Using Ghost Variables to Prove Refinement*

M. Marcus and A. Pnueli

Department of Computer Science,
Weizmann Institute of Science, Rehovot 76100, Israel.
e-mail: monica@wisdom.weizmann.ac.il

Abstract. We propose a method for proving refinement between programs, based on augmenting the program by ghost (auxiliary) variables and statements that assign values to these variables. We show that, in many cases, this augmentation can replace the need for an explicit refinement mapping from the variables of one system to the private variables of the other system. A novel feature of the proposed methodology is that the expressions assigned to the ghost variables may depend on the future. This may replace the need for prophecy variables that are defined by a decreasing induction and require some version of finite-image hypothesis. We believe that proving refinement by program augmentation leads to a more natural style of refinement and is easier to design and comprehend. Allowing future-dependent expressions in a program requires extensions to the temporal methodology for proving properties of programs. These extensions are explained and discussed in the paper.

1 Introduction

A rigorous development of a complex reactive system often proceeds through several intermediate stages on the way from a top-level abstract specification to a final efficient running version.

$$S_0 \sqsupseteq S_1 \sqsupseteq \cdots S_{k-1} \sqsupseteq S_k$$

In this development sequence, each system S_{i+1} *refines* its predecessor S_i (using the symbol $\sqsupseteq$ to denote the relation of refinement). In our linear semantics framework, refinement means that the set of behaviors allowed by S_{i+1} is a subset of the behaviors allowed by S_i. Behaviors of the two systems are compared by projecting the detailed computations of each of them on a set of *observable variables* which are common to both systems.

Many formalisms have been proposed for proving refinement between systems. We refer the reader to the collection [dBdRR90] for a comprehensive coverage of the main approaches to refinement proofs. The paper [AL91] presents a semantic framework for the study of refinement and proves, under some assumptions, completeness of a proof method based on refinement mapping. The note

* This research was supported in part by a basic research grant from the Israeli Academy of Sciences, and by the European Community ESPRIT Basic Research Action Project 6021 (REACT).

[Lam92] shows how these semantic ideas can be worked out within the TLA framework. In [KMP94], we showed how to prove simulation and refinement within the framework of the temporal logic of [MP91b], using the proof systems presented in [MP91a] and [MP91b].

Let $S^{\mathcal{A}}$ and $S^{\mathcal{C}}$ denote an abstract and a concrete system, and consider the problem of proving that system $S^{\mathcal{C}}$ refines system $S^{\mathcal{A}}$, which we can write as

$$S^{\mathcal{C}} \sqsubseteq S^{\mathcal{A}}.$$

One of the main problems in establishing such a refinement relation is in the case that $S^{\mathcal{A}}$ contains system variables that are not present in $S^{\mathcal{C}}$. To show that every observation of $S^{\mathcal{C}}$ corresponds to some observation of $S^{\mathcal{A}}$, it is necessary to identify the values assumed by the private $S^{\mathcal{A}}$ variables in such an observation. This requires the ability to map $S^{\mathcal{C}}$-states onto $S^{\mathcal{A}}$-states. We refer to such a mapping as a *refinement mapping* and its construction has been identified as one of the central tasks in proving refinements between systems.

In the simplest case the necessary mapping is a simple state-to-state transformation. However, as pointed out in [AL91], this may be impossible in some cases, and the mapping may depend on the prefix of the observation up to the present (which calls for the use of *history variables* in the terminology of [AL91]) and, in even more complicated cases, may also depend on the infinite suffix of the observation from the mapped state on (which calls for the use of *prophecy variables* in the terminology of [AL91]).

In [KMP94], we proposed a single proof rule for establishing refinement relation between two systems. This proof rule allowed the refinement mapping to be an arbitrary temporal function which may refer to the complete observation to determine the local concrete-to-abstract state mapping. This most general license covers the history and prophecy references as special cases. While, in principle, this single rule is adequate for proving all interesting refinement relations, it's overwhelming generality does not provide useful guidance to the user about how to make the first steps in establishing a refinement.

In this paper, we take an opposite approach. Instead of presenting a single monolithic rule that can do everything, we list a tool suite consisting of several smaller rules, each applicable in certain situations. It is fair to say that the proposed approach has a more algebraic proof style in which we establish $S^{\mathcal{C}} \sqsubseteq S^{\mathcal{A}}$ in a sequence of small steps of the form

$$S^{\mathcal{C}} = S_1 \sqsubseteq S_2 \sqsubseteq \cdots \sqsubseteq S_k = S^{\mathcal{A}},$$

where, typically, different rules may be applied in different steps.

An important contribution to the conceptual simplification of the refinement task is that, in many cases, we replaced the abstract notion of refinement mapping, which many practitioners find difficult to design correctly, by the more familiar concept of augmentation with auxiliary variables (e.g. [OG76]). It is not difficult to see how history variables, which are usually defined by an ascending inductive definition, can be viewed or implemented as auxiliary variables which are modified by assignments augmenting the program. What is less obvious is

that a similar thing can be done with prophecy variables which are defined by a descending inductive definition. To do this, we extended the programming language and allow statements which assign future-dependent temporal expressions to auxiliary variables. This requires some extension of the proof rules for establishing invariants over a program.

