

Eternity Variables to Simulate Specifications

Wim H. Hesselink

Dept. of Mathematics and Computing Science University of Groningen
P.O.Box 800, 9700 AV Groningen, The Netherlands
wim@cs.rug.nl, <http://www.cs.rug.nl/~wim>

Corrected MPC contribution (version whh263n, June 20, 2002)

Abstract. Simulation of specifications is introduced as a unification and generalization of refinement mappings, history variables, forward simulations, prophecy variables, and backward simulations.

Eternity variables are introduced as a more powerful alternative for prophecy variables and backward simulations. This formalism is semantically complete: every simulation is a composition of a forward simulation, an extension with eternity variables, and a refinement mapping. This result does not need finite invisible nondeterminism and machine closure as in the Abadi-Lamport Theorem. Internal continuity is replaced by preservation of quiescence.

1 Introduction

In the theory of Abadi and Lamport [1] on the existence of refinement mappings, a specification is a state machine with a supplementary property. Behaviours of a specification are infinite sequences of states. Behaviours become visible by means of an observation function. A specification implements another one when all visible behaviours of the first one are visible behaviours of the second one.

Under some technical assumptions (finite invisible nondeterminism and internal continuity of L , machine-closedness of K), Abadi and Lamport [1] proved that, when a specification K implements a specification L , there exists an extension M of K with history variables and prophecy variables together with a refinement mapping from M to L . Such a result is described as semantic completeness.

Lamport [8] draws from it the following conclusion: “We have learned that to be complete, assertional methods for reasoning about specifications must allow dummy variables (or their equivalent) – not only history variables for recording past behavior, but prophecy variables for predicting future behavior.” Lamport’s argument is not completely convincing, however, since adding these dummy variables is not enough for completeness when the technical assumptions are not met.

One may argue that imposing finiteness should be acceptable since computer storage is always finite. Yet, it is often useful to prove algorithms about integers or strings that can only be correct on idealized computers with infinite storage. A good proof methodology should be applicable to such algorithms. We therefore prefer proof methods without finiteness assumptions.

In this paper we remove these technical conditions of [1] by introducing eternity variables as an alternative for prophecy variables. We first extend the conceptual framework by unifying the ideas of refinement mapping and extension with history variables or prophecy variables to the concepts of simulation. Actually, the term “simulation” has been introduced by Milner [12] in 1971. He used it for a kind of relation, which was later called downward or forward simulation to distinguish it from so-called upward or backward simulation [4, 11]. It seems natural and justified to reintroduce the term “simulation” for the common generalization.

We then introduce eternity variables, which turn out to be simpler and more powerful than prophecy variables and backward simulations. We prove semantic completeness: every simulation that “preserves quiescence” is a composition of a forward simulation, an extension with an eternity variable and a refinement mapping.

1.1 Stuttering Specifications

Although they can change roles, let us call the implementing specification the concrete one and the implemented specification the abstract one. Since the concrete specification may have to perform computation steps that are not needed for the abstract specification, we allow all specifications to stutter: a behaviour remains a behaviour when a state in it is duplicated.

At this point we deviate from [1] where it is allowed that the concrete specification is faster than the abstract one: a concrete behaviour may have to be slowed down by adding stutterings in order to match the abstract behaviour. In our view, this hides an important aspect of the idea of implementation. When the concrete specification needs less steps than the abstract one, perhaps the abstract one is not abstract enough.

In this paper, we therefore present a stricter theory than [1]. This results in a finer hierarchy of implementations. In [5], we treat both hierarchies and provide some more proofs.

1.2 Simulations of Specifications

A refinement mapping is a function between the states that, roughly speaking, preserves the initial states, the next-state relation and the supplementary property. Adding history or prophecy variables to the state gives rise to relations, called downward and upward simulations in [4], forward and backward simulations in [11]. We unify these concepts by introducing simulations.

Our simulations are certain binary relations. For the sake of simplicity, we treat binary relations as sets of pairs, with some notational conventions. Since we use $X \rightarrow Y$ for functions from X to Y , and $P \Rightarrow Q$ for implication between predicates P and Q , we write $F : K \rightarrow L$ to denote that relation F is a simulation of specifications from K to L . We hope the reader is not confused by the totally unrelated arrows \rightarrow used in [2].

Our first main result is a completeness theorem: a specification implements another one if and only if there is a certain simulation between them. This shows that our concept of simulation is general enough to capture the relevant phenomena.

1.3 Eternity Variables and Completeness

We introduce eternity variables as an alternative for backward simulations or prophecy variables, with a similar kind of “prescient behaviour”. An eternity variable is a kind of logical variable with a value constrained by the current execution.

Technically, an eternity variable is an auxiliary variable m , which is initialized nondeterministically and is never modified. The value of m is constrained by a relation with the state. A behaviour that would violate such a constraint, is discarded. The verifier of a program has to prove that the totality of constraints is not contradictory. In our current examples, we do this by taking m to be an infinite array and to ensure that the conditions constrain different elements of array m .

Our Abadi-Lamport theory goes as follows. Every specification K has a so-called unfolding $K^\#$ [11] with a forward simulation $K \rightarrow K^\#$. Given a simulation $F : K \rightarrow L$ that preserves quiescence, we construct an intermediate specification W as an extension of $K^\#$ with an eternity variable, together with a refinement mapping $W \rightarrow L$, such that the composition of the simulations $K \rightarrow K^\#$ and $K^\# \rightarrow W$ and $W \rightarrow L$ is a subset of F .

