Science of Computer Programming 76 (2011) 739–755

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Simulation refinement for concurrency verification

Wim H. Hesselink*

Department of Computing Science, University of Groningen, P.O. Box 407, 9700 AK Groningen, The Netherlands

A R T I C L E   I N F O

Article history:
Received 29 August 2008
Received in revised form 27 April 2009
Accepted 2 September 2009
Available online 15 September 2009

Keywords:
Refinement
Simulation
Atomicity
Verification
Semantic completeness

A B S T R A C T

In recent years, we applied and extended the theory of Abadi and Lamport (1991) [1] on the existence of refinement mappings. The present paper presents an overview of our extensions of the theory. For most concepts we provide examples or pointers to case studies where they occurred. The paper presents the results on semantic completeness. It sketches out how the theory is related to the other formalisms in the area. It discusses the tension between semantic completeness and methodological convenience. It concludes with our experience with the theorem provers NQTHM and PVS that were used during these projects.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

The aim of this paper is to present and extend the methods we developed to verify concurrent and reactive algorithms during the last years. It is a complete revision of [22].

The central thesis of refinement theory is that programs and algorithms are nothing but specifications that happen to be executable, and that implementation is a relation between an abstract specification and an executable one. Using the same language for specifications of different levels of abstractness eliminates irrelevant syntactic differences. It therefore highlights and clarifies the genuine differences between the levels.

The starting point for every verification is the abstract specification. Concurrent algorithms are often reactive: 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, reactive programs cannot be specified by preconditions and postconditions only.

In principle, a terminating concurrent algorithm specified by means of preconditions and postconditions can be treated as a highly nondeterministic sequential program in the way advocated and developed by Dijkstra and others, e.g., cf. [7]. Yet, even for such an algorithm, the proof of termination may rely on an extensive analysis of the conceivably nonterminating algorithm.

One of the central problems of concurrency is the question of refinement of atomicity, cf. [6,33]. This is the question as to whether an atomic command at a certain level of abstractness can be refined or implemented by a sequence of fine-grain instructions at a lower level of abstractness. In such cases, it is unknown and irrelevant whether the algorithm that contains the coarse grain atomic command terminates or not. Refinement of atomicity is therefore usually modelled in a nonterminating setting. Mutual exclusion is the very first case of this, but the serializable database interface of, for instance, [45,30,16] is a more complicated example.

Around 1990, it was realized that, for concurrency, the nonterminating setting was more natural than the terminating setting. Several formalisms were proposed for the nonterminating setting. Chandy and Misra [10] proposed UNITY, Abadi and

* Tel.: +31503633933; fax: +31503800.
E-mail address: w.h.hesselink@rug.nl.

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.09.006

Lamport [1] proposed state machine specifications, Back and von Wright [5] proposed trace semantics for action systems, Manna and Pnueli [36] proposed fair transition systems.

These formalisms are syntactically different, but semantically they are closely related. The main idea is that, after some initialization phase, the global state of the system is subject to an infinite sequence of nondeterministic changes, as specified by the algorithm. The distribution of the changes over concurrent processes is of later concern, as argued by [10,32].

In most of these formalisms, the behaviors are infinite sequences of states, and termination is defined by the condition that the state eventually remains constant. In general, progress conditions are specified in some form of temporal logic, though the formalisms differ in the emphasis they put on this.

In order to structurally allow infinite behaviors to terminate in eventually constant states, some of these formalisms allow, in every state, a skip step in which the state remains unchanged. In other words, the next state relation is supposed to reflexive. This also has the advantage that, in refinement, the abstract and concrete state space can be kept in lockstep even when the abstract specification skips and the concrete specification performs unobservable but useful computation.

Reflexivity of the next state relation is also very natural when the system is in parallel composition with an environment that may stop to be interested in the services that the system provides. In order to enforce any kind of progress, however, reflexivity of the next state relation requires some nontrivial progress condition in the specification.

Based on these considerations, we have adopted the formalism of [1] where a specification is a state machine with a supplementary property. The latter usually serves as a progress requirement. We do not use TLA of [31], because we prefer to distinguish the various state spaces involved and to distinguish the step relations within one state space from the simulation relations between different state spaces. We postpone the introduction of program variables in the theory of simulation relations, in deviation of [1]. Of course, program variables are unavoidable in concrete verifications, and there we use a syntactic format similar to Back's action systems.

Contributions. The main purpose of the paper is to present an overview of the methods and results developed in [17,19, 23] as applied in our case studies in [16,18,20,21]. This paper is a complete revision of [22]. There are several new ingredients.

In (2.4), we indicate how explicit termination can be treated in our seemingly nonterminating setting. We specialize the refinement mappings of [1] to refinement functions and weak fairness refinement functions, which are more convenient for concrete verifications. We give simple but new examples of refinement functions in which the progress conditions require special care. In (3.2) and (3.4), we show how, in both specification and implementation, the environment can be separated from the system that is to be implemented. In (5.6), we indicate how Lipton's theory for mutual exclusion fits in the episodic simulations introduced in [22].

Overview. Section 2 gives the basic formalisms for temporal logic, specifications, executions, behaviors, invariants, visibility, and explicit termination.

In Section 3, we introduce refinement functions with special attention to preservation of progress. A small case study of Compare and Swap variables illustrates the refinement concepts. We introduce weak fairness refinement functions. This concept is illustrated by an implementation of a barrier with no other synchronization primitives than atomic variables.

In Section 4, we present strict and nonstrict implementations and simulations. The nonstrict versions require some more care in the treatment of stuttering. We present forward simulations as special cases of strict simulations, and the clocking simulation as a strict simulation that need not be a forward simulation.

In Section 5, we introduce Lamport's concept of prophecies. Three formalizations are treated: backward simulations, eternity extensions, and episodic simulations. We give an example of an eternity extension, which shows an invariant that cannot be proved by induction from the initial states. The episodic simulations specialize to Lipton's simulation for mutual exclusion. This has applications to mutex abstraction in pthread programs.

In Section 6, we present the result of semantic completeness of the simulation concepts of [23] and sketch the tension between semantic completeness and methodological convenience.

Section 7 gives a comparison with some other formalisms in the area. We conclude in Section 8 with remarks about our experience with the use of the theorem provers NQTHM and PVS.

2. Specifications

In this section, we present our formalism for specifications, a syntactic variation of [1]. If X stands for the state space, predicates on X correspond to sets of states, relations over X correspond to possible state transformations, computations give rise to infinite sequences over X. A specification is a state machine over X with a supplementary property to specify progress.

2.1. Predicates, subsets, and relations

A predicate (boolean function) on a set X is identified with the subset of X where the predicate holds. We can therefore identify conjunction ($\wedge$) with intersection ($\cap$) and disjunction ($\vee$) with union ($\cup$). Negation ($\neg$) is the same as complementation with respect to X. Implication is the set operation with $(U \Rightarrow V) = (\neg U \vee V)$. On the other hand, $U \subseteq V$ expresses that predicate U is stronger than predicate V, i.e., that $(\neg U \vee V) = X$.

A binary relation on a set X is identified with the set of pairs that satisfy the relation; this is a subset of the Cartesian product $X \times X = X^2$. We write $\mathbf{1}$ for the identity relation of X.