The general strategy for proving refinement by program augmentation can be simplistically described as follows. Starting with system $S^{\mathcal{C}}$, we add the private variables of $S^{\mathcal{A}}$ as auxiliary variables. This leads us to a combined system, let us call it S_2, in which assignments to the private variables of $S^{\mathcal{C}}$ and $S^{\mathcal{A}}$ coexist side-by-side but the observable variables are assigned values depending only on the variables of $S^{\mathcal{C}}$. Next, we prove some invariants for the combined system S_2 whose main role is to establish some relations between the private $S^{\mathcal{A}}$ variables and the $S^{\mathcal{C}}$ variables. As the next step, we use these relations to replace the assignments to the observable variables by assignments which refer only to $S^{\mathcal{A}}$ variables. This leads us to the next version of the system, say S_3. At this point, the private $S^{\mathcal{C}}$ variables no longer affect the observables and they may be considered as auxiliary variables, not essential to the observed behavior of the system. As the final step, we de-augment these variables from the system, reproducing system $S^{\mathcal{A}}$.

Auxiliary variables are used by Reynolds [Rey81] in the development of data representation structuring for sequential programs. Their role is to help proving invariants about the program, which are used in the process of replacing the abstract variables by concrete ones. In [Rey81] the definition of auxiliary variables is attributed to Owicki and Gries [OG76] but it is said that the basic concept can be found already in [Luc68].

The general idea of constructing a joint cross product system that runs $S^{\mathcal{C}}$ and $S^{\mathcal{A}}$ together, establishing some invariants for the combined system and using them to simplify or transform the system into a version of $S^{\mathcal{A}}$ was already suggested in [Jon87]. Our main extension of this method is the introduction of future-dependent auxiliary variables, and treating them as any other program variables.

The rest of the paper is organized as follows. In Section 2, we present the basic model of fair transition systems and show how distributed programs can be represented in this model. In Section 3 we define the refinement relation between fair transition systems and present a verification rule for refinement. In Section 4, two types of equivalence transformations of programs are defined, namely spacing congruence and safe augmentation. Section 5 shows how to modify the proof rules for invariance properties of programs such that future-dependent temporal expressions are allowed in assignment statements. In Section 6, the proposed proof method is illustrated by examples. The final Section 7 contains discussions and conclusions.

2 Basic Concepts and Definitions

We assume a universal set of typed variables $\mathcal{V}$, called *vocabulary*. A *state* s is a type-consistent interpretation of $\mathcal{V}$, assigning to each variable $u \in \mathcal{V}$ a value $s[u]$ over its domain. The set of all states is denoted by Σ.

The specification language used in this paper is that of *linear quantified temporal logic* whose syntax and semantics can be found in [MP91b]. Temporal logic is constructed out of state formulas (to which we refer as *assertions*) to which we apply boolean and temporal operators.

A *model* for a temporal formula is an infinite sequence of states

$$\sigma: \quad s_0, s_1, \ldots .$$

Model σ': $s'_0, s'_1, \ldots$ is said to be a *stuttering variant* of model σ: $s_0, s_1, \ldots$, if σ' can be obtained from σ by the duplication of some states and the deletion of some duplicate states.

The computational model for reactive systems is given by a *fair transition system* (FTS) $S = \langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ consisting of:

- $V = \{u_1, ..., u_n\}$: A finite set of *system variables*.
- Θ : The *initial condition*. A satisfiable assertion characterizing the initial states.
- $\mathcal{T}$: A finite set of *transitions*. Each transition $\tau \in \mathcal{T}$ is a function

  $$\tau : \Sigma \mapsto 2^{\Sigma},$$

  mapping each state $s \in \Sigma$ into a (possibly empty) set of τ-*successor* states $\tau(s) \subseteq \Sigma$.
  The function associated with a transition τ is represented by an assertion $\rho_\tau(V, V')$, called the *transition relation*, which relates a state $s \in \Sigma$ to its τ-successor $s' \in \tau(s)$ by referring to both unprimed and primed versions of the system variables. An unprimed version of a system variable refers to its value in s, while a primed version of the same variable refers to its value in s'. For example, the assertion $x' = x + 1$ states that the value of x in s' is greater by 1 than its value in s.
- $\mathcal{J} \subseteq \mathcal{T}$: A set of *just* transitions. The requirement of justice for $\tau \in \mathcal{J}$ disallows a computation in which τ is continually enabled beyond a certain point but taken only finitely many times.
- $\mathcal{C} \subseteq \mathcal{T}$: A set of *compassionate* transitions. The requirement of compassion for $\tau \in \mathcal{C}$ disallows a computation in which τ is enabled infinitely many times but taken only finitely many times.

The enableness of a transition τ can be expressed by the formula

$$En(\tau) : \ \exists V' \ \rho_\tau(V, V'),$$

which is true in s iff s has some τ-successor.

To ensure that every state $s \in \Sigma$ has at least one transition enabled on it, we standardly include in $\mathcal{T}$ the *idling* transition τ_I, whose transition relation is $\rho_I : V' = V$.

A *computation* of a fair transition system S is a model $\sigma : s_0, s_1, s_2, ...$, satisfying the following requirements:

- *Initiation:* s_0 satisfies Θ.
- *Consecution:* For each position $j > 0$, the state s_{j+1} is a τ-successor of the state s_j, for some $\tau \in \mathcal{T}$. In this case, we say that the transition τ is *taken* at position j in σ.
- *Justice:* For each $\tau \in \mathcal{J}$, there is no position $j \geq 0$ such that τ is continually enabled but not taken beyond j.
- *Compassion:* For each $\tau \in \mathcal{C}$, it is not the case that τ is enabled at infinitely many positions of σ but taken at only finitely many.

Let S: $\langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ be a system, and let $\mathcal{O} \subseteq V$ be a subset of the system variables of S, to which we refer as the *observable variables*. We say that the model $\tilde{\sigma}$ is an $\mathcal{O}$-observation of S if it can be obtained from a computation of S by finitely many stuttering and $\mathcal{O}$-preserving transformations. In the case that $\mathcal{O}$ is understood from the context, we refer to $\tilde{\sigma}$ simply as an *observation* of S.

For the presentation of programs we use the SPL programming language introduced in [MP95]. We refer the reader to [MP95] for the syntax and the semantics of SPL. Here, we shall only illustrate the interpretation of an SPL program as an FTS through an example.

Every program P is associated with a fair transition system S_P, which defines the semantics of P.

Consider program 2INC1 presented in Figure 1. The fair transition system

$$x, y : \textbf{integer where } x = y = 0$$
$$\left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } y := x + 1 \\ \ell_2\text{: } x := y + 1 \end{array}\right] \end{array}\right]$$

Fig. 1. Program 2INC1.

(FTS) associated with program 2INC1 is given by $S_P = \langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$, where

- *System Variables:* $V = \{x, y, \pi\}$,
  where π is the control variable. The domain of π consists of the set of indices of the locations occurring in 2INC1, i.e., $\{0, 1, 2\}$.
- *Initial Condition:* Θ : $x = 0 \wedge y = 0 \wedge at\_\ell_0$,
  where at_ℓ_0 stands for the assertion $\pi = 0$.
- *Transitions:* $\mathcal{T} = \{\tau_0, \tau_1, \tau_2, \tau_I\}$. The transition relations are :
  ρ_{τ_0} : $at\_\ell_0 \wedge at'\_\ell_1 \wedge x' = x \wedge y' = y$
  ρ_{τ_1} : $at\_\ell_1 \wedge at'\_\ell_2 \wedge x' = x \wedge y' = x + 1$
  ρ_{τ_2} : $at\_\ell_2 \wedge at'\_\ell_0 \wedge x' = y + 1 \wedge y' = y$
  ρ_{τ_I} : $\pi' = \pi \wedge x' = x \wedge y' = y$,
  where, at_ℓ_i stands for $\pi = i$ and at'_ℓ_j stands for $\pi' = j$.

- *Justice*: $\mathcal{J} = \{\tau_0, \tau_1, \tau_2\}$
- *Compassion*: $\mathcal{C} = \emptyset$

If P is a program and φ a formula, we say that φ is P-valid if every computation of P satisfies φ. If φ is an assertion, we say that φ is P-state valid if every state that appears in a computation of P satisfies φ. An assertion φ is said to be an *invariant* of program P if it is P-state valid.

3 Refinement of Fair Transition Systems

Let $S^{\mathcal{A}}$ and $S^{\mathcal{C}}$ be two fair transition systems, and let $\mathcal{O} \subseteq V^{\mathcal{A}} \cap V^{\mathcal{C}}$ be a set of observable variables common to both. We say that $S^{\mathcal{C}}$ *refines* $S^{\mathcal{A}}$ *relative to* $\mathcal{O}$ (equivalently, $S^{\mathcal{C}}$ $\mathcal{O}$-*refines* $S^{\mathcal{A}}$) and write $S^{\mathcal{C}} \sqsubseteq_{\mathcal{O}} S^{\mathcal{A}}$, if every $\mathcal{O}$-observation of $S^{\mathcal{C}}$ is an $\mathcal{O}$-observation of $S^{\mathcal{A}}$.

Usually, when we compare two systems, we assume that $\mathcal{O} = V^{\mathcal{A}} \cap V^{\mathcal{C}}$. This can always be achieved by renaming some of the non-observable variables of the two systems.

Two systems S_1 and S_2 such that $S_1 \sqsubseteq_{\mathcal{O}} S_2$ and $S_2 \sqsubseteq_{\mathcal{O}} S_1$, are said to be $\mathcal{O}$-*equivalent*. The equivalence relation is denoted by $\sim_{\mathcal{O}}$. The relation of refinement is reflexive and transitive. That is, for every system S, $S \sqsubseteq_{\mathcal{O}} S$, and if $S_1 \sqsubseteq_{\mathcal{O}} S_2$ and $S_2 \sqsubseteq_{\mathcal{O}} S_3$, then $S_1 \sqsubseteq_{\mathcal{O}} S_3$. When the set of observable variables is understood from the context, e.g. $\mathcal{O} = V^{\mathcal{A}} \cap V^{\mathcal{C}}$, we write $\sqsubseteq_{\mathcal{O}}$ and $\sim_{\mathcal{O}}$, simply as $\sqsubseteq$ and $\sim$.

Let $L^{\mathcal{C}} = V^{\mathcal{C}} - \mathcal{O}$ and $L^{\mathcal{A}} = V^{\mathcal{A}} - \mathcal{O}$ denote the sets of *private* variables of the concrete and abstract systems, respectively. Let

$$\alpha\colon\ L^{\mathcal{A}} \mapsto \mathcal{E}(V^{\mathcal{C}})$$

be a *substitution* that assigns to each private abstract variable $x \in L^{\mathcal{A}}$ an expression over the concrete variables. We denote the expression assigned to x by $x_\alpha(V^{\mathcal{C}})$, or simply x_α. For an assertion φ, we denote by $\varphi[\alpha]$ the assertion obtained by replacing each occurrence of a private abstract variable x occurring in φ by x_α and each primed variable x' by $x_\alpha((V^{\mathcal{C}})')$. Such substitution α induces a mapping m_α on states as follows:

Let $s^{\mathcal{C}}$ be a state which appears in a computation of $S^{\mathcal{C}}$. We refer to $s^{\mathcal{C}}$ as a concrete state. The abstract state $s^{\mathcal{A}} = m_\alpha(s^{\mathcal{C}})$ corresponding to the concrete state $s^{\mathcal{C}}$ is such that the value of each private abstract variable $x \in L^{\mathcal{A}}$ in $s^{\mathcal{A}}$ is the value of the expression x_α when evaluated in $s^{\mathcal{C}}$. For the observable variables, on which $S^{\mathcal{C}}$ and $S^{\mathcal{A}}$ agree, $s^{\mathcal{A}}[y] = s^{\mathcal{C}}[y]$, for each $y \in \mathcal{O}$. We say that m_α is a *state mapping* from $S^{\mathcal{C}}$ to $S^{\mathcal{A}}$, induced by α, with respect to the set $\mathcal{O}$ of observable variables.

The mapping m_α can be extended to a mapping between models. Thus, if $\sigma\colon\ s_0, s_1, \ldots$ is a model, then its image under m_α is $m_\alpha(\sigma)\colon\ m_\alpha(s_0), m_\alpha(s_1), \ldots$.

A state mapping m_α from $S^{\mathcal{C}}$ to $S^{\mathcal{A}}$ is called a *refinement mapping* if it maps computations of $S^{\mathcal{C}}$ to computations of $S^{\mathcal{A}}$.

The existence of a refinement mapping from $S^{\mathcal{C}}$ to $S^{\mathcal{A}}$ is a sufficient condition for $S^{\mathcal{C}}$ to be a refinement of $S^{\mathcal{A}}$.

The purpose of the refinement mapping is to represent the values of the private abstract variables in terms of the concrete system variables.

In some cases, we can identify a mapping $\kappa : \mathcal{T}^{\mathcal{C}} \mapsto \mathcal{T}^{\mathcal{A}}$, which maps each concrete transition to a corresponding abstract transition. The intended meaning of the mapping is that, when transforming a computation of $S^{\mathcal{C}}$ into a computation of $S^{\mathcal{A}}$, we can always mimic the effect of $\tau^{\mathcal{C}}$ in the concrete computation, by taking the corresponding abstract transition $\kappa(\tau^{\mathcal{C}})$ in the abstract computation. The *transition mapping* κ may also depend on the state.

In Figure 2 we present proof rule REF [KMP94], which can be used to prove that system $S^{\mathcal{C}}$ refines system $S^{\mathcal{A}}$, given a substitution α: $L^{\mathcal{A}} \mapsto \mathcal{E}(V^{\mathcal{C}})$ and a transition mapping $\kappa : \mathcal{T}^{\mathcal{C}} \mapsto \mathcal{T}^{\mathcal{A}}$. All premises of the rule are interpreted as

SF1.	$\Theta^{\mathcal{C}} \rightarrow \Theta^{\mathcal{A}}[\alpha]$	
SF2.	$\rho_{\tau^{\mathcal{C}}} \rightarrow \rho_{\kappa(\tau^{\mathcal{C}})}[\alpha]$	for every $\tau^{\mathcal{C}} \in \mathcal{T}^{\mathcal{C}}$
SF3.	$En(\tau^{\mathcal{A}})[\alpha] \Rightarrow \diamondsuit(\neg En(\tau^{\mathcal{A}}) \vee taken(\tau^{\mathcal{A}}))[\alpha]$	for every $\tau^{\mathcal{A}} \in \mathcal{J}^{\mathcal{A}}$
SF4.	$\square\diamondsuit En(\tau^{\mathcal{A}})[\alpha] \Rightarrow \diamondsuit taken(\tau^{\mathcal{A}})[\alpha]$	for every $\tau^{\mathcal{A}} \in \mathcal{C}^{\mathcal{A}}$
	$S^{\mathcal{C}} \sqsubseteq S^{\mathcal{A}}$	

Fig. 2. Rule REF.

$S^{\mathcal{C}}$-state valid (SF1 and SF2) or $S^{\mathcal{C}}$-valid (SF3 and SF4). This means that in proving the premises, we are allowed to add any invariant assertion established for system $S^{\mathcal{C}}$ to the left-hand side of the implication or entailment.

Premise SF1 requires that the concrete initial condition $\Theta^{\mathcal{C}}$ implies the substituted abstract initial condition $\Theta^{\mathcal{A}}[\alpha]$. Premise SF2 guarantees that any concrete transition $\tau^{\mathcal{C}}$ can be mimicked by the corresponding abstract transition $\kappa(\tau^{\mathcal{C}})$, after substitution. Premise SF3 requires that all abstract just transitions are treated with justice. That is, every abstract transition which is enabled at some position, must be disabled or taken at a later position. Under the substitution α this is a requirement that only refers to the concrete system variables $V^{\mathcal{C}}$, and can therefore be posed as a requirement that concrete computations must satisfy. In a similar way, premise SF4 requires that compassionate abstract transitions be treated with compassion. That is, any abstract compassionate transition which is enabled infinitely many times, must be taken at least once beyond every position, and hence must be taken infinitely many times.

Without loss of generality, we may assume that the justice and compassion sets of transitions are disjoint. This implies that every abstract transition appears in at most one premise of the form SF3 or SF4.