1.4 Overview

In Sect. 1.5, we briefly discuss related work. Sect. 1.6 contains technical material on relations and lists. In Sect. 1.7, we treat stuttering and temporal operators. We introduce specifications and simulations, and prove the characterizing theorem for them in Sect. 2. In Sect. 3, we present the theory of forward and backward simulations in our setting and introduce quiescence and preservation of quiescence. Eternity variables are introduced in Sect. 4, where we also prove soundness and semantic completeness. Conclusions are drawn in Sect. 5.

New results in this paper are the completeness theorem in Sect. 2, the concepts of quiescence and preservation of quiescence, and the eternity variables with their soundness and completeness in Sect. 4.

1.5 Related Work

Our primary inspiration was [1] of Abadi and Lamport. Lynch and Vaandrager [11] and Jonsson [6] present forward and backward simulations and the associated results on semantic completeness in the closely related settings of untimed automata and fair labelled transition systems. Our investigation was triggered by the paper [3] of Cohen and Lamport on Lipton’s Theorem [10] about refining

atomicity. Jonsson, Pnueli, and Rump [7] present another way of proving refinement that avoids the finiteness assumptions of backward simulations. They use a very flexible concept of refinement based on so-called pomsets, but have no claim of semantic completeness.

1.6 Relations and Lists

We treat a binary relation as a set of pairs. So, a binary relation between sets X and Y is a subset of the cartesian product $X \times Y$. We use the functions fst and snd given by $fst(x, y) = x$ and $snd(x, y) = y$. A binary relation on X is a subset of $X \times X$. The identity relation 1_X on X consists of all pairs (x, x) with $x \in X$. Recall that a binary relation A on X is called *reflexive* iff $1_X \subseteq A$. The *converse* $cv(A)$ of a binary relation A is defined by $cv(A) = \{(x, y) \mid (y, x) \in A\}$.

For a state x and a binary relation A between sets of states, we define the set $(x; A) = \{y \mid (x, y) \in A\}$. For binary relations A and B , the composition $(A; B)$ is defined to consist of all pairs (x, z) such that there exists y with $(x, y) \in A$ and $(y, z) \in B$. A function $f : X \rightarrow Y$ is identified with its graph $\{(x, f(x)) \mid x \in X\}$ which is a binary relation between X and Y . We thus have $(x; f) = \{f(x)\}$ for every $x \in X$. The composition of functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ is a function $g \circ f : X \rightarrow Z$, which equals the relational composition $(f; g)$.

We use lists to represent consecutive values during computations. If X is a set, we write X^* for the set of the finite lists, X^+ for the set of the nonempty finite lists, and X^ω for the set of infinite lists over X .

We write $\ell(xs)$ for the length of list xs . The elements of xs are xs_i for $0 \leq i < \ell(xs)$. If xs is a list of length $\ell(xs) \geq n$, we define $(xs|n)$ to be its prefix of length n . We define $last : X^+ \rightarrow X$ to be the function that returns the last element of a nonempty finite list, and $init : X^+ \rightarrow X^*$ to be the function that removes the last element from such a list.

A function $f : X \rightarrow Y$ induces a function $f^\omega : X^\omega \rightarrow Y^\omega$. For a binary relation $F \subseteq X \times Y$, we have induced binary relations $F^* \subseteq X^* \times Y^*$ and $F^\omega \subseteq X^\omega \times Y^\omega$ given by

$$(xs, ys) \in F^* \quad \equiv \quad \ell(xs) = \ell(ys) \quad \wedge \quad (\forall i : i < \ell(xs) : (xs_i, ys_i) \in F) ,$$

and similarly for F^ω .

1.7 Stuttering and Properties

We define a list xs to be an *unstuttering* of a list ys , notation $xs \preceq ys$, iff xs is obtained from ys by replacing some finite nonempty subsequences ss of consecutive equal elements of ys with their first elements ss_0 . The number of such subsequences that are replaced may be infinite. For example, $(abb)^\omega$ is an unstuttering of $(aaabbb)^\omega$. It can be proved that relation \preceq is a partial order.

A finite list xs is called *stutterfree* iff every pair of consecutive elements differ. An infinite list xs is called *stutterfree* iff it stutters only after reaching its final state, i.e., iff $xs_i = xs_{i+1}$ implies $xs_{i+1} = xs_{i+2}$ for all i . In either case, xs is stutterfree iff xs is minimal (in X^* or X^ω) for the order relation \preceq .

A subset P of X^ω is called a *property over X* iff $xs \preceq ys$ implies that $xs \in P \equiv ys \in P$. This definition is equivalent to the one of [1].

We write $\neg P$ to denote the complement (negation) of a property P . We write $Suf(xs)$ to denote the set of infinite suffixes of an infinite list xs . We define $\Box P$ (always P), and $\Diamond P$ (sometime P) as the properties given by

$$\begin{aligned} xs \in \Box P &\equiv Suf(xs) \subseteq P, \\ \Diamond P &= \neg \Box \neg P. \end{aligned}$$

For $U \subseteq X$ and $A \subseteq X \times X$, the subsets $\llbracket U \rrbracket$ and $\llbracket A \rrbracket$ of X^ω are defined by

$$\begin{aligned} xs \in \llbracket U \rrbracket &\equiv xs_0 \in U, \\ xs \in \llbracket A \rrbracket &\equiv (xs_0, xs_1) \in A; \end{aligned}$$

$\llbracket U \rrbracket$ is always a property; $\Box \llbracket A \rrbracket$ is a property when A is reflexive.

2 Specifications and Simulations

In this section we introduce the central concepts of the theory. Following [1], we define specifications in Sect. 2.1. In 2.2, we define simulations and refinement mappings. Sect. 2.3 contains an example of them. In Sect. 2.4, we define visible specifications and their implementation relations, and we prove that simulations characterize the implementations between visible specifications.

