Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 214 (2008) 3–23

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# Simulation Refinement for Concurrency Verification

Wim H. Hesselink[1] ,[2]

*Dept. of Mathematics and Computing Science*
*University of Groningen*
*P.O.Box 407, 9700 AK Groningen, The Netherlands*

**Abstract**

In recent years, we extended the theory of Abadi and Lamport (1991) on the existence of refinement mappings. The present paper gives an overview of several extensions of the theory and of a number of recent applications to practical verifications. It concludes with a sketch of the results on semantic completeness, and a discussion of the relationship between semantic completeness and methodological convenience.

*Keywords:* refinement, simulation, atomicity, verification, semantic completeness.

## 1 Introduction

The aim of this paper is to present an overview of the methods and results we developed in several verifications of concurrent algorithms during the last years.

The design of an algorithm ideally starts with the specification and proceeds via a number of motivated refinement steps to conclude with an implementation. In the field of concurrency, however, this ideal is more often proclaimed than executed. In our practice, the designer starts with an idea for a concurrent algorithm, and while trying to develop and prove the algorithm, they change the algorithm until it is concrete enough and has an

[1] Email: w.h.hesselink@rug.nl

[2] Web: www.cs.rug.nl/~wim

doi:10.1016/j.entcs.2008.06.002

associated proof. Just as often, at the other side of the spectrum, there may be an algorithm with or without any proof, and one may want to decide its correctness.

In other words, for concurrent algorithms, I do not believe in refinement as the one and only design methodology. Refinement is for me the relation between specification and implementation that is to be verified, and, as such, fundamental to the processes of design and verification of concurrent algorithms.

Indeed, one of the first problems with concurrent algorithms is the specification. They are usually reactive programs: they start in a rather blank initial state, interact in meaningful ways with their environment, and when they terminate it is often by mishap. In particular they cannot be specified by preconditions and postconditions only.

The starting point of refinement theory is that programs and algorithms are nothing but specifications that happen to be executable. The next point is to define and construct implementation and simulation relations between specifications, of various types and for various purposes. Our formalism is a variation of [1].

**Overview**. Section 2 gives the basic formalism with specifications, executions, behaviours, invariants, strict implementations and simulations, refinement functions and forward simulations. In section 3, we discuss a range of (non)atomicity conditions, to be used in section 4 where we verify a lock-free implementation of a row of atomically modifiable variables.

In section 5, we introduce prophecies. Three formalizations are treated: backward simulations, eternity extensions, and episodic simulations with Lipton's simulation as a special case. We give an exampe of an eternity extension, which shows an invariant that cannot be proved by induction from the initial states. In section 6, we discuss stutterings and introduce nonstrict implementations and simulations.

In section 7, we present the result of semantic completeness of our simulation concepts [16] and sketch the tension between semantic completeness and methodological convenience. We conclude in section 8 with remarks that link our efforts to the use of the theorem provers NQTHM and PVS.

## 2 Specifications and strict simulations

A *specification* [1] is defined to be a tuple $K = (X, X_0, N, P)$ where $X$ is a set, $X_0$ is a subset of $X$, $N$ a reflexive relation on $X$, and $P$ is a property over $X$ (see section 6.1). The set $X$ is called the *state space*, its elements are called *states*, the elements of $X_0$ are called *initial states*. Relation $N$ is called the *next-state* relation. The set $P$ is called the *supplementary* property.

An *initial execution* of $K$ is defined to be a finite or infinite sequence $xs$ over $X$ with $xs(0) \in X_0$ and such that every pair of consecutive elements belongs to $N$. A *behaviour* of $K$ is an infinite initial execution $xs$ of $K$ with $xs \in P$. The requirements that relation $N$ is reflexive and that set $P$ is a property, are imposed to allow stuttering: if $xs$ is a behaviour of $K$, any sequence $ys$ obtained from $xs$ by repeating elements of $xs$ or by removing subsequent duplicates must also be a behaviour of $K$.

We write $Beh(K)$ to denote the set of behaviours of $K$. By definition, we have that $Beh(K) = [\![ X_0 ]\!] \cap \Box [\![ N ]\!]_2 \cap P$. It follows that $Beh(K)$ is a property.

Specifications are only useful by what we can observe. We therefore assume that our specifications are *visible*, i.e., have a given observation function $obs$ from the state space $X$ to some set of observables. In principle, we are therefore primarily interested in the *observed behaviours*, the sequences $obs \circ xs$ where $xs$ ranges over the behaviours.

A subset $J \subseteq X$ is called an *invariant* if $xs(n) \in J$ for every behaviour $xs$ of $K$ and every $n \in \mathbb{N}$. A subset $J$ is called *inductive* if $X_0 \subseteq J$ and $y \in J$ for every pair $(x, y) \in N$ with $x \in J$. It is well-known and easy to verify that every inductive set is invariant. It is not true that an invariant necessarily contains all reachable states. See section 5.3 for a counterexample.

The components of specification $K = (X, X_0, N, P)$ are denoted $states(K) = X$, $start(K) = X_0$, $step(K) = N$ and $prop(K) = P$.

## *2.1 Strict implementations and simulations*

A visible specification $K$ is said to *strictly implement* a visible specification $L$ if every observed behaviour of $K$ is an observed behaviour of $L$. In order to prove such a thing, however, we must be able to look behind the scenes. We therefore introduce simulation relations.

A relation $F$ between the state spaces of specifications $K$ and $L$ is called a *strict simulation* from specification $K$ to specification $L$ (notation $F : K \multimap L$) if, for every $xs \in Beh(K)$, there exists $ys \in Beh(L)$ with $(xs, ys) \in F_\omega$ where

$$(xs, ys) \in F_\omega \quad \equiv \quad (\forall\, i : (xs(i), ys(i)) \in F) \ .$$

If $K$ and $L$ are visible, a relation $F$ between the state spaces of specifications $K$ and $L$ is called *nondisturbing* if $obs(x) = obs(y)$ for all pairs $(x, y) \in F$.

It is easy to see that $K$ strictly implements $L$ if and only if there is a nondisturbing simulation $F : K \multimap L$. We can therefore use simulations to prove implementaton relations, and we are mainly interested in nondisturbing simulations. Mostly, however, we take nondisturbingness for granted, and even ignore all visibility aspects.

### 2.2 Refinement mappings and forward simulations

A central aim in the methodology of concurrency verification is to eliminate behaviours as much as possible from our considerations and rather argue about states and the next state relation. It is therefore important to give criteria for simulations which are as much as possible in terms of states and the next state relation.

If $K$ and $L$ are specifications, a function $f : \mathit{states}(K) \to \mathit{states}(L)$ is called a *refinement mapping* [1] from $K$ to $L$ iff
(f0) $f(x) \in \mathit{start}(L)$ for every $x \in \mathit{start}(K)$;
(f1) $(f(x), f(x')) \in \mathit{step}(L)$ for every pair $(x, x') \in \mathit{step}(K)$;
(f2) $f \circ \mathit{xs} \in \mathit{prop}(L)$ for every $\mathit{xs} \in \mathit{Beh}(K)$.

In practice, condition (f1) is often stronger than necessary and convenient. Let us therefore define a function $f$ to be a *refinement function* iff it satisfies conditions (f0), (f1f) and (f2), where (f1f) is given by
(f1f) $K$ has an invariant $J$ such that $(f(x), f(x')) \in \mathit{step}(L)$ for every pair $(x, x') \in \mathit{step}(K) \cap (J \times J)$.

Every refinement mapping is a refinement function since we can use $\mathit{states}(K)$ itself as an invariant.

It is well-known that refinement functions are not enough to prove all simulation relations. A natural way to prove that one specification simulates another is by starting at the beginning and constructing the corresponding behaviour in the other specification inductively. This idea is formalized in forward simulations [7,22,23], defined as follows.

A relation $F$ between $\mathit{states}(K)$ and $\mathit{states}(L)$ is called a *forward simulation* from specification $K$ to specification $L$ iff

(F0) For every $x \in \mathit{start}(K)$, there is $y \in \mathit{start}(L)$ with $(x, y) \in F$.
(F1) For every pair $(x, y) \in F$ and every $x'$ with $(x, x') \in \mathit{step}(K)$, there is $y'$ with $(y, y') \in \mathit{step}(L)$ and $(x', y') \in F$.
(F2) For every initial execution $\mathit{ys}$ of $L$ and every behaviour $\mathit{xs}$ of $K$, we have that $(\mathit{xs}, \mathit{ys}) \in F_\omega$ implies $\mathit{ys} \in \mathit{prop}(L)$.

The following well-known lemma justifies the nomenclature and shows the relationships between refinement functions, simulations and forward simulations.

**Lemma 2.1** *(a) Let $f : \mathit{states}(K) \to \mathit{states}(L)$ be a refinement function from a specification $K$ to a specification $L$, say with invariant $J$. Then the graph $\{(x, y) \mid x \in J \wedge f(x) = y\}$ is a forward simulation from $K$ to $L$.*
*(b) Let $F$ be a forward simulation from $K$ to $L$. Then $F$ is a strict simulation $F : K \rightarrowtriangle L$.*

In view of this Lemma, we use the notation $f : K \multimapdotinv L$ also for a refinement function from $K$ to $L$.

## 3 Safeness, atomicity, and atomic modification

In order to illustrate the refinement concepts, let me recall (or introduce) some concepts of safeness and atomicity of shared variables.

The minimal correctness assumption for a *shared variable* is that the correctness of its read and write operations is guaranteed if and only if these operations are performed under mutual exclusion. In other words, chaos can result whenever two processes concurrently access the variable. Let us call such variables *unsafe*.

Following [19], a shared variable is called *safe* if read operations that overlap with a single write operation are allowed in the sense that they return legitimate values of the domain of the variable. Write operations need to be performed under mutual exclusion because concurrent write operations may give chaos.

A shared variable is called *atomic* iff read and write operations behave as if they never overlap but always occur in some total order that refines the *precedence* order (an operation *precedes* another iff it terminates before the other starts).

A shared variable is said to be *atomically modifiable* if the operations to inspect and modify it behave as if they never overlap but always occur in some total order that refines the *precedence* order. Atomically modifiable variables are stronger than usually considered, but for some purposes they are useful, and we shall give a method to implement rows of them.

We introduced formal models for unsafe and safe variables in [9,15]. Here, we ignore such details but concentrate on the refinement relations.

By convention, shared variables are written in typewriter font and private variables are written slanted.

We use the following general format for atomic modification. The actions to be performed are described by

$$C(\textbf{in}\ arg : Arg,\ \textbf{ref}\ \mathtt{x} : Node,\ \textbf{out}\ result : Item)\ .$$

In command $C$, parameter $\mathtt{x}$ is a reference variable, $arg$ is an input variable, and $result$ is an output variable. We express the semantics of command $C$ by the four place predicate $C(arg, \mathtt{x}, \mathtt{x}^+, result^+)$ where $\mathtt{x}^+$ and $result^+$ stand for the values after execution.

A CAS variable (compare and swap) is a special case of an atomically modifiable variable. It is a shared variable, say $\mathtt{x}$, that can be read and written atomically, but it also supports the conditional update:

```
$\mathtt{CAS}(\mathtt{x}, u, v) : \mathbb{B} =$
    $\langle$ **if** $\mathtt{x} = u$ **then** $\mathtt{x} := v$ ; **return** *true*
      **else** **return** *false* **end** $\rangle$ ,
```

where $u$ and $v$ are private variables or expressions of the acting process. The angular brackets $\langle$ and $\rangle$ are used to indicate atomicity.

A CAS variable can be used for a lock-free implementation of an atomically modifiable variable in the following way:

```
**repeat** $u := \mathtt{x}$ ; $v := u$ ; $C(arg, v, result)$ ;
**until** $\mathtt{CAS}(\mathtt{x}, u, v)$ .
```

Note that the correctness of this implementation is not threatened by the ABA phenomenon that $\mathtt{x} \neq u$ can happen between the assignment $u := \mathtt{x}$ and a subsequent succeeding CAS.

In other applications, the ABA phenomenon can be a problem. This problem is avoided in the load-linked, store-conditional primitive LL/SC, which for a shared variable x is described as follows. The variable x gets a field *links* of type set of process.

```
$\mathtt{LL}(\mathtt{x})$    =   $\langle$ *add self to* $\mathtt{x}.links$ ; **return** x $\rangle$ .
$\mathtt{SC}(\mathtt{x}, v)$  =   $\langle$ **if** $self \notin \mathtt{x}.links$ **then** **return** *false*
                  **else** $\mathtt{x}.links := \emptyset$ ; $\mathtt{x} := v$ ; **return** *true* **end** $\rangle$ .
```

Here *self* is the process identifier of the acting process. LL/SC can be used for a lock-free implementation of atomic modification in the following way.

```
**repeat** $v := \mathtt{LL}(\mathtt{x})$ ; $C(arg, v, result)$ ;
**until** $\mathtt{SC}(\mathtt{x}, v)$ .
```

## 4 Lock-free atomic modification

When CAS variables or LL/SC variables are offered by the concurrency platform, they have simple types like *Integer*. It is therefore important to implement atomically modifiable variables of arbitrary types by means of CAS or LL/SC variables of simple types. Alternatively, one may look for wait-free implementations of LL/SC variables by means of CAS or LL/SC variables of simple types, e.g. [17].

