The Prophecy of Undo

Martín Abadi*

University of California, Santa Cruz, USA

Abstract. Prophecy variables are auxiliary program variables whose values are defined in terms of current program state and future behavior. This paper explains their relevance to reasoning about systems with "undo" operations, and develops an approach that facilitates their use.

1 Introduction

Auxiliary variables are often helpful in reasoning about systems and in proving their correctness. The most common auxiliary variables are history variables, whose values are defined in terms of current system state and past behavior. On the other hand, the definitions of prophecy variables, which can be seen as dual to history variables, refer to the future behavior of a system rather than to its past behavior [1]. Prophecy variables are considerably less mundane and well-understood than history variables. Although they may seem counter-intuitive, prophecy variables are sometimes necessary for the completeness of reasoning methods. They have not been employed frequently in reasoning about actual systems, but their uses (for example, linearizability proofs [6,5]) have been compelling and significant. This paper aims to contribute to their study and to their practical application.

Specifically, this paper highlights a class of important applications for prophecy variables, namely reasoning about systems with "undo" operations. These operations play a variety of roles, such as rolling back the effects of aborted transactions, restoring from snapshots in case of a failure, and recovering from attacks (e.g., [3,4,8]). In all these cases, the correctness of "undo" is both delicate and critical. In particular, although an "undo" may be selective (for example, in security, applying only to the effects of an attack), the "undo" should not give rise to inconsistencies in a system's observable behavior. When one system component rolls back a non-deterministic computation, other components should generally roll back any of their own actions influenced by the relevant non-deterministic choices; those choices may later be revisited, leading to different results. Prophecy variables can help in reasoning about the state of the system, basically because they can predict which computations will be rolled back.

Work on these applications leads us to the development of proof strategies that facilitate the use of prophecy variables. In the presence of liveness properties, the soundness of prophecy variables generally requires a certain finiteness condition

* Most of this work was done while M. Abadi was at Microsoft Research. He is now at Google.

A. Egyed and I. Schaefer (Eds.): FASE 2015, LNCS 9033, pp. 347–361, 2015.
DOI: 10.1007/978-3-662-46675-9_23

(roughly, that for each state there are only finitely many possible prophecies). For instance, when a variable represents an internal decision on how long a system will run, the condition helps ensure that this decision cannot be "undone" infinitely often and that the system cannot run forever. Even when this finiteness condition can be satisfied, it complicates the definition of prophecy variables. We devise an approach that allows us to ignore the condition during the bulk of the reasoning, and then ensure finiteness via a separate quotient construction. (To date, the experience with this approach is all related to "undo"; but the approach is more general, at least in theory.)

The paper focuses on two small, instructive examples. In both cases, we prove that a lower-level system with "undo" is a correct refinement of a higher-level system without "undo". The same ideas and techniques apply to reasoning about larger systems, and indeed the origin of this paper is ongoing research on a practical dataflow computing platform named Naiad [12]. The small examples of this paper have the advantage of not including the peculiarities of Naiad and other intricate features not directly relevant to our present purposes.

The literature contains a few other, somewhat related examples. In particular, Lampson has observed the relevance of prophecy variables to refinements proofs for protocols for reliable message delivery despite crashes [10]. His writings, however, do not include a precise definition of the required prophecy variables. Lampson's ideas are, to our knowledge, the most closely related to the present paper. Other work deals with the use of prophecy variables in proofs of program properties, rather than in refinement proofs, sometimes with the support of automated tools (e.g., [2]), which we do not consider in this paper. Thus, Sezgin, Tasiran, and Qadeer have developed a method for static verification of concurrent programs that includes prophecy variables [13]; they employed prophecy variables for expressing the consequences of interference between threads.

The literature also contains variations on the notion of prophecy variable and related concepts, such as backwards simulation relations [11]. In particular, Jonsson relaxed the finiteness requirement so that it needs to apply only infinitely often [7]. (He also provided an explanation of the connections between extant proof techniques, to which we refer the reader for additional background.) Those variations may well be helpful in reasoning about some systems with "undo" operations, complementing the present work.

The next section is a review of our framework for specifications and refinement proofs. Section 3 presents the first example. Section 4 introduces an approach to finiteness. Section 5 presents the second example, which leverages the results of Section 4. Section 6 concludes. Because of space constraints, we give the proofs for examples but omit those for the metatheory.

2 Specifications and Refinement Proofs (Review)

This section reviews our framework for specifications and the basic machinery of refinement mappings and prophecy variables (mostly from [1]). It contains no technical novelties.

2.1 State Spaces, Behaviors, and Properties

We assume a fixed set Σ_E of externally visible states. A *state space* Σ is a subset of $\Sigma_E \times \Sigma_I$ for some set Σ_I of internal states, or the set Σ_E itself. We let Π_E be the obvious projection mapping from $\Sigma_E \times \Sigma_I$ onto Σ_E.

The sequence $\langle\!\langle s_0, s_1, s_2, \ldots \rangle\!\rangle$ is said to be *stutter-free* if, for each i, either $s_i \neq s_{i+1}$ or the sequence is infinite and $s_i = s_j$ for all $j \geq i$. We define $\natural\sigma$ as the stutter-free sequence obtained from σ by replacing every maximal finite subsequence $s_i, s_{i+1}, \ldots, s_j$ of identical elements with the single element s_i. When S is a set of sequences, we let $\Gamma(S) = \{\tau : \exists \sigma \in S. \natural\sigma = \natural\tau\}$, and we say that S is *closed under stuttering* if $S = \Gamma(S)$.

For any set Σ, we write Σ^ω for the set of all infinite sequences of elements in Σ. We write $\sigma|_m$ for the prefix of σ of length m. We say that an infinite sequence $\langle\!\langle \sigma_0, \sigma_1, \sigma_2, \ldots \rangle\!\rangle$ of sequences in Σ^ω *converges* to the sequence σ in Σ^ω if for all $m \geq 0$ there exists $n \geq 0$ such that $\sigma_i|_m = \sigma|_m$ for all $i \geq n$. In this case, we define $\lim \sigma_i = \sigma$. We say that σ is a *limit point* of a set S if there exist elements σ_i in S such that $\lim \sigma_i = \sigma$. (While every element of S is trivially a limit point of S, in general S may have additional limit points.)

If Σ is a state space, then a *Σ-behavior* is an element of Σ^ω. A Σ_E-behavior is called an *externally visible behavior*. A *Σ-property* is a set of Σ-behaviors that is closed under stuttering. If the property contains all its limit points, then it is called a *safety property*. A Σ_E-property is called an *externally visible property*. If P is a Σ-property, then $\Pi_E(P)$ is a set of externally visible behaviors but not necessarily an externally visible property because it need not be closed under stuttering. The externally visible property *induced* by a Σ-property P is the set $\Gamma(\Pi_E(P))$. When Σ is clear from context or is irrelevant, we may use the terms *behavior* and *property* instead of Σ-behavior and Σ-property. We sometimes apply the adjective "complete", as in "complete behavior", to distinguish behaviors and properties from externally visible behaviors and properties.

2.2 State Machines

A *state machine* is a triple (Σ, F, N) where

- Σ is a state space,
- F, the set of *initial* states, is a subset of Σ,
- N, the *next-state relation*, is a subset of $\Sigma \times \Sigma$.

The *(complete) property generated by* a state machine (Σ, F, N) consists of all infinite sequences $\langle\!\langle s_0, s_1, \ldots \rangle\!\rangle$ such that $s_0 \in F$ and, for all $i \geq 0$, either $\langle s_i, s_{i+1} \rangle \in N$ or $s_i = s_{i+1}$. (We use angle brackets for elements of N, as in $\langle s_i, s_{i+1} \rangle$.) This set is closed under stuttering, so it is a Σ-property. It is also a safety property. The *externally visible property generated by* a state machine is the externally visible property induced by its complete property. In general, this externally visible property need not be a safety property.

For simplicity, we do not consider fairness conditions or other explicit liveness properties that can be imposed on state machines. Even without them, the

externally visible properties generated by state machines can imply non-trivial liveness properties (see [1, Section 3] and the example of Section 5), so the finiteness condition on prophecy variables cannot be ignored.

We loosely follow the TLA approach for specifying state machines [9]. Thus, we write specifications as logical formulas, of the form:

$$\exists y_1, \ldots, y_n.\ F \wedge \Box[N]_v$$

where:

- the state is represented by a set of state functions, which we write as variables $x_1, \ldots, x_m, y_1, \ldots, y_n$;
- we distinguish external variables and internal variables, and the internal variables (in this case, $y_1, \ldots, y_n$) are existentially quantified;
- F is a formula that may refer to the variables;
- $\Box$ is the temporal-logic operator "always";
- N is a formula that may refer to the variables and also to primed versions of the variables (which denote the values of those variables in the next state);
- v is a list of variables $v_1, \ldots, v_k$, and $[N]_v$ abbreviates $N \vee ((v'_1 = v_1) \wedge \ldots \wedge (v'_k = v_k))$.

The role of the subscript v is to guarantee closure under stuttering.

Such formulas do not describe state spaces, which we define separately.

2.3 Implementations and Refinement Mappings

A state machine $\mathbf{S}$ *implements* a state machine $\mathbf{S}'$ if and only if the externally visible property induced by $\mathbf{S}$ is a subset of the externally visible property induced by $\mathbf{S}'$. In other words, $\mathbf{S}$ implements $\mathbf{S}'$ when every externally visible behavior allowed by $\mathbf{S}$ is also allowed by $\mathbf{S}'$.

A *refinement mapping* from a state machine $\mathbf{S} = (\Sigma, F, N)$ to a state machine $\mathbf{S}' = (\Sigma', F', N')$ is a mapping $f : \Sigma \to \Sigma'$ such that:

R1. For all $s \in \Sigma$, $\Pi_E(f(s)) = \Pi_E(s)$.
R2. $f(F) \subseteq F'$.
R3. If $\langle s, t \rangle \in N$ then $\langle f(s), f(t) \rangle \in N'$ or $f(s) = f(t)$.

The following is a straightforward specialization of the soundness theorem for refinement mappings [1]:

Proposition 1. *If there exists a refinement mapping from* $\mathbf{S}$ *to* $\mathbf{S}'$*, then* $\mathbf{S}$ *implements* $\mathbf{S}'$*.*

Thus, refinement mappings, which work at the level of states, offer a convenient method for proving containment between sets of sequences of states.

2.4 Very Simple Prophecy Variables

Sometimes one needs to prove invariants and add auxiliary variables before constructing refinement mappings. As indicated in the Introduction, the most common auxiliary variables are history variables; their values are defined in terms of current system state and past behavior. For instance, for a program with an integer variable x, a history variable h may record the largest value of x seen so far; formally, h would be defined by the initial condition $h = x$ and the transition relation $h' = \mathtt{max}(h, x')$, which says that at every step the new value of h is the maximum of the previous value of h and the new value of x. Conversely, a prophecy variable might be defined by $h = \mathtt{max}(h', x)$, going from the future to the past, without an initial condition. The exact requirements on prophecy variables are, unfortunately, somewhat more intricate.

Formally, our starting point will be the notion of *simple* prophecy variables [1]. (The difference with "full-scale" prophecy variables has to do with stuttering.) Moreover, since we focus on refinement but not equivalences, and since our state machines do not include fairness conditions or other explicit liveness properties, we can omit requirements named P3 and P5 in [1]. We say that a state machine $\mathbf{S^P} = (\Sigma^P, F^P, N^P)$ is *obtained from* $\mathbf{S} = (\Sigma, F, N)$ *by adding a very simple prophecy variable* when the following requirements are satisfied:

P1. $\Sigma^P \subseteq \Sigma \times \Sigma_P$ for some set Σ_P.
P2′. $F^P = \{(s, p) \in \Sigma^P \mid s \in F\}$.
P4′. If $\langle s, s' \rangle \in N$ and $(s', p') \in \Sigma^P$
then there exists $(s, p) \in \Sigma^P$ such that $\langle (s, p), (s', p') \rangle \in N^P$.
P6. For all $s \in \Sigma$, the set $\{(s, p) \in \Sigma^P\}$ is finite and nonempty.

The following is a special case of the soundness theorem for prophecy variables [1]:

Proposition 2. *If* $\mathbf{S^P}$ *is obtained from* $\mathbf{S}$ *by adding a very simple prophecy variable, then* $\mathbf{S}$ *implements* $\mathbf{S^P}$.

3 First Example

Our first example is rather minimal. Its purpose is to illustrate why and how prophecy variables are useful for proving that a lower-level system with "undo" refines a higher-level system without "undo". Non-deterministic choice is prominent in this example and in the second one, because non-determinism typically complicates "undo", as suggested in the Introduction. More technically, the example also demonstrates how the definition of a state space with a prophecy variable can play a role similar to that of an invariant.

3.1 High-Level Specification (No "undo")

In our high-level system, an integer is chosen internally and then revealed, once. The internal choice is made by assigning a value to the internal variable y.

The publication of that choice consists in copying that value to the external variable x. Both variables have the value `ready` before those assignments.

The state space Σ_{High} is thus $(\mathbb{Z} \cup \{\texttt{ready}\}) \times (\mathbb{Z} \cup \{\texttt{ready}\})$.

Initial condition:

$$\mathit{InitProp} \triangleq (x = y = \texttt{ready})$$

Steps:

1. Choosing an integer:

$$\mathit{Choose} \triangleq ((x = x' = y = \texttt{ready}) \wedge (y' \in \mathbb{Z}))$$

2. Publishing the choice:

$$\mathit{Publish} \triangleq ((x = \texttt{ready}) \wedge (y \in \mathbb{Z}) \wedge (x' = y) \wedge (y' = y))$$

The high-level specification:

$$\mathit{SpecH} \triangleq \exists y.(\mathit{InitProp} \wedge \Box[\mathit{Choose} \vee \mathit{Publish}]_{x,y})$$

3.2 Low-Level Specification (with "undo")

In the low-level specification, additional transitions can undo choices. The specification is silent on why rollbacks happen—whether because of failures or as deliberate steps. Accordingly, for simplicity, we do not model failure detection, nor do we distinguish a "redo" from an original choice.

The state space Σ_{Low} equals Σ_{High}; moreover, the initial condition and the actions for choosing and for publishing a choice are exactly as above in the high-level specification. We have one additional action for undoing a choice:

$$\mathit{Undo} \triangleq ((x = x' = \texttt{ready}) \wedge (y \in \mathbb{Z}) \wedge (y' = \texttt{ready}))$$

The low-level specification:

$$\mathit{SpecL} \triangleq \exists y.(\mathit{InitProp} \wedge \Box[\mathit{Choose} \vee \mathit{Undo} \vee \mathit{Publish}]_{x,y})$$

3.3 Prophecy Variable

There is no direct refinement mapping from the low-level specification to the high-level specification, basically because there is no way to know whether to map a low-level state $(\texttt{ready}, n)$ to $(\texttt{ready}, n)$ or to $(\texttt{ready}, \texttt{ready})$ without predicting whether the choice of n will persist. To be conservative, we could always map $(\texttt{ready}, n)$ to $(\texttt{ready}, \texttt{ready})$, but then a low-level transition that publishes n would need to be expanded into two high-level transitions, something that plain refinement mappings do not support. In more realistic variants of this example, the number and complexity of the additional high-level transitions may be larger; in the example of Section 5, the conservative approach is not viable at all.

On the other hand, we can find a suitable refinement mapping after introducing a prophecy variable, as we show next. This prophecy variable is an internal variable that we add by defining an enriched state space:

Construction 1. *The state space* Σ^P_{Low} *consists of tuples* (x, y, D) *where*

1. $(x, y) \in \Sigma_{\text{Low}}$,
2. $D \in \{\texttt{done}, \texttt{notdone}\}$,
3. $x \in \mathbb{Z}$ *implies* $D = \texttt{done}$.

Intuitively, `done` indicates that a choice is final; `notdone` that it is not.

Initial condition:

$$\mathit{InitPropP} \triangleq \mathit{InitProp}$$

Steps:

1. Choosing an integer:

$$\mathit{ChooseP} \triangleq (\mathit{Choose} \wedge (D = D'))$$

2. Undoing the choice:

$$\mathit{UndoP} \triangleq (\mathit{Undo} \wedge (D = \texttt{notdone}))$$

3. Publishing the final choice:

$$\mathit{PublishP} \triangleq (\mathit{Publish} \wedge (D = D'))$$

The enriched low-level specification:

$$\mathit{SpecP} \triangleq \exists y, D.(\mathit{InitPropP} \wedge \Box[\mathit{ChooseP} \vee \mathit{UndoP} \vee \mathit{PublishP}]_{x,y,D})$$

Note the absence of an initial condition on D, and that the value of D affects the enabledness of the action *UndoP*. These features clearly indicate that D is not an ordinary history variable. In other respects, D is quite tame: trivial equations such as $D = D'$ and $D = \texttt{notdone}$ are easy to handle in reasoning with these definitions.

Proposition 3. *SpecP is obtained from SpecL by adding a very simple prophecy variable.*

Proof: The conditions on the state space (P1), on initial states (P2′), and on existence and finiteness (P6) are immediate.

The condition on backward steps (P4′) is the only non-trivial one, but it is still easy. Since $\mathit{Choose} \vee \mathit{Undo} \vee \mathit{Publish}$ implies $x \notin \mathbb{Z}$, the choice of D is unconstrained by condition (3) in Construction 1. So, for backward steps, taking whatever value of D the corresponding action suggests is appropriate ($D = D'$ or $D = \texttt{notdone}$, depending on whether the transition corresponds to *Choose* or *Publish* or to *Undo*). □

3.4 Refinement Mapping

Next we construct a refinement mapping from the low-level specification with a prophecy variable to the high-level specification:

Proposition 4. *Let* $f : \Sigma^P_{\mathrm{Low}} \to \Sigma_{\mathrm{High}}$ *be such that:*

$$f(x, y, \texttt{done}) = (x, y)$$
$$f(x, y, \texttt{notdone}) = (\texttt{ready}, \texttt{ready})$$

Then f *is a refinement mapping from SpecP to SpecH.*

Proof:
- The constraint that $x \in \mathbb{Z}$ implies $D = \texttt{done}$ (condition (3) in Construction 1) entails that $f(x, y, \texttt{notdone}) = (\texttt{ready}, \texttt{ready})$ preserves the value of the external variable x.
- Initial states are mapped to initial states, trivially.
- *ChooseP* transitions are mapped to stutters if $D = \texttt{notdone}$ and to *Choose* transitions otherwise: in a *ChooseP* transition, $(x, y, D) = (\texttt{ready}, \texttt{ready}, D)$ and $(x', y', D') = (\texttt{ready}, y', D')$ with $y' \in \mathbb{Z}$; hence $f(x, y, D) = (\texttt{ready}, \texttt{ready})$ and $f(x', y', D') = (\texttt{ready}, \texttt{ready})$ if $D = \texttt{notdone}$ and $= (\texttt{ready}, y')$ otherwise.
- *UndoP* transitions are mapped to stutters: in a *UndoP* transition, $(x, y, D) = (\texttt{ready}, y, \texttt{notdone})$ and $(x', y', D') = (\texttt{ready}, \texttt{ready}, D')$; hence $f(x, y, D) = (\texttt{ready}, \texttt{ready})$ and $f(x', y', D') = (\texttt{ready}, \texttt{ready})$.
- *PublishP* transitions are mapped to *Publish* transitions: in a *PublishP* transition $(x, y, D) = (\texttt{ready}, y, D)$ where $y \in \mathbb{Z}$ and $(x', y', D') = (y, y, D)$; hence $x' \in \mathbb{Z}$, so $D' = D = \texttt{done}$ by condition (3) in Construction 1, so $f(x, y, D) = (\texttt{ready}, y)$ where $y \in \mathbb{Z}$ and $f(x', y', D') = (y, y)$.

□

3.5 Main Result

We conclude:

Proposition 5. *SpecL implements SpecH.*

Proof: By Propositions 3 and 2, *SpecL* implements *SpecP*. By Propositions 4 and 1, *SpecP* implements *SpecH*. The claim follows by transitivity. □

4 An Approach to Finiteness

In this section we present results that can facilitate proofs with prophecy variables. Although they do not constitute a panacea, they conveniently enable us to shift the finiteness requirement on prophecy variables through a quotient construction. We demonstrate their use in the example of Section 5, below. (They would also have been helpful in the example of Section 3 if we had foolishly allowed not only `done` and `notdone` but all strings as possible values of D.)

4.1 Quotients

When $\mathcal{Q}$ is an equivalence relation on Σ, the *quotient* of the state machine $\mathbf{S} = (\Sigma, F, N)$ by $\mathcal{Q}$ is the state machine $\mathbf{S}_{/\mathcal{Q}} = (\Sigma_{/\mathcal{Q}}, F_{/\mathcal{Q}}, N_{/\mathcal{Q}})$ such that:

Q1. $\Sigma_{/\mathcal{Q}} = \Sigma/\mathcal{Q}$ (the set of equivalence classes of states from Σ).
Q2. For $s \in \Sigma_{/\mathcal{Q}}$, $s \in F_{/\mathcal{Q}}$ if and only if there exists $s' \in s$ such that $s' \in F$.
Q3. For $s, t \in \Sigma_{/\mathcal{Q}}$, $\langle s, t\rangle \in N_{/\mathcal{Q}}$ if and only if there exist $s' \in s$ and $t' \in t$ such that $\langle s', t'\rangle \in N$.

We represent each equivalence class in $\Sigma/\mathcal{Q}$ by an arbitrary member (say, the smallest in a fixed ordering of Σ), so that $\Sigma/\mathcal{Q} \subseteq \Sigma \subseteq \Sigma_E \times \Sigma_I$ for some set Σ_I.

4.2 Refinement Mappings via Quotients

We say that a function f with domain Σ *respects* the equivalence relation $\mathcal{Q}$ on Σ when, for all $s, t \in \Sigma$, if $s\mathcal{Q}t$ then $f(s) = f(t)$.

Proposition 6. *Let $\mathbf{S} = (\Sigma, F, N)$ and $\mathbf{S}' = (\Sigma', F', N')$, and let $\mathbf{S}_{/\mathcal{Q}}$ be the quotient of $\mathbf{S}$ by an equivalence relation $\mathcal{Q}$ on Σ. Assume that:*

- *f is a refinement mapping from $\mathbf{S}$ to $\mathbf{S}'$,*
- *f respects $\mathcal{Q}$.*

Let $f_{/\mathcal{Q}} : \Sigma/\mathcal{Q} \to \Sigma'$ be such that, for all $s \in \Sigma/\mathcal{Q}$, $f_{/\mathcal{Q}}(s) = f(s)$. Then $f_{/\mathcal{Q}}$ is a refinement mapping from $\mathbf{S}_{/\mathcal{Q}}$ to $\mathbf{S}'$.

4.3 Very Simple Prophecy Variables via Quotients

We define a state machine $\mathbf{S^P} = (\Sigma^P, F^P, N^P)$ to be *obtained from* $\mathbf{S} = (\Sigma, F, N)$ *almost by adding a very simple prophecy* when the usual conditions P1, P2′, and P4′ hold, and instead of P6 we have only part of it:

P6′. For all $s \in \Sigma$, the set $\{(s, p) \in \Sigma^P\}$ is nonempty.

The following proposition enables us to recover a simple prophecy variable:

Proposition 7. *Let $\mathbf{S} = (\Sigma, F, N)$ and $\mathbf{S^P} = (\Sigma^P, F^P, N^P)$, and let $\mathbf{S^P_{/\mathcal{Q}}} = (\Sigma^P_{/\mathcal{Q}}, F^P_{/\mathcal{Q}}, N^P_{/\mathcal{Q}})$ be the quotient of $\mathbf{S^P}$ by the equivalence relation $\mathcal{Q}$ on Σ^P. Assume that:*

1. *$\mathbf{S^P}$ is obtained from $\mathbf{S}$ almost by adding a very simple prophecy variable,*
2. *for all $(s_1, p_1), (s_2, p_2) \in \Sigma^P$, if $(s_1, p_1)\mathcal{Q}(s_2, p_2)$ then $s_1 = s_2$ (in other words, projecting to the first component respects $\mathcal{Q}$),*
3. *for all $s \in \Sigma$, the set $\{(s, p) \in \Sigma^P\}/\mathcal{Q}$ is finite.*

Then $\mathbf{S^P_{/\mathcal{Q}}}$ is obtained from $\mathbf{S}$ by adding a very simple prophecy variable.

4.4 Invariants and Quotienting

Invariants are often needed before other arguments in proofs. In particular, the required conditions for refinements mappings (R1 and especially R3) sometimes hold only for reachable system states, and the role of invariants is to focus attention on those states. The propositions in this section enable us to combine invariants with quotienting.

We say that *Inv* is an *inductive invariant* of the state machine $\mathbf{S} = (\Sigma, F, N)$ if $Inv \subseteq \Sigma$, $F \subseteq Inv$, and, for all $s, t \in \Sigma$, if $\langle s, t \rangle \in N$ and $s \in Inv$ then $t \in Inv$. Given a subset *Inv* of Σ and an equivalence relation $\mathcal{Q}$ on Σ, we write $Inv_{/\mathcal{Q}}$ for the subset of $\Sigma_{/\mathcal{Q}}$ such that $s \in Inv_{/\mathcal{Q}}$ if and only if there exists $s' \in s$ such that $s' \in Inv$. (This notation generalizes the definition of $F_{/\mathcal{Q}}$.) We say that *Inv respects* $\mathcal{Q}$ when, for all $s, t \in \Sigma$, if $s \mathcal{Q} t$ and $s \in Inv$ then $t \in Inv$.

Proposition 8. *Assume that:*

- *Inv is an inductive invariant of* $\mathbf{S} = (\Sigma, F, N)$*,*
- *Inv respects the equivalence relation* $\mathcal{Q}$ *on* Σ*.*

Then $Inv_{/\mathcal{Q}}$ *is an inductive invariant of* $\mathbf{S}_{/\mathcal{Q}} = (\Sigma_{/\mathcal{Q}}, F_{/\mathcal{Q}}, N_{/\mathcal{Q}})$*.*

Given a specification $\mathbf{S} = (\Sigma, F, N)$ and a subset *Inv* of Σ, we write $\mathbf{S} + Inv$ for $(\Sigma, F, N \cap (Inv \times \Sigma))$.

Proposition 9. *Assume that Inv is an inductive invariant of* $\mathbf{S} = (\Sigma, F, N)$*. Then* $\mathbf{S}$ *and* $\mathbf{S} + Inv$ *generate the same complete property.*

Proposition 10. *Assume that Inv respects* $\mathcal{Q}$*. Then the complete property that* $\mathbf{S}_{/\mathcal{Q}} + Inv_{/\mathcal{Q}}$ *generates is included in that of* $(\mathbf{S} + Inv)_{/\mathcal{Q}}$*.*

5 Second Example

Our second example is slightly longer and much trickier than the first. It illustrates how the finiteness requirement on prophecy variables can be conveniently ignored in the core of a refinement argument.

5.1 High-Level Specification (No "undo")

In the high-level system, integers are chosen internally in ascending order, and then revealed, gradually. The choice is made by adding elements to an internal variable y that holds a finite set. The publication of a choice consists in moving the largest element of y to the external variable x, which also holds a finite set.

The state space Σ_{High} is thus $\mathcal{P}_f(\mathbb{Z}) \times \mathcal{P}_f(\mathbb{Z})$ where $\mathcal{P}_f(\mathbb{Z})$ is the set of finite sets of integers.

Initial condition:

$$InitProp \triangleq (x = y = \emptyset)$$

Steps:

1. Choosing one more integer:

$$Choose \triangleq \exists n \in \mathbb{Z}.((n > \max(x \cup y)) \wedge (y' = y \cup \{n\}) \wedge (x' = x))$$

2. Publishing the largest pending choice:

$$Publish \triangleq \exists n \in \mathbb{Z}.((n = \max(y)) \wedge (y' = y - \{n\}) \wedge (x' = x \cup \{n\}))$$

The high-level specification:

$$SpecH \triangleq \exists y.(InitProp \wedge \Box[Choose \vee Publish]_{x,y})$$

Note that *SpecH* allows all behaviors where x takes a sequence of values $\emptyset$, $\{0\}$, $\{0, -1\}$, ..., $\{0, -1, \ldots, -k\}$, but not their limit, so it is not a safety property.

5.2 Low-Level Specification (with "undo")

In the low-level specification, additional transitions can undo choices. Again, the state space Σ_{Low} equals Σ_{High}; moreover, the initial condition and the actions for choosing and for publishing a choice are exactly as above in the high-level specification. We have one additional action for undoing choices:

$$Undo \triangleq \exists S \subseteq \mathbb{Z}.((y' = y \cap S) \wedge (x' = x))$$

This action models a selective rollback, in which we keep choices only if they are in a given set S of "survivors".

The low-level specification:

$$SpecL \triangleq \exists y.(InitProp \wedge \Box[Choose \vee Undo \vee Publish]_{x,y})$$

5.3 Prophecy Variable

Again, we cannot directly find a refinement mapping from the low-level specification to the high-level specification. The constraints on the order in which integers are chosen and published contribute to this difficulty.

- For instance, suppose that we were to map the low-level state $(\{4\}, \{2, 3\})$ to the high-level state $(\{4\}, \{2, 3\})$, naively. This trivial mapping cannot possibly be satisfactory. According to the low-level specification, $(\{2, 4\}, \emptyset)$ is reachable from $(\{4\}, \{2, 3\})$ (via an "undo" of 3 and the publication of 2), while according to the high-level specification no state of the form $(\{2, 4\}, \cdot)$ is reachable from $(\{4\}, \{2, 3\})$.
- On the other hand, unlike in the first example (see Section 3.3), we cannot pretend that choices are not made "until the last minute". For instance, we cannot map the low-level state $(\{4\}, \{2, 3\})$ to the high-level state $(\{4\}, \emptyset)$. According to the low-level specification, $(\{2, 3, 4\}, \emptyset)$ is reachable from $(\{4\}, \{2, 3\})$ (by publishing 3 and then 2), while according to the high-level specification no state of the form $(\{2, 3, 4\}, \cdot)$ is reachable from $(\{4\}, \emptyset)$.

So we introduce a prophecy variable. We add the prophecy variable as an internal variable in an enriched state space:

Construction 2. *The state space* Σ^P_{Low} *consists of tuples* (x, y, D) *where*

1. $(x, y) \in \Sigma_{\mathrm{Low}}$,
2. $D \subseteq \mathbb{Z}$,
3. $x \subseteq D$.

Intuitively, choices of the integers in D will never be undone in the future. Accordingly, condition (3) says that choices that have been revealed cannot be undone.

Initial condition:

$$\mathit{InitPropP} \triangleq \mathit{InitProp}$$

Steps:

1. Choosing one more integer:

$$\mathit{ChooseP} \triangleq (\mathit{Choose} \wedge (D = D'))$$

2. Undoing some choices:

$$\mathit{UndoP} \triangleq \exists S \subseteq \mathbb{Z}.((y' = y \cap S) \wedge (x' = x) \wedge (D = (D' \cap S) \cup x))$$

3. Publishing a choice:

$$\mathit{PublishP} \triangleq (\mathit{Publish} \wedge (D = D'))$$

The enriched low-level specification:

$$\mathit{SpecP} \triangleq \exists y, D.(\mathit{InitPropP} \wedge \Box[\mathit{ChooseP} \vee \mathit{UndoP} \vee \mathit{PublishP}]_{x,y,D})$$

Note the equation $D = (D' \cap S) \cup x$ in *UndoP*, which defines D from D'. Such definitions are typical for prophecy variables. In this case, the equation might be read as saying that the choices of integers that will never be undone henceforth are the choices of integers already in x or the choices that "survive" this *UndoP* step and will never be undone afterwards.

Several variants are possible. In particular, we could change the state space so that D is included in $x \cup y$, thus ensuring that for each (x, y) there are only finitely many possible values for D. Accordingly, we would have to make other adjustments, not all of them attractive. For instance, in *ChooseP*, we would have to replace $D = D'$ with a more complicated equation, such as $D = D' \cap (x \cup y)$. Such expressions would then appear pervasively throughout proofs. In a larger example (like the one of interest to us in the context of Naiad), such a change can create more work than it avoids. So we proceed without a finiteness guarantee; quotienting will nevertheless allow us to complete the verification.

Proposition 11. *SpecP is obtained from SpecL almost by adding a very simple prophecy variable.*

Proof: The conditions on the state space (P1), on initial states (P2′), and on existence (P6′) are again immediate.

The condition on backward steps (P4′) is the only non-trivial one: it requires care in order to ensure that $x \subseteq D$. In the cases of transitions *Choose* and *Publish*, we take $D = D'$ as suggested by the definitions of *ChooseP* and *PublishP*, and the desired result follows since $x \subseteq x'$ in both cases. In the case of a transition *Undo*, we take $D = (D' \cap S) \cup x$ for a set S as suggested by the definition of *UndoP*, and immediately obtain that $x \subseteq D$. □

5.4 Refinement Mapping

We let *Inv* be the predicate $x \cap y = \emptyset$, which says that x and y are disjoint. It is a straightforward inductive invariant of *SpecH*, *SpecL*, and *SpecP*. Relying on *Inv*, we construct a refinement mapping:

Proposition 12. *Let* $f : \Sigma^P_{\text{Low}} \to \Sigma_{\text{High}}$ *be such that:*

$$f(x, y, D) = (x, y \cap D)$$

Then f is a refinement mapping from SpecP + Inv to SpecH.

Proof:
- The mapping trivially preserves the external variable x.
- It maps initial states to initial states.
- It maps *ChooseP* transitions to *Choose* transitions or to stutters, depending on whether the integer chosen is in D.
- It maps *PublishP* transitions to *Publish* transitions: if a *PublishP* transition adds n to x, then $n \in y$ and $n \in x'$, so $n \in D'$ by condition (3) in Construction 2, so $n \in D$ since $D = D'$, so $n \in y \cap D$.
- It maps *UndoP* transitions to stutters: for some $S \subseteq \mathbb{Z}$, we have $y' \cap D' = (y \cap S) \cap D' = y \cap (D' \cap S) = y \cap ((D' \cap S) \cup x) = y \cap D$ since $D = ((D' \cap S) \cup x)$. The equation $y \cap (D' \cap S) = y \cap ((D' \cap S) \cup x)$ exploits the invariant *Inv*.

□

5.5 Main Result (via Quotienting)

At this point, we have the main ingredients of a proof that *SpecL* implements *SpecH*: an auxiliary variable that (almost) satisfies all expected conditions, an invariant, and a refinement mapping. It remains to put them together. The recipe for this purpose has several steps but is easy and fairly generic. The steps are much as in the first example (Proposition 5), but with an extra layer of quotienting. Crucially, the statement of the final result (Proposition 13) does not mention inductive invariants or quotients.

Proposition 13. *SpecL implements SpecH.*

Proof: We quotient Σ^P_{Low} by a relation $\mathcal{Q}$:

$$(x_1, y_1, D_1)\mathcal{Q}(x_2, y_2, D_2)$$
$$\text{if and only if}$$
$$(x_1, y_1) = (x_2, y_2) \text{ and } y_1 \cap D_1 = y_2 \cap D_2$$

We prove that *SpecL* implements $SpecP_{/\mathcal{Q}}$ and that $SpecP_{/\mathcal{Q}}$ implements *SpecH*. The claim follows by transitivity.

1. For each x and y, the set of equivalence classes $\{(x, y, D) \in \Sigma^P_{\mathrm{Low}}\}/\mathcal{Q}$ is finite, because y is finite. Therefore, by Propositions 11 and 7, $SpecP_{/\mathcal{Q}}$ is obtained from *SpecL* by adding a very simple prophecy variable. By Proposition 2, *SpecL* implements $SpecP_{/\mathcal{Q}}$.
2. Both *Inv* and f respect the equivalence relation $\mathcal{Q}$: the definition of *Inv* does not even mention D, and that of f uses it only in a context of the form $y \cap D$. Since *Inv* is an inductive invariant of *SpecP*, $Inv_{/\mathcal{Q}}$ is an inductive invariant of $SpecP_{/\mathcal{Q}}$ by Proposition 8. Therefore, $SpecP_{/\mathcal{Q}}$ implements $SpecP_{/\mathcal{Q}} + Inv_{/\mathcal{Q}}$ by Proposition 9. In turn, $SpecP_{/\mathcal{Q}} + Inv_{/\mathcal{Q}}$ implements $(SpecP + Inv)_{/\mathcal{Q}}$ by Proposition 10. Finally, $(SpecP + Inv)_{/\mathcal{Q}}$ implements *SpecH* by Proposition 12 (which constructs a refinement mapping), Proposition 6 (which quotients the mapping), and Proposition 1 (which asserts the soundness of refinement mappings). We deduce that $SpecP_{/\mathcal{Q}}$ implements *SpecH* by transitivity.

□

6 Conclusion

Prophecy variables are generally useful for manipulating the timing of non-deterministic choices. Specifically, they help in constructing refinement mappings when a low-level specification makes its non-deterministic choices later than a corresponding high-level specification. Therefore, prophecy variables are broadly relevant to reasoning about systems with "undo" operations: those operations can roll back computations that include non-deterministic choices, which are effectively not made until they are committed, perhaps by a later output action.

The examples of this paper focus on the delicate interaction between "undo" operations and non-determinism. Of course, actual systems have many other aspects, including non-trivial deterministic computations and sophisticated methods for rollback based on logging and checkpointing (e.g., [3]). Nevertheless, reasoning about those systems can benefit from prophecy variables and from the techniques described in this paper. Our ongoing work on Naiad, for instance, features non-deterministic scheduling choices; "undo" is typically a rollback caused by a failure. Prophecy variables are crucial in the proof that a low-level specification with failures refines a high-level specification where such failures are assumed impossible. As in this paper, prophecy variables predict which choices will persist and which will be revisited.

Acknowledgments. I am grateful to Michael Isard, Leslie Lamport, and Butler Lampson for discussions on the subject of this paper.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 82(2), 253–284 (1991)
2. Cook, B., Koskinen, E.: Making prophecies with decision predicates. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 399–410 (2011)
3. Elnozahy, E.N., Alvisi, L., Wang, Y., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys 34(3), 375–408 (2002)
4. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory, 2nd edn. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers (2010)
5. Henzinger, T.A., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013 – Concurrency Theory. LNCS, vol. 8052, pp. 242–256. Springer, Heidelberg (2013)
6. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
7. Jonsson, B.: Simulations between specifications of distributed systems. In: Groote, J.F., Baeten, J.C.M. (eds.) CONCUR 1991. LNCS, vol. 527, pp. 346–360. Springer, Heidelberg (1991)
8. Kim, T., Wang, X., Zeldovich, N., Kaashoek, M.F.: Intrusion recovery using selective re-execution. In: 9th USENIX Symposium on Operating Systems Design and Implementation, pp. 89–104 (2010)
9. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
10. Lampson, B.W.: Reliable messages and connection establishment. In: Mullender, S. (ed.) Distributed Systems, pp. 251–281. Addison-Wesley (1993)
11. Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations: I. Untimed systems. Information and Computation 121(2), 214–233 (1995)
12. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: ACM SIGOPS 24th Symposium on Operating Systems Principles. pp. 439–455 (2013)
13. Sezgin, A., Tasiran, S., Qadeer, S.: Tressa: Claiming the future. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 25–39. Springer, Heidelberg (2010)