2.1 Specifications

A *specification* is defined to be a tuple $K = (X, Y, N, P)$ where X is a set, Y is a subset of X , N a reflexive binary relation on X , and P is a property over X . The set X is called the *state space*, its elements are called *states*, the elements of Y are called *initial states*. Relation N is called the *next-state* relation. The set P is called the *supplementary* property.

We define an *execution* of K to be a nonempty list xs over X for which every pair of consecutive elements belongs to N . An execution of K is called *initial* iff $xs_0 \in Y$. We define a *behaviour* of K to be an infinite and initial execution xs of K with $xs \in P$. We write $Beh(K)$ to denote the set of behaviours of K .

It is easy to see that $Beh(K) = \llbracket Y \rrbracket \cap \Box \llbracket N \rrbracket \cap P$. It follows that $Beh(K)$ is a property. The requirement that relation N is reflexive is imposed to allow stuttering: if xs is a behaviour of K , any list ys obtained from xs by repeating elements of xs or by removing subsequent duplicates is also a behaviour of K .

The components of specification $K = (X, Y, N, P)$ are denoted $states(K) = X$, $init(K) = Y$, $step(K) = N$ and $prop(K) = P$. A state is called *reachable* iff it occurs in an initial execution of K . A subset of $states(K)$ is called an *invariant* iff it contains all reachable states.

Specification K is defined to be *machine closed* [1] iff every finite initial execution of K can be extended to a behaviour of K . Concrete specifications are often machine closed.

2.2 Simulations and Refinement Mappings

Let K and L be specifications. Recall that a relation F between $states(K)$ and $states(L)$ induces a relation F^ω between the sets $(states(K))^\omega$ and $(states(L))^\omega$. Relation F is called a *simulation* $K \rightarrow L$ iff, for every behaviour $xs \in Beh(K)$, there exists a behaviour $ys \in Beh(L)$ with $(xs, ys) \in F^\omega$.

It is easy to verify that simulations can be composed: if F is a simulation $K \rightarrow L$ and G is a simulation $L \rightarrow M$, the composed relation $(F; G)$ is a simulation $K \rightarrow M$.

It should be noted that the mere existence of a simulation $F : K \rightarrow L$ does not imply much. If $F : K \rightarrow L$ and G is a relation with $F \subseteq G$, then $G : K \rightarrow L$. Therefore, the smaller the simulation, the more information it carries.

A function $f : states(K) \rightarrow states(L)$ is called a *refinement mapping* [1] from K to L iff $f(x) \in init(L)$ for every $x \in init(K)$, and $(f(x), f(x')) \in step(L)$ for every pair $(x, x') \in step(K)$, and $f^\omega(xs) \in prop(L)$ for every $xs \in Beh(K)$. It is easy to verify that a refinement mapping $f : states(K) \rightarrow states(L)$, when regarded as a relation as in Sect. 1.6, is a simulation $K \rightarrow L$.

We often encounter the following situation. A specification L is regarded as an extension of specification K with a variable of a type M iff $states(L)$ is (a subset of) the cartesian product $states(K) \times M$ and the function $fst : states(L) \rightarrow states(K)$ is a refinement mapping. The second component of the states of L is then regarded as the variable added. The extension may be called a refinement extension iff the converse $cvf = cv(fst)$ is a simulation $K \rightarrow L$.

2.3 Example

We give an example to show how specifications correspond to programs with supplementary properties and we give a simple case of a refinement mapping and a simulation.

The first specification $K0$ is based on guarded commands with a fairness assumption. We need only one integer variable.

```

var j: Nat := 0 ;
do true   -> j := j + 1 ;
[] j > 0  -> j := 0 ;
od .

```

The first line declares and initializes the variable. Type Nat stands for the natural numbers. We impose the fairness assumption that the second alternative is chosen infinitely often. This program corresponds to the specification $K0$ with $states(K0) = \mathbb{N}$ and $init(K0) = \{0\}$ and relation $step(K0)$ given by

$$(i, j) \in step(K0) \equiv j = i + 1 \vee j = 0 \vee j = i .$$

The third alternative in $step(K0)$ serves to allow stuttering. We take the supplementary property to be $prop(K0) = \square \diamond \llbracket > \rrbracket$, which expresses that the value of j infinitely often decreases.

In our second specification, $K1$, we use two integer variables k and m in the guarded commands program (with demonic choices):

```

var k: Nat := 0, m: Nat := 0 ;
do k < m -> k := k + 1 ;
[] k = m -> m := 0 ; k := 0 ;
[] k = 0 -> k := 1 ; choose m > 0 ;
od .

```

We assume that each of the alternatives is executed atomically. The program is formalized by taking $states(K1) = \mathbb{N} \times \mathbb{N}$ with k as the first component and m as the second. The initial state has $k = m = 0$. So $init(K1) = \{(0, 0)\}$. The next-state relation $step(K1)$ is given by

$$\begin{aligned}
((k, m), (k', m')) \in step(K1) &\equiv \\
&(k < m \wedge k' = k + 1 \wedge m' = m) \\
&\vee (k = m \wedge k' = m' = 0) \\
&\vee (k = 0 \wedge k' = 1 \leq m') \\
&\vee (k' = k \wedge m' = m) .
\end{aligned}$$

The last alternative serves to allow stuttering. The supplementary property only has to ensure that behaviours do not stutter indefinitely. We thus take $prop(K1) = \square \diamond \llbracket \neq \rrbracket$.

The function $f_{1,0} : states(K1) \rightarrow states(K0)$ given by $f_{1,0}(k, m) = k$ is easily seen to be a refinement mapping $K1 \rightarrow K0$.

More interesting is the converse relation $F_{0,1} = cv(f_{1,0})$. It is not difficult to show by ad-hoc methods that $F_{0,1}$ is a simulation $K0 \rightarrow K1$. Every behaviour of $K0$ can be mimicked by $K1$ by choosing adequate values of m . In Sect. 4.2 we show how an eternity variable can be used to prove this more systematically.