2.2. Temporal formulas

System behaviors will be modelled as sequences of consecutive states. We therefore introduce the set X^ω of the infinite sequences on X, which are regarded as functions $\mathbb{N} \to X$. For a sequence $xs \in X^\omega$ and $k \in \mathbb{N}$, we write $xs|k$ (pronounce *xs from k*) for the suffix of xs where the first k elements have been removed, so that $(xs|k)(n) = xs(k+n)$. If P is a set of sequences, the sets $\Box P$ (henceforth P), and $\Diamond P$ (eventually P) are defined by

$$\begin{array}{lll} xs \in \Box P & \equiv & (\forall\, k \in \mathbb{N} : (xs|k) \in P)\ , \\ xs \in \Diamond P & \equiv & (\exists\, k \in \mathbb{N} : (xs|k) \in P)\ . \end{array}$$

So, $xs \in \Box P$ means that all suffixes of xs belong to P, and $xs \in \Diamond P$ means that xs has some suffix that belongs to P. It follows that $\Diamond P = \neg\Box\neg P$

For $U \subseteq X$, we define the subset $[\![\, U \,]\!]$ of X^ω to consist of the sequences whose first element is in U. For a relation C on X, we define the subset $[\![\, C \,]\!]_2$ of X^ω to consist of the sequences that start with an C-transition. So we have

$$\begin{array}{lll} xs \in [\![\, U \,]\!] & \equiv & xs(0) \in U\ , \\ xs \in [\![\, C \,]\!]_2 & \equiv & (xs(0), xs(1)) \in C\ . \end{array}$$

In temporal logic, these operators are usually kept implicit.

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, so that $v = xs(n)$ is replaced by $v^{d(n)}$ for some function $d : \mathbb{N} \to \mathbb{N}_+$. For example, if, for a finite list vs, we write vs^ω to denote the sequence obtained by concatenating infinitely many copies of vs, the sequence $(aaabbbccb)^\omega$ is a stuttering of $(abbccb)^\omega$.

Since we do not want to attach clock speeds to our specifications, it is important to regard behaviors xs and ys with $xs \preceq ys$ as indistinguishable. A subset P of X^ω is called a *property* [1] iff it is insensitive to stutterings, i.e., if $(xs \in P) \equiv (ys \in P)$ whenever $xs \preceq ys$. If P is a property, then $\Box P$, and $\Diamond P$, and $\neg P$ are properties. The conjunctions and disjunctions of properties are properties. $[\![\, U \,]\!]$ is a property for every $U \subseteq X$. If A is a reflexive relation on X, then $\Box\, [\![\, A \,]\!]_2$ is a property. If A is irreflexive, $\Diamond\, [\![\, A \,]\!]_2$ is a property.

Example. The set $\Box\Diamond\, [\![\, \mathbf{1} \,]\!]_2$ consists of the sequences that stutter infinitely often. This set is not a property (if X has more than one element).

2.3. Specifications, programs, and behaviors

As announced, our primary definition is due to [1]:

Definition 2.1. A *specification* is a tuple $K = (X, A, N, P)$ where X is the state space, $A \subseteq X$ is the set of initial states, $N \subseteq X^2$ is the next-state relation and P is the supplementary property. Relation N is required to be reflexive in order to allow stutterings. P is a subset of X^ω, and is required to be a property.

We define an *initial execution* of K to be a finite or infinite sequence xs over X with $xs(0) \in A$ and such that every pair of consecutive elements belongs to N. A *behavior* of K is an infinite initial execution xs of K with $xs \in P$. We write $Beh(K)$ to denote the set of behaviors of K. It is easy to see that

$$Beh(K) = [\![\, A \,]\!] \cap \Box[\![\, N \,]\!]_2 \cap P\ .$$

The rules for properties imply that $Beh(K)$ is always a property.

For a specification $K = (X, A, N, P)$, we write $states(K) = X$, $start(K) = A$, $step(K) = N$, $prop(K) = P$.

When presenting a specification, we often use a program-like notation, as in the action systems of [5]. Then the state space is spanned by the variables declared. The set of initial states is determined by the initial values of the variables, as given by the declaration. The next-state relation N is given as a program in guarded command notation, where we keep the possibility of stuttering steps implicit. A construct of the form

$$\begin{array}{llll} [] & U_0 & \to & S_0\ . \\ [] & U_1 & \to & S_1\ . \end{array} \qquad \text{(W)}$$

denotes a next-state relation that is the union of the identity relation $\mathbf{1}$ with the sets $S_i \cap (U_i \times X)$. So, it is a nondeterminate choice between the *skip* command and the guarded commands $U_i \to S_i$, which is taken atomically and repeatedly. It can be compared with the assignment section in UNITY [10]. A parallel composition of such constructs (W) denotes the union of their next-state relations. The difference with Dijkstra's **do od** notation is that the **do od** construct terminates when none of the guards hold, whereas (W) never terminates. When none of the guards hold, the construct (W) just skips, waiting for some other component to modify a guard.

The supplementary property is given separately by means of some temporal logic formula, preceded by **prop**. In the design of our specifications, we prefer to keep the supplementary properties as weak as possible. They are mainly used to express progress conditions. Note that, since it is reflexive, the next-state relation can never express progress.

A specification K is called *machine-closed* [1] if every finite initial execution can be extended to a behavior. We do not require machine closure but give the definition because it is occasionally used later.

A state x of K is called *reachable* iff there is an initial execution xs of K and an index n with $xs(n) = x$. It is called *occurring* if there is a behavior xs of K and an index n with $xs(n) = x$.

We define a set of states J to be an *invariant* if J contains all occurring states. J is called *inductive* if $start(K) \subseteq J$ and $y \in J$ for every pair $(x, y) \in step(K)$ with $x \in J$.

Clearly, every occurring state is reachable. If K is machine-closed, every reachable state is occurring. Every inductive set contains all reachable states and hence all occurring states, and is therefore an invariant. As the next example shows, reachable states need not be occurring, and invariants need not be inductive.

Example. Let specification K be given by

```
K :      var j : ℕ := 0 .
         []  true   →   j := j + 1 .
         []  j = 5  →   j := 0 .
         prop ◇□[[ j = 0 ]] .
```

By this notation, we mean that K has the state space $\mathbb{N}$ and the start set $\{0\}$. The next-state relation is $\{(5, 0)\} \cup \{(j, k) \mid k \in \{j, j+1\}\}$ (note that reflexivity is kept implicit in the guarded-command notation). The supplementary property is given by the temporal formula, which expresses that the system terminates in a state with $\mathtt{j} = 0$.

This specification is not machine-closed. In fact, the behaviors of K are the stutterings of the sequences $(012345)^*0^\omega$, but all natural numbers are reachable. The numbers > 5 are not occurring. The two sets given by the predicates $\mathtt{j} \le 5$ and $\mathtt{j} < 9$ are invariants. In this case, $\mathbb{N}$ is the only inductive set. □

2.4. Visibility and explicit termination

Systems are only useful by what we can observe of their behaviors. Following [1], we therefore assume that our specifications are *visible*, i.e., have a given observation function *obs* from the state space to some set *Obs* of observables. In principle, we are therefore primarily interested in the *observed behaviors*, the sequences $obs \circ xs$ where xs ranges over the behaviors. As noted by a referee, however, if P is a property, the set $\{obs \circ xs \mid xs \in P\}$ need not be a property.

We can accommodate explicit termination by postulating a termination predicate *tm* on *Obs* and requiring that every visible specification K satisfies the *termination axiom* that there are no visible changes after termination:

$$(x, x') \in step(K) \ \wedge \ tm(obs(x)) \quad \Rightarrow \quad obs(x) = obs(x') \ . \tag{TA}$$

If K is in a state x with $tm(obs(x))$, K is said to have *terminated explicitly*. Specification K guarantees explicit termination if and only if $Beh(K) \subseteq \Diamond[\![\, tm \circ obs \,]\!]$.

In this way, we can smoothly incorporate visibility and explicit termination in our formalism. On the other hand, if we want to ignore these phenomena, we are free to do so, and the remainder of the formalism is not affected. Indeed, in the remainder of this paper, we usually ignore *obs*, and we do not use *tm*.

2.5. Weak fairness specifications

Weak fairness specifications are specifications with a supplementary property that consists of weak-fairness requirements. Recall that, in a setting with concurrent processes, a system is called *weakly fair* for a process q if q acts infinitely often in every behavior in which it is eventually always enabled. This is formalized as follows.

Let C be an irreflexive binary relation on a set X. The subset of X where C is *disabled* is defined as $dis(C) = \{x \in X \mid \forall\, y \in Y : (x, y) \notin C\}$. The *weak fairness* property $WF(C)$ of C is defined by

$$WF(C) = \Box\Diamond[\![\, C \,]\!]_2 \cup \Box\Diamond[\![\, dis(C) \,]\!] \ .$$

The set $\Box\Diamond[\![\, dis(C) \,]\!]$ is a property. Because C is irreflexive, the set $\Box\Diamond[\![\, C \,]\!]_2$ is a property as well. Therefore, $WF(C)$ is a property. Note that $WF(\emptyset) = X^\omega$ because $dis(\emptyset) = X$.

It is easy to see that $WF(C) = \Box\Diamond[\![\, \varepsilon(C) \,]\!]_2$ where $\varepsilon(C) = C \cup \{(x, y) \mid x \in dis(C)\}$. Let us define an *almost C step* to be a step of $\varepsilon(C)$. Then a sequence xs satisfies $WF(C)$ iff it has infinitely many almost C steps.

Definition 2.2. A *weak fairness specification* is a pair (K, Φ) where K is a specification, Φ is a set of irreflexive relations on $states(K)$, and $prop(K) = \bigcap_{C \in \Phi} WF(C)$.

We usually take each $C \in \Phi$ to be a subrelation of $step(K)$, but this is not necessary. The set Φ is allowed to be infinite, but in practice it is often a finite set indexed by process identifiers. The concept of a weak fairness specification is a variation of the splittings of [19, section 5.1]. There, however, we allowed a choice between weak and strong fairness and we imposed conditions on Φ that we abandon here.

3. Refinement functions

Roughly speaking, a specification K refines or implements a specification L iff every observed behavior of K is an observed behavior of L. To investigate refinement or implementation relations, we need to compare specifications and their behaviors.

A central aim in the methodology of concurrency verification, however, is to eliminate behaviors as much as possible from consideration, and rather argue about states and the next state relation. It is therefore important to give refinement criteria which are as much as possible in terms of states and the next state relation. As specifications may have arbitrary supplementary properties, we cannot ignore behaviors completely. The primary method for comparing specifications is by means of refinement functions. We give three versions of this concept. Two examples are given, with special care for the treatment of progress properties.

3.1. Refinement mappings and functions

A natural way to prove that one specification simulates another is by starting at the beginning and constructing the corresponding behavior in the other specification inductively, guided by a function from the first state space to the second one. This is formalized in the concept of refinement mapping of [1]:

Definition 3.1. If K and L are specifications, a function $f : \mathit{states}(K) \to \mathit{states}(L)$ is a *refinement mapping* 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) $f \circ xs \in \mathit{prop}(L)$ for every $xs \in \mathit{Beh}(K)$.

In practice, condition (f1) is stronger and often less convenient than condition (f1f) used in the following variation:

Definition 3.2. A function $f : \mathit{states}(K) \to \mathit{states}(L)$ is a *refinement function* iff it satisfies conditions (f0), (f1f) and (f2), where (f1f) is given by
(f1f) K has an invariant J such that $(f(x), f(x')) \in \mathit{step}(L)$ holds 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. The next subsection gives an example in which we do need a refinement function.

3.2. Atomic modification, a tiny case study

In order to illustrate the refinement concepts, let us introduce some concepts of atomicity of shared variables. In the setting of concurrency with shared memory, there are usually several processes that concurrently inspect and modify their own private variables but also some shared variables.

We use p and q to range over processes, i.e., over process identifiers. By convention, shared variables are written in typewriter font and private variables are written slanted. If v is a private variable, we write $v.q$ for the value of v of process q outside the program of q.

A shared variable is called *atomic* iff read and write operations of processes to it 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 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 atomic variables and for some purposes they are very useful. We here present an implementation of atomic modification by means of compare-and-swap variables.

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

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

where parameter x refers to the shared variable, *arg* is a private input variable, and *result* is a private output variable. We express the semantics of command C by the four place predicate $\mathit{Cpred}(\mathit{arg}, \mathtt{x}, \mathtt{x}^+, \mathit{result}^+)$ where $\mathtt{x}^+$ and result^+ stand for the values after execution of C. Command C can be nondeterministic: the values $\mathtt{x}^+$ and result^+ need not be functions of *arg* and x.

A CAS variable (Compare and Swap) is a special case of an atomically modifiable variable. It is a shared variable, say x, that can be read and written atomically, and that also supports the conditional update:

```
$\mathtt{CAS}(\mathtt{x}, u, v)$ returns $b : \mathbb{B}$ =
    $\langle$ if $\mathtt{x} = u$ then $\mathtt{x} := v$ ; $b := \mathit{true}$
      else $b := \mathit{false}$ 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. Note that CAS is a boolean function with a side effect on a shared variable.

A CAS variable can be used for lock-free implementation of general atomic modification in the following way:

$$\begin{array}{l}\textbf{repeat}\ \ u := \mathtt{x}\ ;\ \ v := u\ ;\ \ C(arg, v, result)\ ;\\ \textbf{until}\ \ \mathtt{CAS}(\mathtt{x}, u, v)\ .\end{array} \tag{RC}$$

The current value of x is copied to a private variable. Command C is executed on private variables so that there is no risk of interference. The only shared actions are the repeated CAS actions.

In order to prove the correctness of this implementation, we first give a specification. We split responsibilities. For every process p, its *environment* provides the calls and the arguments, and inspects the results, according to

$$\begin{array}{lllll} env.p: & [] & pc = 0 & \to & \textbf{choose}\ arg\ ;\ \ pc := 10\ .\\ & [] & pc = 1 & \to & out := result\ ;\ \ pc := 0\ .\end{array}$$

The *system* is specified by

$$sys.p: \quad [] \quad pc = 10 \quad \to \quad C(arg, \mathtt{x}, result)\ ;\ pc := 1\ .$$

The progress condition of *lock-freedom* for some process q is

$$LF.q: \qquad \Box(pc.q = 10 \ \Rightarrow\ \Diamond(\exists\ r : pc.r = 10 \land pc^{+}.r = 1))\ .$$

This means that, whenever $sys.q$ is enabled, eventually some process r (possibly q itself) will execute $sys.r$.

The only observable variables are $arg.q$ and $out.q$ for all processes q. The system is not allowed to modify $arg.q$ and $out.q$ (for a formal treatment of such stipulations, see [20]).

When formalizing states, private variables of processes are modelled as functions on processes. In the concrete specification, we therefore use the state space:

$$\begin{array}{l} Cstate : \mathtt{TYPE} =\\ \quad [\#\ \ \mathtt{x} : Node\,,\\ \qquad u, v\ : Process \to Node\,,\\ \qquad arg, result, out\ : Process \to Item\,,\\ \qquad pc\ : Process \to \mathbb{N}\quad \#]\,,\end{array}$$

where [# and #] are the type constructors for records of PVS. The abstract state space is the type *Astate*, which is *Cstate* without the variables u and v.

The step relation is now given by the abstract environment in parallel composition with the implemented system (RC) which is represented by

$$\begin{array}{lllll} Csys.p: & [] & pc = 10 & \to & u := \mathtt{x}\ ;\ \ v := u\ ;\ \ pc := 11\ .\\ & [] & pc = 11 & \to & C(arg, v, result)\ ;\ \ pc := 12\ .\\ & [] & pc = 12 & \to & pc := (\mathtt{CAS}(\mathtt{x}, u, v)\ ?\ 1 : 10)\ .\end{array}$$

The only progress condition needed for $LF.q$, i.e., lock-freedom for process q, is weak fairness for $Csys.q$.

In this case, there is no refinement function from the concrete specification to the abstract one because, once some process p has executed 11, the old value of $result.p$ is lost while the new value is not yet available. This can be resolved as follows. We introduce private history variables *oldresult* with the update $oldresult.p := result.p$ if *CAS* of p succeeds in 12. The introduction of history variables is treated below in Section 4.2. Adding *oldresult* to *Cstate* gives a state space *Hstate*, say. Line 12 of $Csys.p$ is replaced by

$$\begin{array}{l} []\quad pc = 12 \quad \to\\ \qquad \textbf{if}\ \ \mathtt{CAS}(\mathtt{x}, u, v)\ \ \textbf{then}\ \ oldresult := result\ ;\ \ pc := 1\\ \qquad \textbf{else}\ \ pc := 10\ \ \textbf{end}\,.\end{array}$$

Again, the progress condition is weak fairness for the modified version of $Csys.q$.

We now can relate the modified concrete specification with the abstract specification by means of the refinement function $f : Hstate \to Astate$ given by

$$\begin{array}{l} f(s : Hstate) =\\ \quad (\#\ \ \mathtt{x} := s.\mathtt{x}\ ,\ \ arg := s.arg\ ,\ \ out := s.out\ ,\\ \qquad result := s.oldresult\ ,\\ \qquad pc := \lambda\, q : \min(10, s.pc.q)\quad \#)\,,\end{array}$$

where (# and #) are the record constructors corresponding to [# and #].

In order to prove condition (f1f), we need the invariant

$$pc.q = 12 \quad \Rightarrow \quad Cpred(arg.q, u.q, v.q, result.q)\ . \tag{J0}$$

In order to prove that (J0) is an invariant, we need the obvious auxiliary invariant

$$pc.q = 11 \quad \Rightarrow \quad u.q = v.q\ . \tag{J1}$$

Condition (f2) is proved as follows. Consider a behavior of the concrete specification in which at some point $pc.q \geq 10$. According to $LF.q$, we have to prove that after this point some process r does a step with $pc.r \geq 10 \ \wedge \ pc^{+}.r = 1$. Assume this is not the case. Then no process executes a succeeding `CAS` anymore. Therefore `x` remains constant. Since q itself executes under weak fairness and is always enabled, it has $u.q = \mathtt{x}$ or it will set $u.q := \mathtt{x}$ in 10, it will then execute 11, followed by a succeeding `CAS` in 12, which is a contradiction.

3.3. Weak fairness refinement functions

While we strive to eliminate the behaviors from our considerations, we cannot remove them in condition (f2) of Definitions 3.1 and 3.2 because of the generality of the supplementary property. We now consider specifications with supplementary properties given by weak fairness in order to give a criterion for refinement functions between such specifications that does not mention behaviors. This criterion is applied to the correctness proof of a simple barrier.

Let (K, Φ) and (L, Ψ) be weak fairness specifications (Section 2.5). Consider a function $f : states(K) \to states(L)$ as a candidate refinement function. We cannot expect that function f maps every C step for $C \in \Phi$ to some D step with $D \in \Psi$, because a refinement function usually has the role to abstract from irrelevant detail, so that it maps some steps to skip steps, which are not allowed in irreflexive relations. Yet, we want to guarantee that enough C steps are mapped to D steps. We use a so-called variant function for this purpose. Even when there is some correspondence between the sets Φ and Ψ, progress for some $D \in \Psi$ may require collaboration of various alternatives $C \in \Phi$. We therefore let the set of productive alternatives be given by a function du (for *duty*).

Definition 3.3. Let (K, Φ) and (L, Ψ) be weak fairness specifications. A function $f : states(K) \to states(L)$ is called a *weak fairness refinement function* iff K has an invariant J such that the conditions (f0) and (f1f) of Definitions 3.1 and 3.2 hold and that:
(f2wf) For every $D \in \Psi$, there exist functions $vf : J \to \mathbb{N}$ and $du : J \to \mathbb{P}(\Phi)$ such that, for every pair $(x, y) \in step(K) \cap J^2$, we have:

$$\begin{aligned} &(f(x), f(y)) \in \varepsilon(D) \quad \vee \quad vf(y) < vf(x) \\ \vee \quad &(vf(x) = vf(y) \ \wedge \ \emptyset \neq du(x) \subseteq du(y) \\ &\quad \wedge \ (\forall\, C \in du(x) : (x, y) \notin \varepsilon(C))\ ) \ . \end{aligned}$$

Roughly speaking, for every occurring step of K and every $D \in \Psi$, four possibilities are allowed. Either it maps to a step of D, or it is disabled as a step of D, or vf decreases, or vf remains constant and the set du is nonempty and does not shrink, and, for all C in the set du, the step C is neither taken nor disabled.

Remarks. The reader may replace the target set $\mathbb{N}$ of vf by any well-founded set. The theorem below and its proof can remain unchanged. Weak fairness refinement functions can be compared with the liveness preserving forward simulations of [4]. In the case that the sets Φ and Ψ are countable, we do have an extension of Definition 3.3 to forward simulations, but we leave this aside because of the complicated proof and the lack of a suitable example.

Theorem 3.1. *Let (K, Φ) and (L, Ψ) be weak fairness specifications. Let $f : states(K) \to states(L)$ be a weak-fairness refinement function. Then f is a refinement function.*

Proof. According to the Definitions 3.2 and 3.3, it suffices to prove condition (f2). Let xs be a behavior of K. We need to prove that $f \circ xs \in WF(D)$ for every $D \in \Psi$. Assume $f \circ xs \notin WF(D)$ for some D. Then there is $i \in \mathbb{N}$ such that $(f(xs(n)), f(xs(n+1))) \notin \varepsilon(D)$ for all $n \geq i$. We now use function vf as provided by condition (f2wf). This function satisfies $vf(xs(n+1)) \leq vf(xs(n))$ for all $n \geq i$. It follows that this sequence eventually becomes constant because vf has values in $\mathbb{N}$. So, there is $j \geq i$ such that $vf(xs(n+1)) = vf(xs(n))$ for all $n \geq j$. By condition (f2wf), we then have $\emptyset \neq du(xs(n)) \subseteq du(xs(n+1))$ for all $n \geq j$. We can then choose $C \in du(xs(j))$, and obtain $(xs(n), xs(n+1)) \notin \varepsilon(C)$ for all $n \geq j$. This implies $xs \notin WF(C)$, contradicting that xs is a behavior of K. □

3.4. Design of a barrier

We illustrate Theorem 3.1 by a simple implementation of a so-called barrier. We therefore first introduce the idea of a barrier, then give a formal specification, provide an implementation with atomic shared variables, and finally give a proof with a weak fairness refinement function.

When a task is distributed over several processes, it is often the case that, at certain points in the computation, the processes must wait for each other before they can proceed with the next part of the computation. This is called *barrier synchronization* [2]. One can model the problem by letting each process execute the infinite loop:

```
while true do  TNS ; Barrier  end .
```

Here *TNS* stands for the terminating noncritical section, the actions of which are irrelevant for the problem at hand. The problem is to implement *Barrier* in such a way that, when some process is in its $n+1$st execution of *TNS*, all other processes have completed their nth execution of *TNS*.

To separate the implementation responsibilities, we distinguish within each process an environment part that executes *TNS*, counts the number of times *TNS* has been executed, and calls the barrier, and a barrier that re-enables the environment part when allowed. When this distinction has been made the irrelevant command *TNS* can be eliminated.

We thus specify the barrier as follows. We give each process p a specification variable $cnt.p$ to count the number of completed executions of *TNS* and a program counter $pc.p$ to enable the environment or the barrier. These variables have the initial values $cnt.p = pc.p = 0$ for all processes p. The actions of the environment are modelled as:

$$env.p: \quad pc = 0 \;\rightarrow\; cnt := cnt + 1 \;;\; pc := 10\,.$$

The barrier can enable the environment again when all processes have completed the previous execution of *TNS*:

$$bar.p: \quad pc = 10 \;\wedge\; (\forall\, q : cnt.p \le cnt.q) \;\rightarrow\; pc := 0\,.$$

In this specification, the abstract barrier is allowed to inspect the specification variables $cnt.q$. These variables are not available for the implementation of *bar*. Since they are not implemented, we need not be concerned with the fact that they are unbounded integers. The progress condition is weak fairness of $bar.p$ for all processes p: if, for some p, the step $bar.p$ is enabled indefinitely, the step will be taken eventually.

Every barrier implementor has to reckon with the possibility that some process p has passed the barrier, executed *TNS* again, and comes at the barrier, while some other process is still sleeping at the "previous" barrier and has not yet observed that it can pass. This scenario is perfectly legal, but some proposed barrier implementations cannot deal with it.

We now discuss an implementation of the barrier that only uses atomic read–write variables. Let N be the number of processes. Every process p only writes at an output variable $\texttt{tag}[p]$ by incrementing it modulo some number $R \ge 3$, and then waits until all other processes q have modified their output variables $\texttt{tag}[q]$ as well.

```
        var tag : array Process of $\mathbb{N} := (\lambda\, q : 0)$ ,
        privar $old : \mathbb{N} := 0$ ; $qq : Process$ ;
              $lis$ : set of $Process := \emptyset$ .

Cbar.p :  [] $pc = 10 \;\rightarrow\; \texttt{tag}[p] := (old + 1)$ mod $R$ ;
                         $lis := Process \setminus \{p\}$ ; $pc := 11$ .
          [] $pc = 11 \wedge lis = \emptyset \;\rightarrow\; old := \texttt{tag}[p]$ ; $pc := 0$ .
          [] $pc = 11 \wedge lis \ne \emptyset \;\rightarrow$ choose $qq \in lis$ ; $pc := 12$ .
          [] $pc = 12 \wedge old \ne \texttt{tag}[qq] \;\rightarrow\; lis := lis \setminus \{qq\}$ ; $pc := 11$ .
```

At 12, process p waits until process qq has executed 10. In the absence of other synchronization primitives, this is supposed to be done by busy waiting. The concrete specification is the parallel composition of the programs $env.p$ [] $Cbar.p$ for all processes p. The progress condition is weak fairness of $Cbar.p$ for all processes p: if, for some p, the step $Cbar.p$ is enabled indefinitely, the step will be taken eventually.

We construct a refinement function *fca* from the concrete state space to the abstract state space. As the abstract state space only has the (private) variables pc and cnt, we take

```
        $fca(s) = ($#
            $pc := \lambda\, q : \min(10, s.pc.q)$ ,
            $cnt := s.cnt$
        #) .
```

As before, the brackets (# and #) are the record constructors of PVS and $s.pc.q$ is pc of process q in the concrete state space s.

We claim that *fca* is a weak-fairness refinement function. Condition (f0) holds because the initial values of all $pc.p$ and $cnt.p$ of specification and implementation agree.

For condition (f1f), we need to show that $Cbar.p$ resets $pc.p := 0$ only under the precondition $(\forall\, q : cnt.p \le cnt.q)$. For this purpose, it is convenient to define state functions

$$ct(q) = (pc.q = 10\ ?\ cnt.q - 1 : cnt.q)\,.$$

Function $ct(q)$ is only modified when q executes 10, and then it is incremented with 1. If $pc.p \ne 10$ then $ct(p) \le ct(q)$ implies $cnt.p \le cnt.q$. It therefore suffices to prove that $Cbar.p$ resets $pc.p := 0$ only under the precondition $(\forall\, q : ct(p) \le ct(q))$. This follows when we postulate the invariant:

$$r \notin lis.q \;\Rightarrow\; ct(q) \le ct(r)\,. \tag{J0}$$

Predicate (J0) clearly holds initially. It is threatened only by the removal of qq from lis in 12. Therefore, (J0) is preserved if we also have the invariant:

$$pc.q > 10 \;\Rightarrow\; (old.q \ne \texttt{tag}[r] \;\equiv\; ct(q) \le ct(r)). \tag{Ja}$$

The equivalence is stronger than necessary at this point, but it will be needed below for progress. To prove (Ja), we postulate the invariants

$$\texttt{tag}[q] = ct(q) \;\mathbf{mod}\; R\,, \tag{J1}$$

$$pc.q > 10 \quad\Rightarrow\quad \mathtt{tag}[q] = (old.q + 1) \ \mathbf{mod}\ R \ , \tag{J2}$$

$$pc.q \le 10 \quad\Rightarrow\quad lis.q = \emptyset \ , \tag{J3}$$

$$ct(q) \le ct(r) + 1 \ . \tag{J4}$$

The values of $\mathtt{tag}[q]$, $ct(q)$, $old.q$, and $lis.q$ are only modified by actions of process q itself. Therefore, (J1), (J2), and (J3) are easily verified. Predicate (J4) can only be violated when process q executes 10. Then the precondition $ct(q) \le ct(r)$ holds because of (J0) and (J3). Therefore (J4) is preserved.

It remains to show that (Ja) is implied by (J1), (J2), and (J4). Under assumption of the antecedent of (Ja), its consequent is proved in

$$\begin{array}{ll}
 & old.q \ne \mathtt{tag}[r] \\
\equiv & \{\, pc.q > 10 \text{ and (J2); } \mathtt{tag}[r] \text{ and } old.q \text{ are in } [0 \ldots R) \,\} \\
 & \mathtt{tag}[q] \ne (\mathtt{tag}[r] + 1) \ \mathbf{mod}\ R \\
\equiv & \{\, \text{(J1) for } r \text{ and } q\text{; arithmetic} \,\} \\
 & ct(q) \ \mathbf{mod}\ R \ne (ct(r) + 1) \ \mathbf{mod}\ R \\
\equiv & \{\, \text{(J4) implies } ct(q) = ct(r) \text{ or } ct(r) \pm 1\text{; use } R \ge 3 \} \\
 & ct(q) \le ct(r) \ .
\end{array}$$

We finally verify condition (f2wf). Both specifications are weak-fairness specifications with the sets of alternatives indexed by processes. We now verify (f2wf) for a given process p. We thus assume that the concrete system does a step (x, y) such that $(fca(x), fca(y)) \notin \varepsilon(bar.p)$. Therefore the concrete step is not $pc.p := 0$, while the abstract step $bar.p$ is enabled. The latter condition means that, in the concrete system, we have $pc.p \ge 10$ and $(\forall\, q : cnt.p \le cnt.q)$.

The concrete system can do $pc.p := 0$ only after all processes q have modified $\mathtt{tag}[q]$ in 10, and process p itself has verified $old.p \ne \mathtt{tag}[q]$ for all q. We therefore define

$$\begin{array}{l}
DU = \{q \mid cnt.q = cnt.p \ \wedge\ pc.q = 10\} \ , \\
du = (DU \ne \emptyset \ ?\ DU : \{p\}) \ , \\
vf \ = \#DU + (pc.p = 10 \ ?\ 2 \cdot \#Process : 0) \\
\qquad\quad +2 \cdot \#lis.p + (pc.p = 11 \ ?\ 1 : 0) \ .
\end{array}$$

Now assume that concrete step (x, y) satisfies $vf(x) \ \ge\ vf(y)$. We have to verify the second line of the formula in Definition 3.3. First note that every step of p itself with precondition $pc.p \ge 10$ decreases vf. The preconditions $pc.p \ge 10$ and $(\forall\, q : cnt.p \le cnt.q)$ imply that no process can enter DU and, hence, that vf cannot increase. It follows that DU remains the same in the step. It also follows that du remains the same.

Finally, let $q \in du$. We have to prove $(x, y) \notin \varepsilon(Cbar.q)$. Firstly, assume $q \in DU$. Then q is at 10 and is therefore enabled. As it does not leave DU, it does not do a step. Therefore $(x, y) \notin \varepsilon(Cbar.q)$. Otherwise, $DU = \emptyset$ and $q = p$. Since DU is empty, we have $pc.p > 10$ and $ct(p) \le ct(r)$ for all r. By invariant (Ja), this proves that process p is enabled. Since vf does not decrease, p does not do a step. Therefore $(x, y) \notin \varepsilon(Cbar.p)$.

Remark. If one takes $R = 2$ and $N \ge 2$, this proof fails and the barrier can reach deadlock.

4. Implementations, simulations, and strictness

In Example 3.2, we needed to add a history variable in order to construct a refinement function. Extension with history variables is a special case of forward simulation. Even forward simulations, however, are not always sufficient to prove refinement.

We therefore go back to the foundations, to see what we really need. This leads to two theories (cf. [1]): a theory of strict implementations and simulations and a slightly more complicated theory of nonstrict implementations and simulations. For many applications, the strict theory is strong enough. In [17], we developed the strict theory only, and therefore used the terms implementation and simulation for the strict versions introduced below. The need for the nonstrict theory was already argued by [29], but we first needed it in [18].

4.1. The strict theory

Ultimately, specifications are only judged by their observed behaviors (Section 2.4). This suggests the following definition:

Definition 4.1. A visible specification K *strictly implements* a visible specification L if every observed behavior of K is an observed behavior of L.

In order to prove strict implementation, we must be able to look behind the scenes. We therefore introduce simulation relations.

Definition 4.2. A relation F between the state spaces of specifications K and L is a *strict simulation* from K to L (notation $F : K \twoheadrightarrow L$) iff, for every $xs \in Beh(K)$, there exists $ys \in Beh(L)$ with $(xs, ys) \in F_\omega$. Here F_ω is the set of pairs (xs, ys) with $(\forall\, i : (xs(i), ys(i)) \in F)$.

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 prove that K strictly implements L if and only if there is a nondisturbing strict simulation $F : K \twoheadrightarrow L$, cf. [17, Thm. 2.6]. We can therefore use simulations to prove implementation relations, and we are mainly interested in nondisturbing simulations. Mostly, however, we take nondisturbingness for granted, and even ignore all visibility aspects.

4.2. Forward simulations

Simulations (strict or otherwise) are defined in terms of behaviors, but we prefer to argue with states and the next state relation. Therefore, for practical verifications, we need simulation criteria that avoid the behaviors as much as possible.

Refinement functions can serve here, but more general are forward simulations [14,35,39]. These also generalize the extensions with history variables [1].

Definition 4.3. A relation F between $states(K)$ and $states(L)$ is called a *forward simulation* from specification K to specification L iff

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

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

Lemma 4.1. *(a) Let $f : states(K) \to states(L)$ be a refinement function from a specification K to a specification L, say with invariant J. Then the graph $\{(x, f(x)) \mid x \in J\}$ 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 $F : K \twoheadrightarrow L$.

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

In Example 3.2, a forward simulation F between *Cstate* and *Hstate* can be defined by requiring equality when *oldresult* is ignored. It is straightforward to verify that this relation satisfies the conditions (F0), (F1), and (F2). Indeed, this example belongs to the following class of forward simulations for which condition (F2) can be proved directly from the definitions.

Definition 4.4. Relation F between $states(K)$ and $states(L)$ is a *history extension* iff it satisfies the conditions (F0) and (F1) and there is a function $f : states(L) \to states(K)$ with $F = \{(f(y), y) \mid y \in states(L)\}$ and $prop(L) = \{ys \mid f \circ ys \in prop(K)\}$.

This definition differs from the one in [23, 4.2], but it seems to coincide with it in all relevant cases, and it is more useful in practice. The main reason is the following easy result:

Lemma 4.2. *Every history extension is a forward simulation.*

For example, in [20], the forward simulations $HRW \twoheadrightarrow KRW$ and $PRW \twoheadrightarrow QRW$ are both history extensions.

4.3. The clocking extension

The idea of weak fairness refinement functions can be generalized to forward simulations, but the technicalities are too much for the present paper. There is one case, however, that we need in Section 6, and that we can easily present. It is a construction that extends an arbitrary specification with 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 $W = cl(K)$ be the specification defined by

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

It is easy to verify that $step(W)$ is reflexive and that $prop(W)$ is a property. So, indeed, W is a specification. The projection function fst is a refinement mapping $W \to K$.

The important point is, that the inverse relation $ivf = fst^{-1}$ is a strict simulation $ivf : K \twoheadrightarrow W$ because, for every behavior xs of K, the sequence $ys = \lambda i : (xs(i), i)$ is a behavior of W with $(xs, ys) \in ivf_\omega$. It is called the *clocking extension* of K. Usually, $ivf : K \twoheadrightarrow cl(K)$ is not a forward simulation because it need not satisfy condition (F2).

4.4. Stutterings and the nonstrict theory

In concurrency, we abstract from time, although not from the order in which phenomena occur. It is therefore not observable when a state remains unchanged during a finite number of steps. This is formalized by the concept of stuttering.

Recall from Section 2.2, that 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. 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.

It may happen that all observed behaviors of a specification K are $(abb)^\omega$ and its stutterings, while the observed behaviors of specification L are $(aab)^\omega$ and its stutterings. In this case, K is not a strict implementation of L. Yet, K must be accepted as an implementation of L as argued by [29]. We therefore need general implementations and general simulations. The following definition is equivalent to the one of [1].

Definition 4.5. Specification K is an *implementation* of specification L if every observed behavior of K has a stuttering that is an observed behavior of L.

We thus allow the observed behaviors of the implementation to be slowed down by inserting stutterings. We now also get nonstrict versions of simulations:

Definition 4.6. A relation F between $states(K)$ and $states(L)$ is a *simulation* from K to L (notation $K \twoheadrightarrow L$) if every behavior xs of K has a stuttering xt such that $(xt, ys) \in F_\omega$ for some behavior ys of L.

Again, it is easy to see that K implements L if, and only if, there is a nondisturbing simulation $K \twoheadrightarrow L$.

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* [23] 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 $y' \in states(L)$ with $(y, y') \in step(L)$ such that $(x', y') \in F$, or $(x, y') \in F$ and $vf(y) \geq 0$ and $vf(y') < vf(y)$.

It is not difficult to prove that, indeed, every stuttering forward simulation is a simulation. Note that it is occasionally useful to allow an integer-valued function vf that can be negative. On the other hand, one may replace the integers in (SF1) by a well-founded partial order as in [28, Fig. 21], [37,13].

Examples of nonstrict simulations are $K_0 \twoheadrightarrow K_2$ in [18, 6.5] and $QRW \twoheadrightarrow TRW$ in [20, 5.1]. We refer to [23] for examples of stuttering forward simulations and a more extensive discussion of the literature.

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*. In this section we give three sound formalizations of this idea. An example is given in Section 5.3 below.

5.1. Backward simulations

Prophecies can be formalized with the prophecy variables of [1] or (more or less equivalently) with the backward simulations of [25,35]. We use the definition of [17].

Definition 5.1. Relation F between $states(K)$ and $states(L)$ is a *backward simulation* from K to L iff
(B0) Every pair $(x, y) \in F$ with $x \in start(K)$ satisfies $y \in start(L)$.
(B1) For every pair $(x, x') \in step(K)$ and every y' with $(x', y') \in F$, there is y with $(x, y) \in F$ and $(y, y') \in step(L)$.
(B2) For every behavior xs of K, there are infinitely many indices n for which the set $\{y \mid (xs(n), y) \in F\}$ is nonempty and finite.
(B3) Requirement (F2) above.

The soundness of these simulations is expressed in:

Lemma 5.1. *Let F be a backward simulation from K to L. Then $F : K \rightarrow L$ is a strict simulation.*

This result is well known [1,25]. Its proof 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. [17, 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 [17] an alternative that is simpler (to prove the soundness of) and theoretically more powerful: the eternity extension. Roughly speaking, this is an extension with an immutable variable that is chosen with angelic nondeterminacy 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 $\mathtt{m}$).

Definition 5.2. A binary relation R between *states*(K) and M is a *behavior restriction* of K in M iff, for every behavior xs of K, there is an $m \in M$ with $(xs(n), m) \in R$ for all $n \in \mathbb{N}$.

The term behavior restriction was chosen in [16,17]. It restricts the behavior xs in the sense that the existence of m is a kind of consistency requirement on xs. If R is a behavior restriction of K, the corresponding *eternity extension* is defined as the specification W given by

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

Here fst is the natural projection from R to *states*(K) and fst^{ω} is the lifting of fst to infinite sequences. Let relation cvf between *states*(K) and R be given by $\mathit{cvf} = \{(x, w) \mid x = \mathit{fst}(w)\}$. It is easy to verify:

Lemma 5.2. *If R is a behavior restriction of K in M, relation cvf is a strict simulation $K \multimap\!\!\triangleright W$.*

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 $\mathtt{m}$ all prophecies that may be needed in the entire behavior, to formalize the requirements in a relation R, and to prove that R is a behavior restriction. Eternity extensions are applied in [16,20]. A closely related concept is introduced by [44].

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

In both specifications, j or k is incremented a positive number of times after which a stable state is reached. The main difference between them is that L chooses the upper bound for k as a first step, while the upper bound for $\mathtt{j}$ in K is chosen in the last nonstuttering step. It therefore requires a prophecy to construct a behavior in L from one in K.

Let relation F between the two state spaces be given by

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

As we need some kind of prophecy, we 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 behavior 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 behavior restriction. Let W be the corresponding eternity extension with the strict simulation $\mathit{cvf} : K \multimap\!\!\triangleright W$.

We form a refinement function $f : W \multimap\!\!\triangleright L$ by

$$f(j, b, m) = (j, (j = 0 \mathbin{?} 0 : m))\,.$$

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 hold for all states that occur in behaviors of W. Indeed, if $w = (j, b, m)$ is in a behavior of W, there is a sequence of steps from w in which eventually $b = true$ holds. At that point, we have $j = m$ because of behavior restriction R. Since steps in W never decrease j and never modify m, it follows that state w satisfies $j \le m$. An invariant like this, which is not proved by forward induction, but by going backwards from infinity, is called a backward invariant.

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. It follows that specification W is not machine-closed.

5.4. Episodic sets and simulations

Recently, inspired by [11], 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 as forward simulations elsewhere.

Let $K = (X, A, N, P)$ be a specification. We define an *episodic set* of K to be a set V of states such that start states are never episodic and that episodes (periods when the state is in V) always terminate. This is formalized in the two conditions:

$$A \cap V = \emptyset \;, \tag{EpS0}$$

$$Beh(K) \subseteq \Box\Diamond[\![\, \neg V \,]\!] \;. \tag{EpS1}$$

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, instead of (F1), for every pair $(x, x') \in N$:

$$x, x' \notin V \quad \Rightarrow \tag{epFW}$$
$$\forall\, y : (x, y) \in F \;\Rightarrow\; \exists\, y' : (x', y') \in F \;\wedge\; (y, y') \in step(L) \;,$$

$$x, x' \in V \quad \Rightarrow \tag{epBW}$$
$$\forall\, y' : (x', y') \in F \;\Rightarrow\; \exists\, y : (x, y) \in F \;\wedge\; (y, y') \in step(L) \;,$$

$$x \in V \;\wedge\; x' \notin V \quad \Rightarrow \tag{epTot}$$
$$\exists\, y, y' : (x, y) \in F \;\wedge\; (y, y') \in step(L) \;\wedge\; (x', y') \in F \;,$$

$$x \notin V \;\wedge\; x' \in V \quad \Rightarrow \tag{epCon}$$
$$\forall\, y, y' : (x, y) \in F \;\wedge\; (x', y') \in F \;\Rightarrow\; (y, y') \in step(L) \;.$$

These four conditions are graphically summarized in:

$$\begin{tikzcd}
 & \text{FW} & & \text{BW} & & \text{Tot} & & \text{Con} & \\
K: \arrow[d, no head, "F"'] & x \arrow[r] \arrow[d, no head] & x' \arrow[d, dashed, no head] & x \arrow[r] \arrow[d, dashed, no head] & x' \arrow[d, no head] & x \arrow[r] \arrow[d, dashed, no head] \arrow[dr, phantom, "?"] & x' \arrow[d, dashed, no head] & x \arrow[r] \arrow[d, no head] & x' \arrow[d, no head] \\
L: & y \arrow[r, dashed] & y'\,? & ?\,y \arrow[r, dashed] & y' & y \arrow[r, dashed] & y' & y \arrow[r, dashed, "?"] & y'
\end{tikzcd}$$

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 behavior of K. In order to construct a corresponding behavior of L, we define the set Φ to consist of the nonempty finite initial executions ys of L that satisfy $(xs(i), ys(i)) \in F$ for $0 \le i < \#ys$. We need to extend the sequences ys to the right. This is easy when (epFW) or (epCon) applies. We therefore define Φ^+ to be the set of $ys \in \Phi$ with $\#ys \ge 1$ and $xs(\#ys - 1) \notin V$. The set Φ^+ is nonempty because of the conditions (F0) and (EpS0). We next prove that every sequence in Φ^+ is a prefix of a longer sequence in Φ^+.

Let $ys \in \Phi^+$, say with $n = \#ys$. Then $n \ge 1$ 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 Φ^+. 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 $n \le i < k$. 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 $k - 1 > i \ge n$. By (epCon), we also have $(ys(n-1), ys(n)) \in step(L)$. In this way, ys is extended to an initial execution of length $k + 1 > n$, while remaining in Φ^+.

Since Φ^+ 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 behavior of L. $\square$

5.5. Lipton's simulation

A special case of this kind of simulation is the *Lipton simulation*. We simplify and slightly weaken the treatment of [11]. Recall that $K = (X, A, N, P)$. Given an episodic set V of K, the idea is to construct a specification L together with an episodic simulation $F : K \twoheadrightarrow L$. The state space of L is the complement $W = X \setminus V$. We assume that the step relation N of K is "approximated" by relations N_V and N_0 on X in the sense that

$$N \cap (V \times X) \subseteq N_0 \cup N_V \ , \tag{Lip0}$$

$$N_V \subseteq V \times X \ , \tag{Lip1}$$

$$N_0 \subseteq N \setminus (V \times W) \ . \tag{Lip2}$$

One can think of V as the set of states where some process is in its critical section. Condition (Lip0) introduces a case distinction for the steps that start in V. According to (Lip1) and (Lip2), all steps that leave V are collected in N_V.

The reason to introduce N_0 and N_V is to formulate a commutation requirement that "implements" the conditions for an episodic simulation. Indeed, 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:

$$N_0; N_V \subseteq N_V^*; N_0 \ . \tag{Lip3}$$

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

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

Let specification L be given by $L = (W, A, S, Q)$. Condition (EpS0) ensures that A 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 $K \twoheadrightarrow L$.*

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

$$(x, y) \in F \ \wedge \ x \in W \quad \equiv \quad x = y \ , \tag{0}$$
$$(x, y) \in F \ \wedge \ x \in V \quad \equiv \quad (x, y) \in N_V^+ \ .$$

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

$$(x, x') \in N \ \wedge \ x \in W \ \wedge \ (x, y) \in F \ \wedge \ (x', y') \in F \quad \Rightarrow \quad (y, y') \in S \ . \tag{1}$$

Formula (1) itself is proved as follows. Let $(x, x') \in N$ and $x \in W$ and $(x, y) \in F$ and $(x', y') \in F$. We have $x = y$ because of (0). 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 $x, x' \in W$ and $(x, y) \in F$. By (0), we have $x = y$ and we can take $y' = x' \in W$ with $(x', y') \in F$. Then $(y, y') \in S$.

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 (0). Because of $(x, x') \in N$ and (Lip0), we have $(x, x') \in N_0$ or $(x, x') \in N_V$. We treat these cases separately.

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 $(x, x') \in N_V$. Therefore, $(x, y') \in N_V^+$ and we can take $y = y' \in W$. Then we also have $(y, y') \in S$ and $(x, y) \in F$.

The definition of Q immediately implies that F satisfies condition (F2). This concludes the proof that $F : K \twoheadrightarrow L$ is an episodic simulation. $\square$

The above is a variation of, and heavily inspired by, [11]. This paper describes the Lipton simulation in TLA, its proofs are in a technical report posted on Lamport's TLA page. The paper 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 application below, this greater generality is not needed, and we see no applications where it would be needed.

5.6. Lipton's theory for mutual exclusion

The theory of [34] was developed for systems of concurrent processes with shared variables, with semaphores for mutual exclusion. We therefore introduce processes with their own step relations.

Let $K = (X, A, N, P)$ be a specification with $N = 1_X \cup \bigcup_{p \in \Pi} N.p$ where the set Π is regarded as a set of processes [32]. To model mutual exclusion, we assume that $\bot \notin \Pi$ and that some state function $\mu : X \to \Pi \cup \{\bot\}$ is given with the *locality* *property* that $\mu(x) = p$ with $p \in \Pi$ can only be made true or false by steps of process p itself. This property is formalized in

$$\begin{array}{l} (x, x') \in N.p \ \wedge \ \mu(x) \neq \mu(x') \\ \Rightarrow \quad (\mu(x) = \bot \ \wedge \ \mu(x') = p) \ \vee \ (\mu(x) = p \ \wedge \ \mu(x') = \bot) \ . \end{array} \qquad \text{(local)}$$

In view of the application below, we say that p *owns the mutex* in state x when $\mu(x) = p$, and that the mutex is *free* when $\mu = \bot$.

We strive at elimination of the episodes that $\mu(x) \neq \bot$. We thus define $V = \{x \in X \mid \mu(x) \neq \bot\}$ and $W = X \setminus V$. The set V is episodic if and only if $\mu(x) = \bot$ holds initially (i.e. in the set A of start states) and, in every behavior, infinitely often, or formally

$$A \ \Rightarrow \ \mu = \bot \ , \qquad \text{(EpS0)}$$

$$\Box\Diamond(\mu = \bot) \ . \qquad \text{(EpS1)}$$

We define

$$\begin{array}{l} N_0.p = \{(x, x') \in N.p \mid \mu(x) \neq p \neq \mu(x')\} \ , \\ N_1.p = \{(x, x') \in N.p \mid \mu(x) = p\} \ . \end{array}$$

In words, $N_0.p$ consists of the steps of p when it does not own, nor acquires, the mutex. $N_1.p$ consists of the steps of p with the precondition that p owns the mutex.

Lemma 5.3. *The subsets N_0 and N_V of N given by $N_0 = 1_X \cup \bigcup_p N_0.p$ and $N_V = \bigcup_p N_1.p$ satisfy the conditions* (Lip0), (Lip1), *and* (Lip2).

Proof. First, let $(x, x') \in N \cap (V \times X)$. If $x = x'$ then $(x, x') \in 1_X \subseteq N_0$. Otherwise, there is some p with $(x, x') \in N.p$. If $p \notin \{\mu(x), \mu(x')\}$, then $(x, x') \in N_0.p$. If $p = \mu(x)$ then $(x, x') \in N_1.p$. Otherwise $\mu(x) \neq p = \mu(x')$. Therefore (local) implies $\mu(x) = \bot$, contradicting $x \in V$. This proves (Lip0).

Condition (Lip1) is immediate. As for (Lip2), let $(x, x') \in N_0$. Clearly, $(x, x') \in N$. Assume there is p with $(x, x') \in N_0.p$. Then (local) implies $\mu(x) = \mu(x')$. The same holds if $x = x'$. The condition $\mu(x) = \mu(x')$ implies $(x, x') \notin V \times W$. This proves (Lip2). □

In order to apply Theorem 5.2, it remains by Lemma 5.3 to ensure commutation rule (Lip3). In the present setting, condition (Lip3): $N_0; N_V \subseteq N_V^*; N_0$ is the requirement that every step (N_V) of the owner of the mutex is a "left-mover" with respect to any step (N_0) of a non-owner. This follows, e.g., if non-owners of the mutex do not inspect or modify the shared variables that can be inspected or modified by owners of the mutex.

The main application is as follows. In thread systems, one of the ways to ensure mutual exclusion is by means of *mutexes*, e.g., in POSIX threads (pthreads). Recall from [8] that a mutex is (or can be regarded as) a shared variable of type *Process*$\cup\{\bot\}$. We say that process p *owns* mutex `mu` iff `mu` $= p$. Mutex `mu` is said to be *free* iff `mu` $= \bot$; this holds initially. The value (owner) of a mutex can only be modified by the commands `lock` and `unlock`. If the mutex is free, a process can become the owner by executing `lock(mu)`. If process p owns the mutex, it can release it be executing `unlock( mu)`. It follows that `mu` is a state function that satisfies the above condition (local) for μ. We can thus apply the above theory with $\mu =$ `mu`. Removal of the episodic states amounts to abstraction from the internal states of mutex guarded regions, i.e., to making the mutex guarded regions atomic.

It is folk knowledge that the code between the acquisition and release of a mutex can be considered atomic if the shared variables accessed in that code are accessed by each process only while holding the mutex. Indeed, programmers routinely use this intuition to reason informally about their parallel programs. In [27], this folk knowledge is described in terms of structured code locking and structured data locking. These conditions indeed imply (Lip3). A treatment of these conditions, however, is out of the scope of the paper.

6. Completeness and methodology

In [14], 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*. The result was transferred to refinement in CSP-like concurrency with failure semantics by [26].

In this section, we discuss semantic completeness and methodology in our setting of possibly nonterminating concurrent specifications. For the ease of reading, we provide section numbers for the various definitions, as far as given in this paper.

Our setting of possibly nonterminating, concurrent specifications follows [1]. There, it was proved that, if specification K is *machine closed* (2.3) and specification L has *finite invisible nondeterminism* and *internal continuity*, then every strict simulation $F : K \twoheadrightarrow L$ (4.1) can be factored over a forward simulation (4.2), a backward simulation (5.1), and a refinement mapping (3.1).

In [16], because we could not satisfy the technical assumption of finite invisible nondeterminism, we replaced the prophecy variables of [1] by eternity extensions (5.2). In [17], we proved that every strict simulation that *preserves quiescence* can be factored over a forward simulation, an eternity extension, and a refinement mapping.

This result was strengthened considerably in [23] where we eliminated the conditions of strictness and preservation of quiescence, using the clocking extension of (4.3) and the stuttering forward simulations of (4.4). We summarize the main results on semantic completeness as follows.

Theorem 6.1. *Let K be a specification. The clocking extension $ivf : K \multimap cl(K)$ can be composed with an eternity extension $e : cl(K) \multimap E$ such that:*
(a) *For every strict simulation $F : K \multimap L$ there is a refinement mapping $f : E \multimap L$ with $(ivf ; e ; f) \subseteq F$.*
(b) *For every simulation $F : K \multimap\!\!\!\!\multimap L$ there is a stuttering forward simulation $g : E \multimap\!\!\!\!\multimap E'$ and a refinement mapping $f : E' \multimap L$ with $(ivf ; e ; g ; f) \subseteq F$.*

Note that the eternity extension $e : cl(K) \multimap E$ *does not* depend on F and L, and that it works for both (a) and (b). 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. For application to a given F and L, one scales E down, but, even so, we have come to regard the method as a kind of sledge hammer that is to be applied sparingly and with 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 [19], 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 along the lines set out in 3.3, because condition (F2) is an invitation for sloppy reasoning but splitting simulations are not often applicable. 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 [20], 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 [21] of algorithm C2 of Haldar and Vidyasankar.

7. