarser-grained description (one with fewer atomic actions) from a finer-grained one by combining multiple atomic actions into a single action [20]. Reduction can be performed in TLA using refinement mappings [7].

ER and later uses of prophecy variables are based on assertional reasoning, introduced by Floyd [9] and Hoare [12]. The fundamental principle underlying this reasoning is that whether a program will do the correct thing in the future depends not on what it did in the past (or what it will do in the future), but on a property of its current state. For properties like invariance (including partial correctness) and termination that can be described in terms of the program's state, auxiliary variables are not necessary. They are needed only for refinement, which relate two specifications that may have different sets of states.

Much of the recent work uses prophecy variables based on a paradigm in which a *havoc* statement nondeterministically assigns to the variable an arbitrary element of some set of values, and a subsequent *assume* statement aborts execution if the "wrong" value was chosen. If the set of values is finite, this is a special case of an ER prophecy variable. Because it is a restricted type of ER variable, it is sound for an infinite set of values even though it doesn't satisfy ER's finite internal nondeterminism condition.

This *assume/havoc* approach is much like our simple prophecy variable, and it too is easier to use than an ER prophecy variable because it involves thinking forward to the future rather than backward from the future. However, like an ER prophecy variable, it predicts the future value of a variable. Our prophecy variables predict future events (action executions). This allows us easily to make predictions about structured sets of events, such as sequences. The prophecy variables of Zhang et al. [26] can predict a finite sequence of future values of a variable, but unlike predictions made with our prophecy sequences, their predictions cannot be modified. And sequences are just one kind of structure that can be put on our predictions.

Some recent work [8, 23] uses prophecy variables to assist in model checking, verifying that something eventually happens by adding a variable that predicts how many steps it takes for it to happen. They are verifying properties of state machines. Other recent work on prophecy variables [15, 19, 25, 26] uses Hoare triples to reason about programming-language code. Some of this work is for proving reduction [25]. The rest is for proving refinement (also called abstraction).

Being based on a programming language limits the power of methods. Reduction and refinement mappings are restricted to conform to the program structure. For example, it seems impossible for them to prove the equivalence (each refining the other) of two formulations of a simple N-process producer-consumer algorithm—one using two processes and the other using N processes [16]. Because their expression languages have no way of describing the control state, these programming-language-based methods require history variables to prove even the simplest properties—for example, that the parallel composition of two atomic statements that each increment x by 1 is a program that increments x by 2. Moreover, they do not seem to be able to express, let alone verify, the rich variety of liveness conditions that arise in concurrency—for example, to distinguish between weakly and strongly fair semaphores.

We do not mean to denigrate programming languages and verification of programs written in them. Programs are what are executed on computers, and it is important to verify them. TLA is for describing and reasoning about algorithms, which are above the code level. Hoare famously said: "Inside every large program is a small program that is struggling to get out." That small program is what we call an algorithm [14]. TLA is for specifying and reasoning about those algorithms. TLA is simple because it is very close to its semantics. It is extremely expressive because it uses the full power of mathematics to describe an algorithm. The algorithm and the properties we prove about it are written in mathematics, the same language in which the proofs are written. There is no need for a special verification logic such as Hoare logic or separation logic. TLA is mathematics.

REFERENCES

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

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

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

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

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

[6] Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2015. Aspect-oriented linearizability proofs. *Logical Methods in Computer Science* 11, 1 (2015), 1–33.

[7] Ernie Cohen and Leslie Lamport. 1998. Reduction in TLA. In *Concurrency Theory (CONCUR'98) (Lecture Notes in Computer Science)*, David Sangiorgi and Robert de Simone (Eds.), Vol. 1466. Springer-Verlag, 317–331.

[8] Byron Cook and Eric Koskinen. 2011. Making prophecies with decision predicates. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 399–410.

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

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

[11] Wim H. Hesselink. 2005. Eternity variables to prove simulation of specifications. *ACM Transactions on Computational Logic* 6, 1 (2005), 175–201.

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

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

[14] C. A. R. Hoare. 2019. (2019). Personal communication.

[15] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: Prophecy variables in separation logic. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 45:1–45:32.

[16] Leslie Lamport. 1997. Processes are in the eye of the beholder. *Theoretical Computer Science* 179 (1997), 333–351.

[17] Leslie Lamport. 2003. *Specifying Systems*. Addison-Wesley, Boston. Also available at http://lamport.org.

[18] Leslie Lamport and Stephan Merz. 2017. 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.

[19] Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 459–470.

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

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

[22] Susan Speer Owicki. 1975. *Axiomatic Proof Techniques for Parallel Programs*. Ph.D. Dissertation. Cornell University.

[23] Oded Padon, Jochen Hoenicke, Kenneth L. McMillan, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2018. Temporal prophecy for proving temporal properties of infinite-state systems. In *2018 Formal Methods in Computer Aided Design (FMCAD'18)*, Nikolaj Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–11.

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

[25] Shaz Qadeer, Ali Sezgin, and Serdar Tasiran. 2009. *Back and Forth: Prophecy Variables for Static Verification of Concurrent Programs*. Technical Report MSR-TR-2009-142. Microsoft.

[26] Zipeng Zhang, Xinyu Feng, Ming Fu, Zhong Shao, and Yong Li. 2012. A structural approach to prophecy variables. In *Theory and Applications of Models of Computation - 9th Annual Conference (TAMC'12), Proceedings (Lecture Notes in Computer Science)*, Manindra Agrawal, S. Barry Cooper, and Angsheng Li (Eds.), Vol. 7287. Springer, 61–71.

Received July 2020; revised June 2021; accepted October 2021