2.4 Visibility and Completeness of Simulation

We are usually not interested in all details of the states, but only in certain aspects of them. This means that there is a function from $states(K)$ to some other set that we regard as an observation function. A *visible specification* is therefore defined to be a pair (K, f) where K is a specification and f is some function defined on $states(K)$. Deviating from [1], we define the set of observations by

$$Obs(K, f) = \{f^\omega(xs) \mid xs \in Beh(K)\} .$$

Note that $Obs(K, f)$ need not be a property. If xs is an observation and $ys \preceq xs$, then ys need not be an observation.

Let (K, f) and (L, g) be visible specifications with the functions f and g mapping to the same set. Then (K, f) is said to *implement* (L, g) iff $Obs(K, f)$ is contained in $Obs(L, g)$. This concept of implementation is stronger than that of [1]: we do not allow that an observation of (K, f) can only be mimicked by (L, g) after inserting additional stutterings.

Our concept of simulation is motivated by the following completeness theorem, the proof of which is rather straightforward.

Theorem 0. Consider visible specifications (K, f) and (L, g) where f and g are functions to the same set. We have that (K, f) implements (L, g) if and only if there is a simulation $F : K \rightarrow L$ with $(F; g) \subseteq f$.

Proof. The proof is by mutual implication.

First, assume the existence of a simulation $F : K \rightarrow L$ with $(F; g) \subseteq f$. Let $zs \in Obs(K, f)$. We have to prove that $zs \in Obs(L, g)$. By the definition of Obs , there exists $xs \in Beh(K)$ with $zs = f^\omega(xs)$. Since F is a simulation, there exists $ys \in Beh(L)$ with $(xs, ys) \in F^\omega$. For every number n , we have $(xs_n, ys_n) \in F$ and, hence, $(xs_n, g(ys_n)) \in (F; g) \subseteq f$ and, hence, $g(ys_n) = f(xs_n) = zs_n$. This implies that $zs = g^\omega(ys) \in Obs(L, g)$.

Next, assume that (K, f) implements (L, g) . We define relation F between $states(K)$ and $states(L)$ by $F = \{(x, y) \mid f(x) = g(y)\}$. For every pair $(x, z) \in (F; g)$ there exists y with $(x, y) \in F$ and $(y, z) \in g$; we then have $f(x) = g(y) = z$. This proves $(F; g) \subseteq f$. It remains to prove that F is a simulation $K \rightarrow L$. Let $xs \in Beh(K)$. Since $Obs(K, f) \subseteq Obs(L, g)$, there is $ys \in Beh(L)$ with $f^\omega(xs) = g^\omega(ys)$. We thus have $(xs, ys) \in F^\omega$. This proves that F is a simulation $K \rightarrow L$. \square

3 Special Simulations

In this section we introduce forward and backward simulations as special kinds of simulations. Forward simulations are introduced in 3.1. They correspond to refinement mappings and to the well-known addition of history variables. We give an example of a forward simulation in Sect. 3.2. In Sect. 3.3, we introduce the unfolding [11] of a specification, which plays a key role in several proofs of semantic completeness. Backward simulations are introduced in Sect. 3.4. In Sect. 3.5, we introduce strictness of simulations which concept is needed in the completeness theorem of Sect. 4.

3.1 Forward Simulations

The easiest way to prove that one specification simulates (the behaviour of) another is by starting at the beginning and constructing the corresponding behaviour in the other specification inductively. This requires a condition embodied in so-called forward or downward simulations [4, 11], which go back at least to [12]. They are defined as follows.

A relation F between $states(K)$ and $states(L)$ is defined to be a *forward simulation* from specification K to specification L iff

- (H0) For every $x \in init(K)$, there is $y \in init(L)$ with $(x, y) \in F$.
- (H1) For every pair $(x, y) \in F$ and every x' with $(x, x') \in step(K)$, there is y' with $(y, y') \in step(L)$ and $(x', y') \in F$.
- (H2) Every infinite initial execution ys of L with $(xs, ys) \in F^\omega$ for some $xs \in Beh(K)$ satisfies $ys \in prop(L)$.

A composition of forward simulations need not be a forward simulation, since condition (H2) is not compositional. Every refinement mapping, when regarded as a relation, is a forward simulation. An auxiliary variable added to the state space via a forward simulation is called a history variable [1]; in such cases, the relation is called a history relation in [11]. The definition of forward simulations is justified by the following well-known result:

Lemma. Every forward simulation F from K to L is a simulation $K \rightarrow L$. \square

3.2 An Example of a Forward Simulation

We extend specification $K0$ of Sect. 2.3 with two history variables \mathbf{n} and \mathbf{q} . Variable \mathbf{n} counts the number of backjumps of \mathbf{j} , while \mathbf{q} is an array that records the values from where \mathbf{j} jumped.

```

var j: Nat := 0, n: Nat := 0,
    q: array Nat of Nat := ([Nat] 0) ;
do true  -> j := j + 1 ; q[n] := q[n] + 1 ;
[] j > 0  -> j := 0 ; n := n + 1 ;
od .

```

This yields a specification $K2$ with the supplementary property $\square\Diamond[[j > j']]$ where j' stands for the value of j in the next state. Its next-state relation $step(K2)$ is given by

$$\begin{aligned}
((j, n, q), (j', n', q')) \in step(K2) &\equiv \\
&(j' = j + 1 \wedge q'[n] = q[n] + 1 \wedge n' = n \wedge (\forall i : i \neq n : q'[i] = q[i])) \\
&\vee (j' = 0 \wedge q' = q \wedge n' = n + 1) \\
&\vee (j' = j \wedge q' = q \wedge n' = n) .
\end{aligned}$$

It is easy to verify that the function $f_{2,0} : states(K2) \rightarrow states(K0)$ given by $f_{2,0}(j, n, q) = j$ is a refinement mapping. Its converse $F_{0,2} = cv(f_{2,0})$ is a forward simulation $K0 \rightarrow K2$. Indeed, the conditions (H0) and (H2) hold almost trivially. As for (H1), if we have related states in $K0$ and $K2$, and the state in $K0$ makes a step, it is clear that $K2$ can take a step such that the states remain related. The variables \mathbf{n} and \mathbf{q} are called history variables since they record the history of the execution.

3.3 The Unfolding

The *unfolding* $K^\#$ of a specification K plays a key role in the proofs of semantic completeness in [1, 11] as well as in our semantic completeness result below.

It is defined as follows: $states(K^\#)$ consists of the stutterfree finite initial executions of K . The initial set $init(K^\#)$ consists of the elements $xs \in states(K^\#)$ with $\ell(xs) = 1$. The next-state relation $step(K^\#)$ and the property $prop(K^\#) \subseteq (states(K^\#))^\omega$ are defined by

$$\begin{aligned}
(xs, xt) \in step(K^\#) &\equiv xs = xt \vee xs = init(xt) , \\
vss \in prop(K^\#) &\equiv last^\omega(vss) \in prop(K) .
\end{aligned}$$

It is easy to prove that $K^\#$ is a specification. The function $last : states(K^\#) \rightarrow states(K)$ is a refinement mapping and, hence, a forward simulation $K^\# \rightarrow K$. We are more interested, however, in the other direction. The following result of [1] is not difficult to prove.

Lemma (0). Relation $cvl = cv(last)$ is a forward simulation $K \rightarrow K^\#$. \square

In Sect. 4.3 below, we shall need the following result.

Lemma (1). Let vss be a stutterfree behaviour of $K^\#$. Then $xs = last^\omega(vss)$ is a stutterfree behaviour of K such that vss_i is a prefix of xs for all indices i and that $vss_i \preceq (xs|i+1)$ for all i .

Proof. Firstly, xs is a behaviour of K since vss is a behaviour of $K^\#$. Since vss is stutterfree, there is r with $0 \leq r \leq \infty$ such that $vss_i = init(vss_{i+1})$ for all i with $0 \leq i < r$ and $vss_i = vss_{i+1}$ for all i with $r \leq i < \infty$. It follows that $vss_i = (xs|i+1)$ for all i with $0 \leq i \leq r$ and $vss_i = (xs|r+1)$ for all i with $r \leq i < \infty$. Since each vss_i is stutterfree, list $(xs|r+1)$ is stutterfree. All remaining elements of xs are equal to xs_r . Therefore, xs is a stutterfree infinite list. It also follows that $vss_i \preceq (xs|i+1)$ for all i . \square

3.4 Backward Simulations

It is also possible to prove that one specification simulates (the behaviour of) another by starting arbitrarily far in the future and constructing a corresponding execution by working backwards. An infinite behaviour is then obtained by a variation of König's Lemma. These so-called backward simulations [11] form a relational version of the prophecy variables of [1] and are related to the upward simulations of [4]. We give a variation of Jonsson's version [6], but we give no example since we prefer the eternity extensions introduced in the next section.

Relation F between $states(K)$ and $states(L)$ is defined to be a *backward simulation* from K to L iff

- (P0) Every state y of L with $(x, y) \in F$ and $x \in init(K)$ satisfies $y \in init(L)$.
- (P1) For every pair $(x', y') \in F$ and every x with $(x, x') \in step(K)$, there is y with $(x, y) \in F$ and $(y, y') \in step(L)$.
- (P2) For every behaviour xs of K there are infinitely many indices n for which the set $(xs_n; F)$ is nonempty and finite.
- (P3) Every infinite initial execution ys of L with $(xs, ys) \in F^\omega$ for some $xs \in Beh(K)$ satisfies $ys \in prop(L)$.

An auxiliary variable added to the state space via a backward simulation is called a prophecy variable [1] since it seems to show "prescient" behaviour. In such a case, the relation is called a prophecy relation in [11]. Note that condition (P3) just equals (H2). The term backward simulations is justified by the following soundness result, the proof of which is a easy adaptation of the proof in [6].

Lemma. Every backward simulation F from K to L is a simulation $K \rightarrow L$. \square

A composition of backward simulations need not be a backward simulation. As shown by Jonsson [6], this can be used to construct more general simulations.

3.5 Preservation of Quiescence

The completeness result of the next section needs preservation of quiescence in the sense that the abstract behaviour should be quiescent whenever the concrete behaviour is quiescent. This is formalized as follows.

A finite initial execution of specification K is called *quiescent* iff it becomes a behaviour by extending it with the infinite repetition of its last state. For a behaviour xs of K , we define the set of quiescent numbers $Q_K(xs)$ to consist of the numbers n such that $(xs|n+1)$ is a quiescent execution.

Let K and L be specifications. A simulation $F : K \rightarrow L$ is said to *preserve quiescence* iff, for every $xs \in Beh(K)$, there exists $ys \in Beh(L)$ with $(xs, ys) \in F^\omega$ and $Q_K(xs) \subseteq Q_L(ys)$.

Example. We construct a simulation that does not preserve quiescence. Consider specifications K and L , both with state space $X = \{0, 1, 2\}$, initial set $\{1\}$, and supplementary property $\diamond \square \llbracket \{0\} \rrbracket$. The next-state relations are given by

$$\begin{aligned} step(K) &= 1_X \cup \{(1, 0), (0, 1)\}, \\ step(L) &= 1_X \cup \{(1, 0), (1, 2), (2, 1)\}. \end{aligned}$$

The behaviours of K are infinite lists over $\{0, 1\}$ that start with 1 and contain only finitely many ones. The behaviours of L are finite lists over $\{1, 2\}$ that start and end with 1, followed by infinitely many zeroes. In either case, the quiescent indices are those of the zero elements in the list.

Let relation F on X be the set $F = \{(0, 0), (0, 2), (1, 1)\}$. Relation F is a simulation $K \rightarrow L$. In fact, for every $xs \in Beh(K)$, there is precisely one $ys \in Beh(L)$ with $(xs, ys) \in F^\omega$. If n is the least number with $xs_i = 0$ for all $i \geq n$, then $ys_j = 2$ for all $j < n$ with $xs_j = 0$, and $ys_j = xs_j$ in all other cases. Since xs_j can be zero when ys_j is not, simulation F does not preserve quiescence. \square

It is easy to verify that, if $F : K \rightarrow L$ and $G : L \rightarrow M$ both preserve quiescence, the composition $(F; G) : K \rightarrow M$ also preserves quiescence. Also, if $F : K \rightarrow L$ preserves quiescence and G is a relation between $states(K)$ and $states(L)$ with $F \subseteq G$, then G is a simulation $K \rightarrow L$ that preserves quiescence.

Lemma (2). Let $F : K \rightarrow L$ be a simulation that satisfies condition (H2) of Sect. 3.1, or equivalently (P3) of Sect. 3.4. Then F preserves quiescence.

Proof. Let $xs \in Beh(K)$. Since F is a simulation, there exists $ys \in Beh(L)$ with $(xs, ys) \in F^\omega$. It suffices to prove that $Q_K(xs) \subseteq Q_L(ys)$. Let $n \in Q_K(xs)$. Define xt and yt as the extensions of $(xs|n+1)$ and $(ys|n+1)$ with the infinite repetitions of their last states xs_n and ys_n . Since $n \in Q_K(xs)$, we have $xt \in Beh(K)$. On the other hand, yt is an infinite initial execution of L and $(xt, yt) \in F^\omega$. By (H2) or (P3), yt is a behaviour of L . This implies $n \in Q_L(ys)$. \square

This lemma implies that refinement mappings and forward and backward simulations all preserve quiescence.

4 An Eternity Variable for a Refinement Mapping

We now develop an alternative for prophecy variables or backward simulations that is both simpler and more powerful. Extending the metaphor of history and prophecy variables, they are named eternity variables, since they do not change during execution.

They are simpler than prophecy variables in the sense that, below, both the proof of soundness in Lemma (4) and the proof of completeness in Theorem 1 are simpler than the corresponding proofs for prophecy variables. They are more powerful in the sense that their completeness does not require additional finiteness assumptions. Whether they are simpler to use than prophecy variables is a matter of future research.

The idea is that an eternity variable has an indeterminate constant value, but that the states impose restrictions on this value. A behaviour in which the eternity variable ever has a wrong value is simply discarded. Therefore, in every behaviour, the eternity variable always has a value that satisfies all restrictions of the behaviour.

The specification obtained by adding an eternity variable is called an eternity extension. We introduce eternity extensions and prove their soundness in Sect. 4.1. Section 4.2 contains an example. Completeness of eternity extension is proved in Sect. 4.3.

4.1 Formal Definition of Eternity Extensions

Let K be a specification. Let M be a set of values for an eternity variable m . A binary relation R between $states(K)$ and M is called a *behaviour restriction* of K iff, for every behaviour xs of K , there exists an $m \in M$ with $(xs_i, m) \in R$ for all indices i :

$$(3) \quad xs \in Beh(K) \Rightarrow (\exists m :: (\forall i :: (xs_i, m) \in R)) .$$

If R is a behaviour restriction of K , we define the corresponding *eternity extension* as the specification W given by

$$\begin{aligned} states(W) &= R , \\ init(W) &= \{(x, m) \in R \mid x \in init(K)\} , \\ ((x, m), (x', m')) \in step(W) &\equiv (x, x') \in step(K) \wedge m = m' , \\ ys \in prop(W) &\equiv fst^\omega(ys) \in prop(K) . \end{aligned}$$

It is clear that $step(W)$ is reflexive and that $prop(W)$ is a property. Therefore W is a specification. It is easy to verify that $fst : states(W) \rightarrow states(K)$ is a refinement mapping. The soundness of eternity extensions is expressed by

Lemma (4). Let R be a behaviour restriction. The relation $cvf = cv(fst)$ is a simulation $cvf : K \rightarrow W$ that preserves quiescence.

Proof. Let $xs \in Beh(K)$. We have to construct $ys \in Beh(W)$ with $(xs, ys) \in cvf^\omega$. By (3), we can choose m with $(xs_i, m) \in R$ for all i . Then we define $ys_i = (xs_i, m)$. A trivial verification shows that the list ys constructed in this way is a behaviour of W with $(xs, ys) \in cvf^\omega$. Preservation of quiescence follows from Lemma (2). \square

In this construction, we fully exploit the ability to consider specifications that are not machine closed. Initial executions of W that cannot be extended to behaviours of W are simply discarded.

4.2 An Example of an Eternity Extension

We use an eternity extension to show that specification $K0$ implements $K1$ in 2.3. The starting point is the forward simulation $F_{0,2} : K0 \rightarrow K2$ of Sect. 3.2. One easily verifies that specification $K2$ preserves the invariant $\square[j = q[n]]$.