Let $\tau^{\mathcal{A}}$ be an abstract transition such that, for every $S^{\mathcal{C}}$-accessible state, there exists precisely one concrete transition $\tau^{\mathcal{C}}$ such that $\kappa(\tau^{\mathcal{C}}) = \tau^{\mathcal{A}}$. For such a transition, we may replace premises SF3 or SF4 for this transition by the following simplified premise:

$$\text{EF3.} \quad En(\tau^{\mathcal{A}})[\alpha] \rightarrow En(\kappa^{-1}(\tau^{\mathcal{A}}))$$

In [KMP94], rule REF was proposed as the only rule necessary for proving refinement. To support this, we considered substitutions α, in which the right-hand-side expressions assigned to private variables of $S^{\mathcal{A}}$ were general temporal functions. In the approach promoted here, we will use rule REF as only one of a larger assortment of rules, and then only for the special case that $V^{\mathcal{A}} = \mathcal{O} \subseteq V^{\mathcal{C}}$. Note that, for such cases, the substitution α becomes empty.

4 Equivalence Transformations of Programs

The definitions of refinement and of equivalence between fair transition systems can be transferred to programs in the obvious way.

4.1 Congruence and Equivalence Transformations

Let $P[S]$ be a program containing the statement S. We refer to $P[\,\cdot\,]$ as a *context* for statement S. Statements S_1 and S_2 are defined to be *congruent*, written

$$S_1 \approx S_2 \qquad \text{iff} \qquad P[S_1] \sim P[S_2], \quad \text{for every context } P[\,\cdot\,].$$

Obviously, any congruence between statements, gives rise to refinement transformations. Following are two useful congruences, called the *spacing* congruences:

- $[S] \approx [S; \mathbf{skip}]$
- $y := e \approx [\mathbf{skip}; y := e]$

The first congruence allows us to introduce a *skip* statement following any statement in the program. The second congruence allows to introduce a *skip* statement before an assignment. It can be shown that $[\mathbf{skip}; S]$ is not congruent to S for a general S[2].

A third useful equivalence can be stated as follows:

- If $at\_\ell \rightarrow e_1 = e_2$ is an invariant of program $P[\ell\colon\ y := e_1]$, then
$$P[\ell\colon\ y := e_1] \ \sim\ P[\ell\colon\ y := e_2]$$

This equivalence allows us to replace the assignment $y := e_1$ by the assignment $y := e_2$ if we have previously established that $e_1 = e_2$ whenever we visit this statement in the program.

---

[2] A typical counter-example is the program

skip or [await F**]** compared to **skip or [skip; await** F**]**

The two programs are not equivalent because the first program always terminate, while the second may get blocked in front of the **await** F statement

4.2 Safe Augmentation

Let $S^{\mathcal{A}} = \langle V^{\mathcal{A}}, \Theta^{\mathcal{A}}, \mathcal{T}^{\mathcal{A}}, \mathcal{J}^{\mathcal{A}}, \mathcal{C}^{\mathcal{A}} \rangle$ and $S^{\mathcal{C}} = \langle V^{\mathcal{C}}, \Theta^{\mathcal{C}}, \mathcal{T}^{\mathcal{C}}, \mathcal{J}^{\mathcal{C}}, \mathcal{C}^{\mathcal{C}} \rangle$ be two fair transition systems. Let $\mathcal{O} = V^{\mathcal{A}} \cap V^{\mathcal{C}}$ be the set of observable variables common to the two systems.

Let us consider again the question of proving that system $S^{\mathcal{C}}$ $\mathcal{O}$-refines system $S^{\mathcal{A}}$. Instead of looking for a refinement mapping, one might augment the concrete system $S^{\mathcal{C}}$ with some *auxiliary (ghost) variables* to obtain a new system $\widehat{S}$, such that $S^{\mathcal{C}} \sqsubseteq \widehat{S}$ and $\widehat{S} \sqsubseteq S^{\mathcal{A}}$. The definition of augmentation will ensure that the first refinement always holds and the second can often be proved with rule REF and the trivial substitution. The transitivity of the refinement relation will then permit us to conclude that $S^{\mathcal{C}} \sqsubseteq S^{\mathcal{A}}$.

The method of using auxiliary variables to prove that a lower level specification implements a higher level specification was discussed, among others, in [AL91].

System $\widehat{S}$: $\langle \widehat{V}, \widehat{\Theta}, \widehat{\mathcal{T}}, \widehat{\mathcal{J}}, \widehat{\mathcal{C}} \rangle$ is defined to be a *safe augmentation* of system S: $\langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$, if the following hold:

- $\widehat{V} = V \cup U$, for some set of auxiliary variables $U = \{u_1, \ldots, u_m\}$ disjoint from V;
- $\Theta \leftrightarrow \exists U : \widehat{\Theta}$ is state-valid;
- $\widehat{\mathcal{T}} = \{\widehat{\tau} | \tau \in \mathcal{T}\} \cup \{\tau_I\}$ such that for every transition τ, $\rho_\tau \leftrightarrow \exists U' : \rho_{\widehat{\tau}}$ is state-valid;
- $\widehat{\mathcal{J}} = \{\widehat{\tau} | \tau \in \mathcal{J}\}$ and $\widehat{\mathcal{C}} = \{\widehat{\tau} | \tau \in \mathcal{C}\}$.

The following theorem states that a safe augmentation of system S is equivalent to S.

Theorem 1. *Let S be a system with system variables V and let $\widehat{S}$ be a safe augmentation of S. The set of V-observations of $\widehat{S}$ coincides with the set of observations of S. It follows that S and $\widehat{S}$ are V-equivalent.*

Proof: First we show how to obtain, for any computation $\sigma : s_0, s_1, \ldots$ of S, a corresponding computation $\widehat{\sigma} : \widehat{s}_0, \widehat{s}_1, \ldots$ of $\widehat{S}$, such that the two computations agree on the interpretation of V.

According to the definition of computation of a fair transition system, $s_0 \models \Theta$. According to the definition of safe augmentation, $s_0 \models \exists U : \widehat{\Theta}$. Thus, there exists some state $\widehat{s}_0$ that might differ from s_0 only by the interpretation of the variables in U, such that $\widehat{s}_0 \models \widehat{\Theta}$. Assume that for some $n \geq 0$, $\widehat{s}_0, \ldots, \widehat{s}_n$ have been defined, such that for every j, $0 \leq j \leq n$, $\widehat{s}_j$ agrees with s_j on the interpretation of V. Let $\tau \in \mathcal{T}$ be the transition taken between states s_n and s_{n+1} of σ, i.e., $\langle s_n, s_{n+1} \rangle \models \rho_\tau(V, V')$ holds. Since s_n and $\widehat{s}_n$ coincide on V, we have also that $\langle \widehat{s}_n, s_{n+1} \rangle \models \rho_\tau(V, V')$ holds. According to the definition of $\widehat{S}$, $\langle \widehat{s}_n, s_{n+1} \rangle \models \exists U' : \rho_{\widehat{\tau}}(\widehat{V}, \widehat{V}')$ holds, hence there exists some state $\widehat{s}_{n+1}$, agreeing with s_{n+1} on the interpretation of V, such that $\langle \widehat{s}_n, \widehat{s}_{n+1} \rangle \models \rho_{\widehat{\tau}}(\widehat{V}, \widehat{V}')$ holds.

Clearly, for every $n \geq 0$, if $\widehat{\tau}$ is enabled in state $\widehat{s}_n$ then τ is enabled in state s_n and if τ is taken in state s_n then $\widehat{\tau}$ is taken in state $\widehat{s}_n$. Therefore $\widehat{\sigma}$ is a computation of $\widehat{S}$ which agrees with σ on the interpretation of V.

In the other direction, it is straightforward to show that every computation of $\widehat{S}$ is, without any changes, a computation of S.

It follows that the set of V-observations of S and of $\widehat{S}$ coincide. □

Applied to programs, a safe augmentation $\widehat{P}$ of a program P can be obtained by

- Declaring auxiliary variables $U = \{u_1, \ldots, u_k\}$ in the program. The declarations can specify initial values.
- Assignments to auxiliary variables of the form $u_i := e$ may be added to P by grouping them together with existing statements.

The grouping may be done either by explicitly grouping an auxiliary assignment with an existing statement or, if the existing statement is an assignment, by transforming the assignment into a multiple assignment with additional entries for the auxiliary variables.

For example, in Figure 3, we present program **B**, which is a safe augmentation of program **A**.

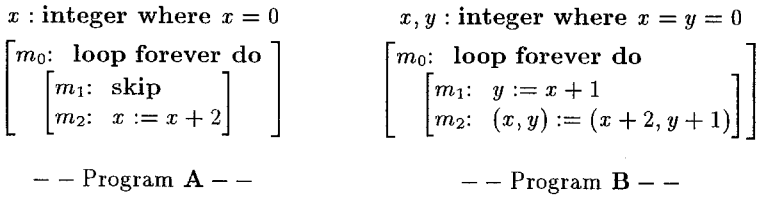

Fig. 3. Example of safe augmentation.

A unique feature of our approach is that the expression e appearing on the right-hand-side of an augmenting assignment may refer to future values of all the variables in V. We allow two forms of future-dependent expressions:

- An assignment of the form
  $$u := \textbf{if } \varphi \textbf{ then } e_1 \textbf{ else } e_2,$$
  where φ is a future temporal formula and e_1, e_2 are normal (future-independent) expressions. To evaluate this expression, we check whether φ holds at the current position of the computation and accordingly select the current value of e_1 or the current value of e_2.
- An assignment of the form
  $$u := e@\varphi,$$
  where φ is a future temporal formula and e is a normal expression. The value of this expression is the value of e at the first position following the current one at which φ holds. If φ holds at no future position, $e@\varphi$ is taken to be some default value which, in the case of integer variables, is 0.

5 Proof Rules for Invariance Properties of Programs

Proving that the conditions of rule REF hold might require some additional invariance properties. Usually, by an invariance property of a program P we mean a P-valid formula $\Box p$ where p is an assertion. Since our augmentation allows assigning future-dependent expressions to auxiliary, the notion of invariance property must be extended to allow p to be a future temporal formula.

Rule INV of Figure 4 can be used to prove invariance properties of programs.

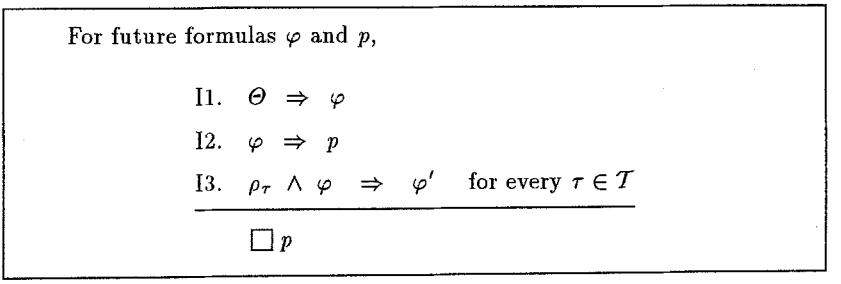

Fig. 4. Rule INV (invariance).

In establishing instances of the verification conditions required by premise I3, it is necessary to relate φ to its primed version φ'. For the simple case that φ is an assertion, φ' is obtained simply by priming each occurrence of a variable appearing in φ. For the extended cases considered here, φ may be a future formula. In this case, we may be helped by the following expansion formulas:

$$\begin{array}{rcl} \Box\varphi & \Leftrightarrow & \varphi \wedge (\Box\varphi)' \\ \Diamond\varphi & \Leftrightarrow & \varphi \vee (\Diamond\varphi)' \\ \varphi_1 \,\mathcal{U}\, \varphi_2 & \Leftrightarrow & \varphi_2 \vee (\varphi_1 \wedge (\varphi_1 \,\mathcal{U}\, \varphi_2)') \end{array}$$

The following entailments (implications holding at every position) can be used for simplifying future expressions of the form $e@\varphi$:

$$\begin{array}{rcl} \Box\neg\varphi & \Rightarrow & e@\varphi = 0 \\ \neg\varphi & \Rightarrow & e@\varphi = (e@\varphi)' \\ \varphi & \Rightarrow & e@\varphi = e \end{array}$$

Rule C-INV in Figure 5 can be used to prove invariance properties of the form $p \Rightarrow \Box q$. All the formulas occurring as premises of the verification rules are

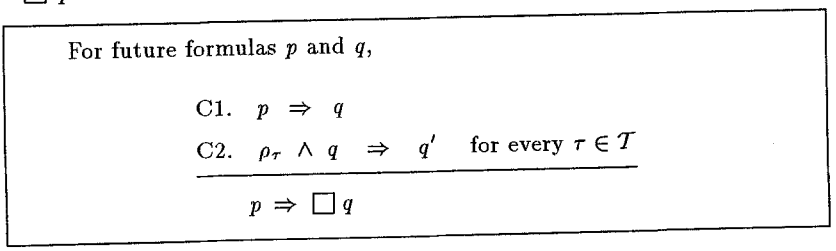

Fig. 5. Rule C-INV (invariance with precondition).

meant to be P-valid formulas.

6 Examples

In this section, we will illustrate the proposed proof methods by establishing equivalences of pairs of programs of increasing complexities.

6.1 Incrementation by 1 and 2

Consider programs 1INC2 and 2INC1, presented in Figure 6. We wish to show that these two programs are equivalent.

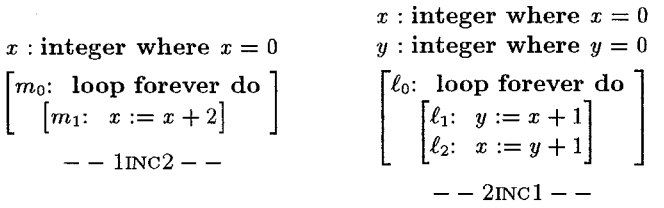

Fig. 6. Two equivalent programs.

In Figure 7, we present a sequence of equivalence transformations leading from 1INC2 to 2INC1 and back.

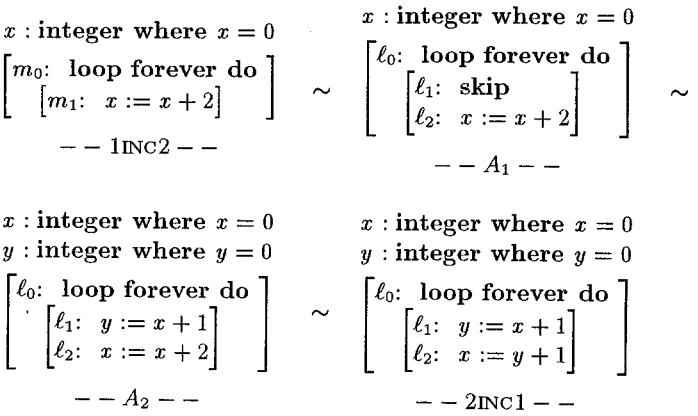

Fig. 7. Transformation sequence.

The transformation from 1INC2 to program A_1 is a spacing transformation. The transformation from A_1 to A_2 is justified by safe augmentation. The equivalence between A_2 and 2INC1 is justified by the invariance of the assertion

$$at\_\ell_2 \quad \rightarrow \quad y = x + 1,$$

which is invariant for both programs.

6.2 Non-Deterministic Assignment

The second example proves equivalence between programs P_3 and P_4, presented in Figure 8. Program P_3 is an infinite loop, whose body consists of nondeterministically choosing a value from the set $\{-1, +1\}$ and assigning it to the observable

variable x. This is done in one step, by the statement $x :\in \{-1, +1\}$. Program P_4 first chooses nondeterministically a value from $\{-1, +1\}$ and assigns it to the local variable y. Only in the next step, the chosen value is assigned to x.

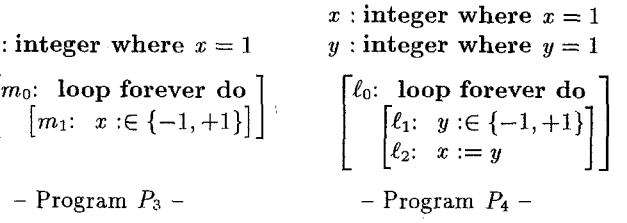

Fig. 8. Programs P_3 and P_4.

As a first step, we apply a spacing transformation to program P_3, obtaining program P_5 of Figure 9. We proceed to show that P_4 is equivalent to P_5. To

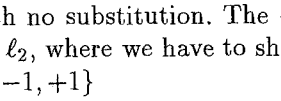

Fig. 9. Program P_5, a spaced version of P_3.

show that P_4 refines P_5, i.e.

$$P_4 \quad \sqsubseteq \quad P_5,$$

we establish first the invariance of the assertion

$$\varphi_1: \quad at\_\ell_2 \;\rightarrow\; y \in \{-1, +1\}.$$

We then apply rule REF with no substitution. The only nontrivial verification condition involves transition ℓ_2, where we have to show the implication

$$x' = y \;\rightarrow\; x' \in \{-1, +1\}$$

This is implied by the invariant assertion φ_1.

To prove the other refinement $P_5 \sqsubseteq P_4$, we first construct P_6 (presented in Figure 10), a safe augmentation of P_5. Program P_5 is augmented with an

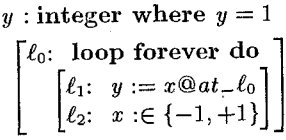

Fig. 10. Program P_6: a safe augmentation of P_5

auxiliary variable y which has to guess the value assigned to x in the next step. Thus, y is defined using the @ operator, in terms of *future values* of the observable variable x.

As a first step in showing that P_6 refines P_4, we establish for program P_6 the invariant

$$at\_\ell_2 \;\Rightarrow\; y = x@at\_\ell_0.$$

To do so, we use rule INV with the appropriate simplifications of the primed value of $x@at\_\ell_0$.

Next, we apply rule REF with the trivial substitution to prove that program P_6 refines P_4. The only nontrivial verification condition involves premise SF2 for transition l_2. For this, the invariant $at\_\ell_2 \Rightarrow y = x@at\_\ell_0$ and the property $\neg at\_\ell_0 \wedge at'\_\ell_0 \Rightarrow x' = x@at\_\ell_0$ of the @ operator are necessary.

Due to space limitations, we do not present here a more advanced example, comparing two versions of a buffer which may lose messages. For detailed presentation of this example, we refer the reader to [MP96].

7 Discussion

Two important questions require further discussion. The first is an assessment of the proving power of the proposed method. In particular, it would be useful if we can establish some completeness result for the method. A second interesting question is a more detailed comparison with the methods proposed by Abadi and Lamport in [AL91], and a reexamination of our initial claim that our future-dependent augmentation can be used to eliminate prophecy variables as they are introduced in [AL91]. The two questions are strongly related.

In Section 4, we introduced two restricted types of future-dependent expressions that may appear in augmentations. They were the expressions

$$\textbf{if } \varphi \textbf{ then } e_1 \textbf{ else } e_2 \qquad \text{and} \qquad e@\varphi,$$

where φ is a future temporal formula. We do not know yet whether completeness of the method can be established when only these two types of expressions are allowed. On the other hand, we may consider a more general future-dependent expression of the form

$$\min v.\, \varphi(v),$$

where v is a rigid variable and $\varphi(v)$ is a future temporal formula that may refer to v. Without loss of generality, we will only consider the case that v ranges over the naturals. The value of this expression at position j of a model σ is defined as follows:

$$\min v.\, \varphi(v) = \begin{cases} \text{minimal } v \text{ such that } (\sigma, j) \models \varphi(v) & \text{if } (\sigma, j) \models \exists v.\varphi(v) \\ 0 & \text{otherwise} \end{cases}$$

The *min* expression is at least as expressive as the two special cases considered in Section 4. The conditional expression (**if** φ **then** e_1 **else** e_2) can be expressed as

$$\min v.\, (\varphi \wedge v = e_1) \vee (\neg\varphi \wedge v = e_2).$$

Similarly, the expression $e@\varphi$ can be represented by a *min* expression of the form

$$\min v.\, (\neg\varphi)\, \mathcal{U}\, (\varphi \wedge v = e).$$

In a related way, also Abadi & Lamport's prophecy variables can be expressed using an appopriate *min* expression. Recall that the essence of prophecy variables is given by a state function f and the search for a flexible variable y satisfying

$$\Box(y = f(\bigcirc y)).$$

That is, an infinite sequence of values satisfying the descending recurrence rule

$$y_i = f(y_{i+1}),$$

for every $i = 0, 1, \ldots$. The requirement that f has the finite-image property is necessary in order to ensure that there exists at least one such sequence y. It is obvious that we can use a *min* expression to represent this special case as

$$\min v.\, \exists y \left(v = y \ \wedge \ \Box(y = f(\bigcirc y)) \right).$$

Using *min* expressions for augmentation, we can establish the following result:

Theorem 2 (Completeness). *The combination of rule* REF *of Figure 2 for the case of empty substitutions and safe augmentation, using min expressions, is complete for proving refinements between fair transition systems.*

We refer the reader to [MP96] for a proof of this theorem.

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[dBdRR90] J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness.* Lecture Notes in Computer Science 430. Springer-Verlag, 1990.

[Jon87] B. Jonsson. Modular verification of asynchronous networks. In *Proc. 6th ACM Symp. Princ. of Dist. Comp.*, pages 152–166, 1987.

[KMP94] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lect. Notes in Comp. Sci.*, pages 273–346. Springer-Verlag, 1994.

[Lam92] L. Lamport. The existence of refinement mapping. TLA Note 92-03-19, March 1992.

[Luc68] P. Lucas. Two constructive realizations of the block concept and their equivalences. Technical Report Technical Report TR 25.085, IBM Laboratory Vienna, 1968.

[MP91a] Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.

[MP91b] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[MP96] M. Marcus and A. Pnueli. Using ghost variables to prove refinement. Technical report, Weizmann Institute, 1996.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[Rey81] J.C. Reynolds. *The Craft of Programming.* Prentice-Hall, Engelwood Cliffs, NJ, 1981.