In [5,6], we considered the problem of implementing a row of atomically modifiable variables of an arbitrary type by means of safe variables of the same type. Given are $N$ processes and $M$ shared variables. Every process needs repeatedly to inspect and modify some variable according to some given argument. Logically the actions on the variables must be independent (done under mutual exclusion), but a lock-free implementation is asked for. We will concentrate on the specification, and the implementation by means of simple LL/SC variables cf. [5]. We refer to [6] for an implementation with simple CAS variables. Actually, in both cases, we solved a slighly different version, which was more suitable to the intended application in [4].

### 4.1 Specifying a row of atomically modifiable variables

As announced, there are $M$ shared variables of type $Node$. Each process has a private variable $k$ that points into the array of nodes, a program counter $pc$, and variables to hold arguments and results. We thus declare:

$$\begin{array}{l} \textbf{var}\ \texttt{node} : Node[M]\ ; \\ \textbf{privar}\ k, pc : \mathbb{N},\ arg : Arg,\ result, out : Item\ . \end{array}$$

For every process $p$, its environment provides the calls and the arguments, and inspects the results according to

$$\begin{array}{llcl} env.p : & [\!]\ \ pc = 0 & \rightarrow & \textbf{choose}\ k < M\ ;\ \textbf{choose}\ arg\ ;\ pc := 10\ . \\ & [\!]\ \ pc = 1 & \rightarrow & out := result\ ;\ pc := 0\ . \end{array}$$

The system, located at 10, is specified to perform an atomic modification by

$$\begin{array}{llcl} sys.p : & [\!]\ \ pc = 10 & \rightarrow & C(arg, \texttt{node}[k], result)\ ;\ pc := 1\ . \end{array}$$

The progress condition required is that, whenever the system as a whole is enabled, eventually it will do something:

$$\Box((\exists\, q : pc.q = 10)\ \Rightarrow\ \Diamond(\exists\, q : pc.q = 10 \wedge pc^{+}.q = 1))\ .$$

The system is not allowed to modify $arg$ and $out$. The only observable variables are $out.p$ for all $p$. In other words, the observation function removes all other variables. The types $Node$, $Arg$, $Item$ and command $C$ are parameters of the problem. In particular, command $C$ can be used to observe the values of array `node` via $result$.

### 4.2 An LL/SC implementation by means of safe variables

We turn to the implementation [5] with LL/SC variables of integer type. The main idea is a formalization of an idea of Herlihy [8] to extend the array of nodes with nodes that can be kept private for the processes, and to model the array of the specification via indirection.

$$\begin{array}{l} \textbf{var}\ \texttt{ar} : \texttt{safe}\ \ Node[M+N]\ , \\ \quad\ \texttt{indir} : \texttt{LL/CS}\ \ \mathbb{N}[M]\ , \end{array}$$

with the intention that

(1) $\forall\, i : i < M \;\Rightarrow\; \texttt{ar}[\texttt{indir}[i]] = \texttt{node}[i]$ .

Array ar is declared to consist of *safe* shared variables. Therefore, concurrent calls of $C(u, \texttt{ar}[j], v)$ with the same $j$ are not allowed, but other processes may concurrently read $\texttt{ar}[j]$ although this may give nondeterministic results.

The implementation uses some additional private variables

$$\textbf{privar } mi, mp : \mathbb{N},\; tmp : Item \;.$$

with the intention that $mi.q$ is a local copy of $\texttt{indir}[k.q]$ and that $mp.q$ is an index in array ar that is secure against interference. More precisely, we intend the invariants that all values of indir and $mp$ are distinct:

*Aq0* : $\texttt{indir}[i] = \texttt{indir}[k] \;\Rightarrow\; i = k$ ,
*Aq1* : $\texttt{indir}[i] \neq mp.q$ ,
*Aq2* : $mp.q = mp.r \;\Rightarrow\; q = r$ .

We therefore propose the initial condition:

$$(\forall\, i : i < M \;\Rightarrow\; \texttt{indir}[i] = i)$$
$$\wedge\;\; (\forall\, q : q < N \;\Rightarrow\; pc.q = 0 \;\wedge\; mp.q = M + q) \;,$$

where we assume that the process identifiers $q$ are natural numbers $< N$. We propose to implement $sys.p$ by the unbounded loop:

```
10:    $mi := \texttt{LL}(\texttt{indir}[k])$ ;
       $\texttt{ar}[mp] := \texttt{ar}[mi]$ ;
       $C(arg, \texttt{ar}[mp], tmp)$ ;
       if $\texttt{SC}(\texttt{indir}[k], mp)$ then $mp := mi$ ; $result := tmp$ ; goto 1
       else goto 10 end .
```

The private variable $tmp$ is superfluous, but in some sense convenient for the proof. We come back to this.

Before dealing with the correctness of this algorithm, we need to model the assumption that the elements of array ar are not more than safe. This means that during a modification of $\texttt{ar}[mp]$, writing by some other process leads to chaos, while reading may return an arbitrary value. We model this by means of a boolean array $\texttt{wr}[M+N]$, initially false, with the intention that $\texttt{wr}[j]$ indicates that some process is writing $\texttt{ar}[j]$. We introduce a location 11 where the acting process sets the flag $\texttt{wr}(mp)$ to indicate that it will be writing there. If it is set already, chaos results. In the next instruction, reading of $\texttt{ar}[mi]$ gives a nondeterministic result when $\texttt{wr}[mi]$ holds. The flag at $mp$ is reset after command $C(arg, \texttt{ar}[mp], tmp)$. We thus obtain:

```
10:    $mi := \texttt{LL}(\texttt{indir}[k])$ ;
11:    if $\texttt{wr}[mp]$ then CHAOS else $\texttt{wr}[mp] := true$ end ;
12:    if $\texttt{wr}[mi]$ then choose $\texttt{ar}[mp]$ else $\texttt{ar}[mp] := \texttt{ar}[mi]$ end ;
```

```
13:     $C(arg, \texttt{ar}[mp], tmp)$ ; $\texttt{wr}[mp] := false$ ;
14:     if $\texttt{SC}(\texttt{indir}[k], mp)$ then $mp := mi$ ; $result := tmp$ ; goto 1
        else goto 10 end .
```

While trying to prove the invariants *Aq0*, *Aq1*, *Aq2*, we are forced to invent three other related invariants. Firstly, to prove that command 11 never gives `CHAOS` forces us to postulate the invariant:

$$Aq3: \quad \texttt{wr}[mp.q] \;\Rightarrow\; pc.q \in \{12, 13\} \ .$$

Let us write $\texttt{set}[i] = \texttt{indir}[i].links$. Note that initially $\texttt{set}[i] = \emptyset$ for all $i$. The intention that $mi.q$ be a local copy of $\texttt{indir}[k.q]$ is strengthened to the invariant:

$$Aq4: \quad q \in \texttt{set}[i] \;\Rightarrow\; mi.q = \texttt{indir}[i] \;\wedge\; i = k.q \;\wedge\; pc.q > 10 \ .$$

The proofs of these invariants completely concentrate on commands 11 and 14. *Aq0* is preserved because of *Aq1* and *Aq3*. *Aq1* is preserved because of *Aq0*, *Aq2*, *Aq3*, and *Aq4*. *Aq2* is preserved because of *Aq1*, *Aq3*, and *Aq4*. *Aq3* is preserved because of *Aq1*, *Aq2*, *Aq4*, and the new predicate

$$Aq5: \quad \texttt{wr}[j] \;\Rightarrow\; (\exists\, q : j = mp.q) \ .$$

*Aq4* and *Aq5* are preserved because of *Aq3*. All this only serves to preclude unwanted interferences.

The purpose of the computation is in the values computed in 13 and transferred in 12 and 14. We expect that the invariants for the values will be forced upon us by the proof of the refinement function. In the concrete algorithm, progress only occurs when the `SC` in 14 succeeds. We therefore take this event as the linearization point. In view of formula (1), we propose the refinement function

$$\begin{aligned} fca(x) = (\#\;\; & out := x.out \ , \\ & arg := x.arg \ , \\ & result := x.result \ , \\ & k := x.k \ , \\ & \texttt{node} := x.\texttt{ar} \circ x.\texttt{indir} \ , \\ & pc := \lambda\, q : ipc(x.pc.q) \\ \#) \ , & \end{aligned}$$

where $ipc(k) = k$ for $k < 10$ and otherwise $ipc(k) = 10$.

When proving with PVS that this is indeed a refinement function, we find that we only need the invariants *Aq1*, *Aq3*, and the additional invariant:

$$\begin{aligned} Aq6: \quad & pc.q = 14 \;\wedge\; q \in \texttt{set}(k.q) \\ & \Rightarrow\; C(arg.q, \texttt{ar}[\texttt{indir}[k.q]], \texttt{ar}[mp.q], tmp.q) \ , \end{aligned}$$

where predicate $C(u, \mathtt{x}, \mathtt{y}, v)$ means that the call $C(u, \mathtt{x}, res)$ may result in $\mathtt{x} = \mathtt{y}$ and $res = v$.

When proving the invariance of *Aq6*, it turns out that we also need the invariant

$$Aq7: \qquad pc.q = 13 \ \wedge \ q \in \mathtt{set}(k.q) \quad \Rightarrow \quad \mathtt{ar}[\mathtt{indir}[k.q]] = \mathtt{ar}[mp.q] \ .$$

The proof of *Aq6* needs *Aq1*, *Aq2*, *Aq3*, and *Aq7*. The proof of *Aq7* needs *Aq1*, *Aq2*, *Aq3*, *Aq4*, and *Aq5*.

We come back to the private variable *tmp* used in lines 13 and 14. The programmer can just as well replace *tmp* by *result* in 13 and omit the assignment to *result* in 14. Then, however, the refinement function *fca* is no longer correct because *result* changes in the line 13, which corresponds to a skip statement in the abstract specification. The easiest solution would be to introduce a history variable, say *prev*, that remembers the previous value of *result*, and to use *prev* in the refinement function. The introduction of *prev* would need a forward simulation.

## 4.3 Progress

According to the specification, progress of the abstract algorithm can be expressed by $\Box(A \ \Rightarrow \ \Diamond B)$ where $A \equiv (\exists\, q : pc.q \geq 10)$ and $B$ is the next state relation

$$B \quad \equiv \quad (\exists\, q : pc.q \geq 10 \ \wedge \ pc^+.q = 1) \ .$$

The expressions here are chosen in such a way that they can also be used on the concrete specification. We therefore have to show that the implementation does $B$ steps often enough. In [12], we introduced the "leads to" relation $Q \ o\!\!\rightarrow_B R$, for state predicates $Q$ and $R$ and a next state relation $B$, to mean $\Box(Q \ \Rightarrow \ \Diamond(R \vee B))$, i.e., if a state in any behaviour satisfies $Q \wedge \neg R$ the behaviour will do a $B$ step or have a later $R$ state. Relation $o\!\!\rightarrow_B$ is reflexive and transitive.

For our concrete algorithm, we claim:

$$\begin{array}{ll} & pc.q > 10 \ \wedge \ q \notin \mathtt{set}[k.q] \\ o\!\!\rightarrow_B & pc.q = 10 \\ o\!\!\rightarrow_B & pc.q = 11 \ \wedge \ q \in \mathtt{set}[k.q] \\ o\!\!\rightarrow_B & pc.q = 12 \ \wedge \ q \in \mathtt{set}[k.q] \\ o\!\!\rightarrow_B & pc.q = 13 \ \wedge \ q \in \mathtt{set}[k.q] \\ o\!\!\rightarrow_B & pc.q = 14 \ \wedge \ q \in \mathtt{set}[k.q] \\ o\!\!\rightarrow_B & \mathit{false} \ . \end{array}$$

This formula implies $\Box(A \ \Rightarrow \ \Diamond B)$ and thus proves progress.

The methods and results of [12] seem to be strong enough to verify progress with the proof assistant PVS, but we did not do this.

## 5 Prophecies

Sometimes, when matching a concrete specification with the abstract specification it is supposed to implement, the verifier feels that the abstract specification does a certain nondeterministic step earlier than the concrete specification. In order to get a simulation between the two, they may then feel forced to extend the concrete specification with a ghost variable the value of which is guessed nondeterministically. This is called a *prophecy*. We give an example in section 5.3 below.

### 5.1 Backward simulations

Prophecies can be formalized with prophecy variables [1] or (more or less equivalently) backward simulations [18,22]. Let us use the following definition [11]. Relation $F$ between $\mathit{states}(K)$ and $\mathit{states}(L)$ is defined to be a *backward simulation* from $K$ to $L$ if
(B0) Every pair $(x, y) \in F$ with $x \in \mathit{start}(K)$ satisfies $y \in \mathit{start}(L)$.
(B1) For every pair $(x, x') \in \mathit{step}(K)$ and every $y'$ with $(x'y') \in F$, there is $y$ with $(x, y) \in F$ and $(y, y') \in \mathit{step}(K)$.
(B2) For every behaviour $\mathit{xs}$ of $K$ there are infinitely many indices $n$ for which the set $\{y \mid (\mathit{xs}(n), y) \in F\}$ is nonempty and finite.
(B3) Requirement (F2) above.

The soundness of these simulations relies on an application of König's Lemma. The finiteness requirement in (B2) therefore cannot be omitted. These simulations therefore usually cannot be applied with infinite nondeterminacy, e.g. [11, 3.8].

Unfortunately, in the rare cases where we needed prophecies, we had to prophesy a sequence number greater than some value. This is infinite nondeterminacy. We therefore developed in [11] an alternative that is simpler (to prove the soundness of) and theoretically more powerful: the eternity extension.

The idea of the eternity extension is that it is an extension with an immutable variable that is chosen nondeterminately at the start of the execution.

### 5.2 Eternity extensions

Let $K$ be a specification. Let $M$ be a set (of values for an eternity variable $\mathtt{m}$). A binary relation $R$ between $\mathit{states}(K)$ and $M$ is called a *behaviour restriction* of $K$ iff, for every behaviour *xs* of $K$, there is an $m \in M$ with $(xs(n), m) \in R$ for all $n \in \mathbb{N}$.

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

$$\begin{array}{l} \mathit{states}(W) = R \ , \\ \mathit{start}(K) = R \cap (\mathit{start}(K) \times M) \ , \\ ((x,m),(x',m')) \in \mathit{step}(W) \quad \equiv \quad (x,x') \in \mathit{step}(K) \ \wedge \ m = m' \ , \\ \mathit{ws} \in \mathit{prop}(W) \quad \equiv \quad \mathit{fst} \circ \mathit{ys} \in \mathit{prop}(K) \ . \end{array}$$

Here *fst* is the natural projection from $R$ to $\mathit{states}(K)$ and $\mathit{fst}^\omega$ is the lifting of *fst* to infinite lists. It is easy to verify that $\mathit{cvf} = \{(x,w) \mid x = \mathit{fst}(w)\}$ is a strict simulation $K \multimap W$.

In some sense the difficulty is moved to the user. The soundness proof of backward simulations is much more difficult than the soundness proof of eternity extensions. In order to adequately use an eternity extension, however, one has to collect into one eternity variable `m` all prophecies that may be needed in the entire behaviour, to formalize the requirements in a relation $R$, and to prove that $R$ is a behaviour restriction.

### *5.3 An example*

We give a simple example to show how a nontrivial eternity variable is used to prove that a given relation is a (strict) simulation. Let the specification $K$ and $L$ be given by

```
K :    var $j : \mathbb{N} := 0,\ b : \mathbb{B} := \mathit{false}$ ;
       ‖  $\neg b \;\to\; j := j + 1$ ;
       ‖  $j > 0 \;\to\; b := \mathit{true}$ ;
       prop $\Diamond b$ .
L :    var $k, n : \mathbb{N} := 0, 0$ ;
       ‖  $n = 0 \;\to\; k := 1$ ; choose $n > 0$ ;
       ‖  $k < n \;\to\; k := k + 1$ ;
       prop $\Diamond(k = n)$ .
```

In both specifications, $j$ or $k$ is incremented a positive number of times after which a stable state is reached. Specification $L$ may look strange: it has finite initial executions that cannot be extended to behaviours with the property $\Diamond(k = n)$. This is expressed by saying that $L$ is not *machine closed* [1].

Let relation $F$ between the two state spaces be given by

$$((j,b),(k,n)) \in F \quad \equiv \quad j = k \ .$$

In every behaviour, $L$ chooses the upper bound for $k$ as a first step. For specification $K$, the upper bound is chosen in the last nonstuttering step. It therefore requires a prophecy to construct the behaviour in $L$ from the one in $K$.

We therefore factor relation $F$ over an eternity extension. We form the eternity extension of $K$ with an eternity variable $m : \mathbb{N}$ and the relation

$$R \quad \equiv \quad (\neg\, b \;\vee\; j = m) \ .$$

Every behaviour *xs* of $K$ has a first state where $b$ holds after which $j$ cannot change anymore. If we choose $m$ equal to the final value of $j$, it is easy to see that all states of *xs* satisfy $R$. This proves that $R$ is a behaviour restriction. Let $W$ be the corresponding eternity extension with the strict simulation $\mathit{cvf} : K \twoheadrightarrow W$.

We form a refinement function $f : W \twoheadrightarrow L$ by $f(j, b, m) = (j, (j = 0\ ?\ 0 : m))$. It is clear that $f$ maps initial states to initial states. A step where $b$ becomes true corresponds to a skip step of $L$. In order to prove that a step of $W$ where $j$ is incremented is mapped to a step of $L$, it suffices to prove that $W$ has the invariant $J : j \le m$.

Predicate $J$ does not hold in all reachable states of $W$ (indeed all states $(j, b, m)$ with $b = \mathit{false}$ are reachable in $W$). Predicate $J$ does hold for all states that occur in behaviours of $W$. Indeed, if $w = (j, b, m)$ is in a behaviour of $W$, there is a sequence of steps from $w$ in which eventually $b = \mathit{true}$ holds. At that point, we have $j = m$ because of behaviour restriction $R$. Since steps in $W$ never decrease $j$ and never modify $m$, it follows that $j \le m$ holds in $w$. An invariant like this, which is not proved by forward induction, but by going backwards from infinity, is called a backward invariant.

## *5.4 Episodic simulations*

Recently, we found a compromise between forward simulations and backward simulations that avoids the finiteness condition in (B2). The idea is to require that, from time to time, all prophecies have been fulfilled. Periods with possibly outstanding prophecies are called *episodes*. Episodic simulations behave as backward simulations during episodes, and forward simulations elsewhere.

Let $K = (X, X_0, N, P)$ be a specification. We define an *episodic set* of $K$ to be a subset $V$ of $\mathit{states}(K)$ that satisfies

(EpS0) $\quad X_0 \cap V = \emptyset$ ,
(EpS1) $\quad \mathit{Beh}(K) \subseteq \Box\Diamond(\neg\, V)$ .

The elements of $V$ are called of *episodic* states. (EpS0) expresses that start states are never episodic; (EpS1) expresses that episodes always terminate.

Now let $L$ be a second specification. A relation $F$ between $X$ and $states(L)$ is called an *episodic* simulation from $K$ to $L$ for $V$ iff $V$ is an episodic set and relation $F$ satisfies the conditions (F0) and (F2) of forward simulations and moreover

$$\begin{array}{ll}
\text{(epFW)} & (x,x') \in N \ \wedge\ (x,y) \in F \ \wedge\ x,x' \in X \setminus V \\
 & \Rightarrow\ \ \exists\, y' : (x',y') \in F \ \wedge\ (y,y') \in step(L)\ , \\
\text{(epBW)} & (x,x') \in N \ \wedge\ (x',y') \in F \ \wedge\ x,x' \in V \\
 & \Rightarrow\ \ \exists\, y : (x,y) \in F \ \wedge\ (y,y') \in step(L)\ , \\
\text{(epTot)} & (x,x') \in N \ \wedge\ x \in V \ \wedge\ x' \notin V \\
 & \Rightarrow\ \ \exists\, y,y' : (x,y) \in F \ \wedge\ (y,y') \in step(L) \ \wedge\ (x',y') \in F\ , \\
\text{(epCon)} & (x,x') \in N \ \wedge\ x \notin V \ \wedge\ x' \in V \ \wedge\ (x,y) \in F \ \wedge\ (x',y') \in F \\
 & \Rightarrow\ \ (y,y') \in step(L)\ .
\end{array}$$

These four conditions are graphically summarized in:

$$\begin{tikzcd}
 & \text{FW} & & \text{BW} & & \text{Tot} & & \text{Con} & \\
K: \arrow[d, dash, "F"'] & x \arrow[r] \arrow[d, dash] & x' \arrow[d, dash, dashed] & x \arrow[r] \arrow[d, dash, dashed] & x' \arrow[d, dash] & x \arrow[r] \arrow[d, dash, dashed] & x' \arrow[d, dash, dashed] & x \arrow[r] \arrow[d, dash] & x' \arrow[d, dash] \\
L: & y \arrow[r, dashed] & y'? & ?y \arrow[r, dashed] & y' & y \arrow[r, dashed, "?"] & y' & y \arrow[r, dashed, "?"] & y'
\end{tikzcd}$$

The conditions (epFW) and (epBW) are restricted versions of (F1) and (B1) for forward and backward simulations. Conditions (epTot) and (epCon) serve to connect episodes with nonepisodic periods. One may note that the graph of a refinement mapping is an episodic simulation for every episodic set.

**Theorem 5.1** *Let $F$ be an episodic simulation from $K$ to $L$. Then $F$ is a strict simulation $K \multimap\!\!\!> L$.*

*Proof.* Let $xs$ be a behaviour of $K$. In order to construct a corresponding behaviour of $L$, we define the set $\Phi$ to consist of the nonempty finite initial executions $ys$ of $L$ that satisfy $(xs(i), ys(i)) \in F$ for $0 \leq i < \#ys$. Let $\Phi^+$ be the set of $ys \in \Phi$ with $xs(\#ys - 1) \notin V$.

The set $\Phi^+$ is nonempty because of the conditions (F0) and (EpS0). We next prove that every sequence in $\Phi^+$ is a prefix of a longer sequence in $\Phi^+$.

Let $ys \in \Phi^+$, say with $n = \#ys$. Then $n \geq 1$ and $xs(n-1) \notin V$. If $xs(n) \notin V$, condition (epFW) enables us to extend $ys$ with a single element in such a way that it remains in $\Phi^+$. Otherwise $xs(n) \in V$. By condition (EpS1), there is a number $k > n$ such that $xs(k) \notin V$ and $xs(i) \in V$ for $n \leq i < k$. By condition (epTot), we can choose $ys(k-1)$ and $ys(k)$ with $(xs(k-1), ys(k-1)) \in F$ and $(ys(k-1), ys(k)) \in step(L)$ and $(xs(k), ys(k)) \in F$. Working backward with (epBW), we can choose $ys(i)$ with $(xs(i), ys(i)) \in F$ and $(ys(i), ys(i+1)) \in step(L)$ for all $i$ with $k - 1 > i \geq n$. By (epCon), we also have $(ys(n-1), ys(n)) \in \mathit{step}(L)$. In this way, $ys$ is extended to an initial execution of length $k > n$, while remaining in $\Phi^+$.

Since $\Phi^+$ is nonempty, and every sequence in it can be extended to a longer sequence in it, we can make an infinite initial execution $ys$ of $L$ with $(xs, ys) \in F_\omega$. Condition (F2) finally implies that $ys$ is a behaviour of $L$. □

### *5.5 Lipton's simulation*

A special case of this kind of simulation is the *Lipton simulation* [21]. We simplify and slightly weaken the results of [3]. Given an episodic set $V$ of $K$, the idea is to construct a specification $L$ together with an episodic simulation $F : K \multimap L$. The state space of $L$ is the complement $W = X \setminus V$. We assume that the step relation $N$ of $K$ is approximated by related relations $N_V$ and $N_0$ on $X$ with

(Lip0) $\quad N \cap (V \times X) \subseteq N_0 \cup N_V$ ,
(Lip1) $\quad N_V \subseteq V \times X$ ,
(Lip2) $\quad N_0 \subseteq N \setminus (V \times W)$ .

The binary relations $N_V$ and $N_0$ on $X$ can be composed (with the operator ;), and repeatedly composed to form $N_V^+$ and $N_V^*$, etc. We need to postulate the commutation rule:

(Lip3) $\quad N_0; N_V \subseteq N_V^*; N_0$ .

We define the relations $S$ on $W$, and $F$ between $X$ and $W$ by

$$S = (N; N_V^*) \cap W^2 \ ,$$
$$F = N_V^* \cap (X \times W) \ .$$

We define $Q \subseteq W^\omega$ by

$$ys \in Q \quad \equiv \quad \exists\, yt \in W^\omega, xt \in \mathit{Beh}(K) : ys \preceq yt \ \wedge \ (xt, yt) \in F_\omega \ .$$

Let specification $L$ be given by $L = (W, X_0, S, Q)$. Condition (EpS0) ensures that $X_0$ can be taken as start space of $L$. Relation $S$ is reflexive on $W$, as required. The complicated definition of $Q$ ensures that $Q$ is insensitive to stuttering.

**Theorem 5.2** *Let $V$ be an episodic set of $K$. Let $W = X \setminus V$. Let $N_V$ and $N_0$ be chosen such that (Lip0) up to (Lip3) hold. Then relation $F$ is an episodic simulation $K \multimap L$.*

*Proof.* It follows from (Lip1) that, for $y \in W$,

(2) $$(x,y) \in F \ \wedge \ x \in W \quad \equiv \quad x = y \ ,$$
$$(x,y) \in F \ \wedge \ x \in V \quad \equiv \quad (x,y) \in N_V^+ \ .$$

By (EpS0), relation $F$ satisfies (F0).

Condition (epTot) is proved as follows. Let $(x, x') \in N$ and $x \in V$ and $x' \in W$. Then $(x, x') \in N_V$ because of (Lip0) and (Lip2). Taking $y = y' = x'$, we have $(x', y') \in F$ and $(y, y') \in S$ and $(x, y) \in F$.

Formula (epCon) clearly follows from the stronger formula

$$(3) \qquad (x, x') \in N \ \wedge \ x \in W \ \wedge \ (x, y) \in F \ \wedge \ (x', y') \in F \quad \Rightarrow \quad (y, y') \in S \ .$$

Formula (3) itself is proved as follows. Let $(x, x') \in N$ and $(x, y) \in F$ and $(x', y') \in F$ and $x \in W$. We have $x = y$ because of (2) and $(x, y) \in F$ and $x \in W$. We have $(x', y') \in N_V^*$. Therefore $(y, y') \in (N; N_V^*) \cap W^2 = S$.

Condition (epFW) is proved as follows. Let $(x, x') \in N$ and $(x, y) \in F$ and $x$, $x' \in W$. We have $x = y$ and we can take $y' = x' \in W$ with $(x', y') \in F$ by (2) and $(y, y') \in \mathit{step}(L)$.

Condition (epBW) is proved as follows. Let $(x, x') \in N$ and $(x', y') \in F$ and $x$, $x' \in V$. Then $(x', y') \in N_V^+$ because of (2). If $(x, x') \in N_0$, then $(x, y') \in N_0; N_V^+ \subseteq N_V^*; N_0$ because of (Lip3). Therefore there is $y$ with $(x, y) \in N_V^*$ and $(y, y') \in N_0$. By (Lip2), we have $y \in W$ and $(y, y') \in N$. It follows that $(y, y') \in S$ and $(x, y) \in F$. Otherwise, we have $(x, x') \in N_V$ because of (Lip0). Therefore, $(x, y') \in N_V^+$ and we can take $y = y' \in W$. Then we have $(y, y') \in S$ and $(x, y) \in F$.

It is immediate that $F$ satisfies condition (F2). This concludes the proof that $F : K \mathrel{-\!\!\triangleright} L$ is an episodic simulation. □

The above is a variation of and heavily inspired by the paper [3]. This paper describes the Lipton simulation in TLA, but gives no proofs. It partitions the state space in three subsets rather than two. It has a third step relation, say $N_U$, with the condition $N_U; N_0 \subseteq N_0; N_U$. In our first application, this greater generality seems not to be needed. The theory is at least applicable to semaphore programs [21] and mutex programs in Posix threads.

## 6 Stutterings and nonstrictness

In concurrency, we abstract from time, but not from the order in which phenomena occur. This means that the fact that a state remains unchanged during a finite number of steps is not observable. This is formalized by the concept of stuttering. We have therefore to complicate matters by allowing nonstrict implementations and simulations.

### 6.1 Stutterings, and nonstrict implementation

A sequence *ys* is defined to be a *stuttering* of a sequence *xs*, notation $xs \preceq ys$, iff *ys* can be obtained from *xs* by replacing its elements by positive iterations of them. For example, if, for a finite list *vs*, we write $vs^\omega$ to denote the sequence obtained by concatenating infinitely many copies of *vs*, the sequence $(aaabbbccb)^\omega$ is a stuttering of $(abbccb)^\omega$. Formally, we define $xs \preceq ys$ to mean that there is a monotonic surjective function $g : \mathbb{N} \to \mathbb{N}$ with $ys = xs \circ g$. For instance, if $g(n) = \lfloor n/2 \rfloor$ and *xs* is stutterfree then *ys* stutters twice for every element of *xs*.

The difference between an infinite sequence and its stutterings must not be observable. Therefore, in section 2, it is postulated that the supplementary property $P$ is a *property* [1]. This means that it is insensitive to stuttering, i.e. that, for all $xs, ys \in X^\omega$,

$$xs \preceq ys \quad \Rightarrow \quad (xs \in P \equiv ys \in P)\ .$$

Recall that specification $K$ is a strict implementation of specification $L$ if every observed behaviour of $K$ is an observed behaviour of $L$. Now it may happen that all observed behaviours of $K$ are $(abb)^\omega$ and its stutterings, while the observed behaviours of $L$ are $(aab)^\omega$ and its stutterings. Therefore $K$ is not a strict implementation of $L$. Yet, $K$ must be accepted as an implementation of $L$ [20].

According to [1], therefore, specification $K$ is called an *implementation* of specification $L$ if every observed behaviour of $K$ has a stuttering that is an observed behaviour of $L$. We thus allow the observed behaviours of the implementation to be slowed down by inserting stutterings. We now also get nonstrict versions of simulations and forward simulations.

### 6.2 Nonstrict simulations

A relation $F$ between $states(K)$ and $states(L)$ is defined to be a *simulation* [16] from $K$ to $L$ (notation $K \twoheadrightarrow\!\!\rhd\rhd L$) if every behaviour *xs* of $K$ has a stuttering *xt* such that $(xt, ys) \in F_\omega$ for some behaviour *ys* of $L$. Again, it is easy to see that $K$ implements $L$ if, and only if, there is a nondisturbing simulation $K \twoheadrightarrow\!\!\rhd\rhd L$.

The nonstrict version of forward simulations is as follows. A relation $F$ between $states(K)$ and $states(L)$ is defined to be a *stuttering forward simulation* [16] from $K$ to $L$ iff it satisfies the conditions (F0), (F2) for forward simulations and

(SF1) For every pair $(x, x') \in step(K)$, there is an integer state function *vf* on $L$ such that, for every state $y \in states(L)$ with $(x, y) \in F$, there is a state

$y' \in \mathit{states}(L)$ with $(y, y') \in \mathit{step}(L)$ such that $(x', y') \in F$, or $(x, y') \in F$ and $\mathit{vf}(y) \geq 0$ and $\mathit{vf}(y') < \mathit{vf}(y)$.

It is not difficult to prove that, indeed, every stuttering forward simulation is a simulation.

## 7 Completeness and methodology

In [7], it was proved that every *data refinement* relation between terminating programs could be proved by a combination of forward and backward simulations. Such a result is called *semantic completeness.*

Our setting of possibly nonterminating, concurrent algorithms follows [1]. There, it was proved that, if specification $K$ is *machine closed* and specification $L$ has *finite invisible nondeterminism* and *internal continuity*, then every strict simulation $F : K \twoheadrightarrow L$ can be factored over a forward simulation, a backward simulation, and a refinement mapping.

In [10], because we could not satisfy the technical assumption of finite invisible nondeterminism, we replaced the prophecy variables by eternity extensions. In [11], we proved that every strict simulation that *preserves quiescence* can be factored over a forward simulation, followed by an eternity extension and a refinement mapping.

This result was strengthened considerably in [16] where we eliminated the conditions of strictness and preservation of quiescence. We summarize the main results here. It may be unexpected, but we have to introduce a history variable that only expresses that time increases forever.

Let $K$ be an arbitrary specification. We augment $K$ with an integer variable $t$ (for *time*) that is incremented with 1 in every nontrivial step, and also infinitely often. Formally, let $W = \mathit{cl}(K)$ be the specification defined by

$$\begin{aligned}
&\mathit{states}(W) = \mathit{states}(K) \times \mathbb{N} \ , \\
&\mathit{start}(W) = \mathit{start}(K) \times \{0\} \ , \\
&((x,t),(y,u)) \in \mathit{step}(W) \quad \equiv \\
&\qquad (x,y) \in \mathit{step}(K) \quad \wedge \quad (u = t+1 \ \vee \ (x = y \ \wedge \ t = u)) \ , \\
&\mathit{ys} \in \mathit{prop}(W) \quad \equiv \quad \mathit{fst} \circ \mathit{ys} \in \mathit{prop}(K) \ \wedge \ (\forall\, n : \exists\, i : \mathit{snd}(\mathit{ys}(i)) \geq n) \ .
\end{aligned}$$

It is easy to verify that $\mathit{step}(W)$ is reflexive and that $\mathit{prop}(W)$ is a property. So, indeed, $W$ is a specification. The projection function $\mathit{fst}$ is a refinement mapping $W \to K$. Its inverse relation $\mathit{ivf} = \mathit{fst}^{-1}$ is a strict simulation $\mathit{ivf} : K \twoheadrightarrow W$ because, for every behaviour $\mathit{xs}$ of $K$, the sequence $\mathit{ys} = \lambda i : (\mathit{xs}(i), i)$ is a behaviour of $W$ with $(\mathit{xs}, \mathit{ys}) \in \mathit{ivf}_\omega$. It is called the *clocking extension* of $K$.

Usually, $\mathit{ivf} : K \twoheadrightarrow \mathit{cl}(K)$ is not a forward simulation because it does not satisfy condition (F2). Our version of semantic completeness reads as follows.

**Theorem 7.1** *Let $K$ be a specification. The clocking extension cl$(K)$ has an eternity extension $e : cl(K) \mathrel{-\!\!\triangleright} E$ such that:*
*(a) for every strict simulation $F : K \mathrel{-\!\!\triangleright} L$ there is a refinement mapping $f : E \mathrel{-\!\!\triangleright} L$ with $(\mathit{ivf}; e\,; f) \subseteq F$.*
*(b) for every simulation $F : K \mathrel{-\!\!\triangleright\!\!\triangleright} L$ there is a stuttering forward simulation $g : E \mathrel{-\!\!\triangleright\!\!\triangleright} E'$ and a refinement mapping $f : E' \mathrel{-\!\!\triangleright} L$ with $(\mathit{ivf}; e\,; g\,; f) \subseteq F$.*

In this theorem, the clocking extension and the stuttering forward simulation $g$ are needed merely to control the execution speed. The real power of the theorem is in the eternity extension. In the strict case, we do not even need arbitrary forward simulations. In the nonstrict case, the stuttering forward simulation $g$ is used only to enforce stutterings. In some sense the eternity extension is too powerful. We have come to regard it as a kind of sledge hammer that is to be applied sparingly and with the utmost care.

Semantic completeness must not be confused with methodological convenience. In the completeness result, we only used refinement mappings, but no refinement functions and no strict forward simulations. Yet refinement functions and strict forward simulations are the main tools used in practical verifications.

In [13], we extended this repertoire with *splitting simulations* in which the progress property (F2) of forward simulations is replaced by a condition in terms of states and the step relation. This work needs further extension, because (F2) is an invitation for sloppy reasoning but splitting simulations are not often applicable. Indeed, whenever possible, it would be good to replace even condition (f2) for refinement functions by a condition in terms of the state and the step relation. Work in progress indicates that this can be done when the supplementary properties are given in terms of weak fairness.

It may well be that the *episodic* simulations of section 5.4 and more specifically the Lipton simulation can be used fruitfully more often than has been done so far.

In [14], we proved a refinement criterion for atomicity of read-write variables. The proof of this criterion is based on forward simulations, refinement functions, eternity extensions, and the new concept of *gliding simulations*. These gliding simulations are conceptually easier than eternity extensions, but technically nasty. The atomicity criterion provides a refinement justification for some older verifications and is also used in the recent verification [15] of algorithm C2 of Haldar and Vidyasankar.

## 8 In conclusion

In the course of several verification projects of concurrent algorithms, we were forced to use theorem provers, first NQTHM [2], later PVS [24], primarily for the administration of proof obligations. Using theorem provers forced us to emphasize the specifications that were to be proved. The adoption of refinement was forced upon us when we needed prophecies of the transactions in the serializable database interface of [10].

When working with a theorem prover, elegance is profitable because clumsy proof efforts take much more time than elegant ones. This also encouraged us to separate the practical verifications from the development of general theory of specifications and simulations.

## References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82:253–284, 1991.

[2] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1997.

[3] E. Cohen and L. Lamport. Reduction in TLA. In D. Sangiorgi and R. de Simone, editors, *CONCUR '98*, volume 1466 of *LNCS*, pages 317–331, New York, 1998. Springer.

[4] H. Gao, J.F. Groote, and W.H. Hesselink. Lock-free parallel and concurrent garbage collection by mark&sweep. *Sci. Comput. Program.*, 64:341–374, 2007.

[5] H. Gao and W.H. Hesselink. A formal reduction for lock-free parallel algorithms. In R. Alur and D.A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004*, volume 3114 of *LNCS*, pages 44–57. Springer, 2004.

[6] H. Gao and W.H. Hesselink. A general lock-free algorithm using compare-and-swap. *Information and Computation*, 205:225–241, 2007.

[7] J. He, C.A.R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *ESOP 86*, volume 213 of *LNCS*, pages 187–196, New York, 1986. Springer.

[8] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15:745–770, 1993.

[9] W.H. Hesselink. An assertional proof for a construction of an atomic variable. *Formal Aspects of Comput.*, 16:387–393, 2004.

[10] W.H. Hesselink. Using eternity variables to specify and prove a serializable database interface. *Sci. Comput. Program.*, 51:47–85, 2004.

[11] W.H. Hesselink. Eternity variables to prove simulation of specifications. *ACM Trans. On Comp. Logic*, 6:175–201, 2005.

[12] W.H. Hesselink. Refinement verification of the lazy caching algorithm. *Acta Inf.*, 43:195–222, 2006.

[13] W.H. Hesselink. Splitting forward simulations to cope with liveness. *Acta Inf.*, 42:583–602, 2006.

[14] W.H. Hesselink. A criterion for atomicity revisited. *Acta Inf.*, 44:123–151, 2007.

[15] W.H. Hesselink. A challenge for atomicity verification. *Sci. Comput. Program.*, 71:57–72, 2008.

[16] W.H. Hesselink. Universal extensions to simulate specifications. *Information and Computation*, 206:108–128, 2008.

[17] P. Jayanti and S. Petrovic. Efficient wait-free implementation of multiword LL/SC variables. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS), June 2005*, pages 59–68. IEEE, 2005.

[18] B. Jonnson. Simulations between specifications of distributed systems. In J.C.M. Baeten and J.F. Groote, editors, *CONCUR '91*, volume 527 of *LNCS*, pages 346–360, New York, 1991. Springer.

[19] L. Lamport. On interprocess communication. Parts I and II. *Distr. Comput.*, 1:77–101, 1986.

[20] L. Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32:32–45, 1989.

[21] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18:717–721, 1975.

[22] N. Lynch and F. Vaandrager. Forward and backward simulations, part I: untimed systems. *Inf. Comput.*, 121:214–233, 1995.

[23] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Int. Joint Conf. on Artificial Intelligence*, pages 481–489. British Comp. Soc., 1971.

[24] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference*, 2001. http://pvs.csl.sri.com