We now extend $K2$ with an eternity variable \mathfrak{m} , which is an infinite array of natural numbers with the behaviour restriction

$$R : j \leq \mathfrak{m}[n] \wedge (\forall i : 0 \leq i < n : \mathfrak{m}[i] = q[i]) .$$

We have to verify that every execution of $K2$ allows a value for \mathfrak{m} that satisfies condition R . So, let xs be an arbitrary execution of $K2$. Since j jumps back infinitely often in xs , the value of n tends to infinity. This implies that $q[i]$ is eventually constant for every index i . We can therefore define function $m : \mathbb{N} \rightarrow \mathbb{N}$ by $\diamond\square[m(i) = q[i]]$ for all $i \in \mathbb{N}$. It follows that $\square[i < n \Rightarrow m(i) = q[i]]$ for all i . Since $q[i]$ is incremented only, it also follows that $\square[q[i] \leq m(i)]$ for all i . This implies $\square[j \leq m(n)]$ because of the invariant $\square[j = q[n]]$. This proves that m is a value for \mathfrak{m} that satisfies R .

Let $K3$ be the resulting eternity extension and $F_{2,3} : K2 \rightarrow K3$ be the simulation induced by Lemma (4). The next-state relation $step(K3)$ satisfies

$$\begin{aligned} ((j, n, q, m), (j', n', q', m')) \in step(K3) &\equiv \\ m = m' \wedge ((j, n, q), (j', n', q')) \in step(K2) & . \end{aligned}$$

Let function $f_{3,1} : states(K3) \rightarrow states(K1)$ be defined by

$$f_{3,1}(j, n, q, m) = (j, (j = 0 ? 0 : m[n])) ,$$

where $(_ ? _ : _)$ stands for a conditional expression as in the language C . We verify that $f_{3,1}$ is a refinement mapping. Since $f_{3,1}(0, 0, 0, m) = (0, 0)$, we have $f_{3,1}(init(K3)) \subseteq init(K1)$. We now show that a step of $K3$ is mapped to a step of $K1$. By convention, this holds for a stuttering step. A nonstuttering step that starts with $j = 0$ increments j to 1. The $f_{3,1}$ -images make a step from $(0, 0)$ to $(1, r)$ for some positive r . This is in accordance with $K1$. A step of $K3$ that increments a positive j has the precondition $j < \mathfrak{m}[n]$ because of R ; therefore, the $f_{3,1}$ -images make a $K1$ -step. A backjumping step of $K3$ increments n and has therefore precondition $j = q[n] = \mathfrak{m}[n]$. Again, the $f_{3,1}$ -images make a $K1$ -step. This proves $f_{3,1}(step(K3)) \subseteq step(K1)$. It is easy to see that $f_{3,1}^\omega(prop(K3)) \subseteq prop(K1)$. This shows that $f_{3,1}$ is a refinement mapping from $K3$ to $K1$.

We thus have a composed simulation $G = (F_{0,2}; F_{2,3}; f_{3,1}) : K0 \rightarrow K1$. One verifies that $(j, (k, m)) \in G$ implies $j = k$. It follows that relation $F_{0,1}$ of Sect. 2.3 satisfies $G \subseteq F_{0,1}$. Therefore, $F_{0,1}$ is a simulation $K0 \rightarrow K1$. This shows that an eternity extension can be used to prove that $F_{0,1}$ is a simulation $K0 \rightarrow K1$.

Remark. Once R is chosen, the problem is to find a value for m in (3). Above we constructed m by means of a history variable (\mathbf{q}) that approximates m . Since condition (3) is equivalent to the property that $\bigcap_i (xs_i; R)$ is nonempty for every behaviour xs of K , one can always approximate m by a set-valued history variable with values of the form $\bigcap_{i < r} (xs_i; R)$ for a state with a history xs of length r . We leave the investigation of this idea to future research.

4.3 Completeness of Eternity Extensions

The combination of unfoldings, eternity extensions and refinement mappings is semantically complete in the following sense.

Theorem 1. Consider a simulation $F : K \rightarrow L$ that preserves quiescence. The unfolding $cvl : K \rightarrow K^\#$ has an eternity extension $cvf : K^\# \rightarrow W$ and a refinement mapping $g : W \rightarrow L$ such that $(cvl; cvf; g) \subseteq F$.

Proof. Firstly, we have a simulation $cvl : K \rightarrow K^\#$ by Lemma (0).

We use an eternity variable in the set $Beh(L)$. We define relation R between $states(K^\#)$ and $Beh(L)$ to consist of the pairs (xs, ys) such that, for some $xt \in Beh(K)$, it holds that

$$xs = (xt | \ell(xs)) \wedge (xt, ys) \in F^\omega \wedge Q_K(xt) \subseteq Q_L(ys) .$$

We show that R is a behaviour restriction by verifying condition (3). Let vss be a behaviour of $K^\#$. In order to verify (3), we may assume that vss is stutterfree. By Lemma (1), we have that $xt = last^\omega(vss)$ is a behaviour of K such that vss_i is a prefix of xt for all indices i . Since $F : K \rightarrow L$ preserves quiescence, specification L has a behaviour ys with $(xt, ys) \in F^\omega$ and $Q_K(xt) \subseteq Q_L(ys)$. For every i , list vss_i is a prefix of xt . This implies that $(vss_i, ys) \in R$ for all $i \in \mathbb{N}$. Taking $m = ys$, this proves that R satisfies condition (3) and is therefore a behaviour restriction.

Let W be the R -eternity extension of $K^\#$. By Lemma (4), we have a simulation $cvf : K^\# \rightarrow W$. Define $g : W \rightarrow states(L)$ by

$$g(xs, ys) = last(ys | \ell(xs)) .$$

We show that g is a refinement mapping $states(W) \rightarrow states(L)$. Firstly, let $w \in init(W)$. Then w is of the form $w = (xs, ys)$ with $\ell(xs) = 1$ and ys_0 is initial in L . This shows $g(w) \in init(L)$. In every nonstuttering step in W , the length of xs is incremented with 1 and then we have $(ys_n, ys_{n+1}) \in step(L)$. Therefore, function g maps steps of W to steps of L .

In order to show that g maps every behaviour of W to a behaviour of L , it suffices to show that $g^\omega(ws) \in prop(L)$ for every stutterfree behaviour of W .

So, let ws be a stutterfree behaviour of W . Since $ws \in \text{prop}(W)$, we have $us = \text{fst}^\omega(ws) \in \text{prop}(K^\#)$. Since ws is a behaviour of W , its elements have a common second component $ys \in \text{Beh}(L)$. Now $(us_n, us_{n+1}) \in \text{step}(K^\#)$ and $ws_n = (us_n, ys)$ for all n . We have $g(ws_n) = \text{last}(ys|\ell(us_n))$. Since ws is stutterfree, us is stutterfree. There are two possibilities. Either $\ell(us_n) = n + 1$ for all n , or there exist a number m , such that $\ell(us_n) = \min(n, m) + 1$ for all n . In the first case, we have $g^\omega(ws) = ys \in \text{prop}(L)$. In the second case, $g^\omega(ws)$ equals the infinite list obtained by extending $(ys|m + 1)$ with the infinite repetition of its last element ys_m . Therefore, $g^\omega(ws) \in \text{prop}(L)$ would follow from $m \in Q_L(ys)$. Since $ws_m \in R$, there exists a behaviour ut of K such that $us_m = (ut|m + 1)$ and $(ut, ys) \in F^\omega$ and $Q_K(ut) \subseteq Q_L(ys)$. On the other hand, $\text{last}^\omega(us) \in \text{prop}(K)$ equals the infinite list obtained by extending us_m with the infinite repetition of its last element. This implies that $m \in Q_K(ut) \subseteq Q_L(ys)$.

It remains to prove $(\text{cvl}; \text{cvf}; g) \subseteq F$. Let (x, y) be in the lefthand relation. By the definition of $(\text{cvl}; \text{cvf}; g)$, there exist $xs \in \text{states}(K^\#)$ and $w \in \text{states}(W)$ with $x = \text{last}(xs)$ and $xs = \text{fst}(w)$ and $g(w) = y$. By the definition of W , we can choose $ys \in \text{Beh}(L)$ with $w = (xs, ys)$. Let $n = \ell(xs)$. Since $(xs, ys) \in R$, we have $(xs, ys|n) \in F^*$. It follows that $y = g(w) = ys_{n-1}$ and hence $(x, y) = (xs_{n-1}, ys_{n-1}) \in F$. This proves the inclusion. \square

Remark. Conversely, it is easy to verify that every simulation F that satisfies the consequent of Theorem 1 preserves quiescence.

5 Concluding remarks

We have introduced simulations of specifications to unify all cases where an implementation relation can be established. This unifies refinement mappings, history variables or forward simulations, and prophecy variables or backward simulations, and refinement of atomicity as in Lipton's Theorem [3, 10].

We have introduced eternity extensions as variations of prophecy variables and backward simulations. We have proved semantic completeness: every simulation that preserves quiescence can be factored as an extension with history variables and eternity variables followed by a refinement mapping. The restrictive assumptions machine-closedness and finite invisible nondeterminism, as needed for completeness of prophecy variables or forward-backward simulations in [1, 11] are superfluous when eternity variables are allowed.

The theory has two versions. In the strict version described here, we allow the concrete behaviours to take more but not less computation steps than the abstract behaviours. This is done by allowing additional stutterings to the abstract specifications. The strict theory is also the simpler one. It results in a finer hierarchy of specifications. The stuttering version of the theory [5] is completely in accordance with the setting of [1]. Here, the eternity extension needs an additional counter to regulate the stutterings of the concrete specification.

Acknowledgements. I would like to thank H.W. de Haan, G.R. Renardel, and the MPC referees for their helpful suggestions and comments that have led to significant improvements in the presentation.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. *Theoretical Computer Science* **82** (1991) 253–284
2. Abadi, M., Lamport, L.: Conjoining specifications. *ACM Transactions on Programming Languages and Systems* **17** (1995) 507–534.
3. Cohen, E., Lamport, L.: Reduction in TLA. In: Sangiorgi, D., Simone, R. de (eds.): *CONCUR '98*. Springer V. 1998 (LNCS 1466), pp. 317–331.
4. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.): *ESOP86* pp. 187–196. Springer Verlag, 1986 (LNCS 213).
5. Hesselink, W.H.: Eternity variables to prove simulation of specifications (draft). www.cs.rug.nl/~wim/pub/whh275.pdf
6. Jonsson, B.: Simulations between specifications of distributed systems. In: Baeten, J.C.M., Groote, J.F. (eds.): *CONCUR '91*. Springer V. 1991 (LNCS 527), pp. 346–360.
7. Jonsson, B., Pnueli, A., Rump, C.: Proving refinement using transduction. *Distributed Computing* **12** (1999) 129–149.
8. Lamport, L.: Critique of the Lake Arrowhead three. *Distributed Computing* **6** (1992) 65–71.
9. Lamport, L.: The temporal logic of actions. *ACM Trans. on Programming Languages and Systems* **16** (1994) 872–923.
10. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. *Communications of the ACM* **18** (1975) 717–721.
11. Lynch, N., Vaandrager, F.: Forward and backward simulations, Part I: untimed systems. *Information and Computation* **121** (1995) 214–233.
12. Milner, R.: An algebraic definition of simulation between programs. In: *Proc. 2nd Int. Joint Conf. on Artificial Intelligence*. British Comp. Soc. 1971. Pages 481–489.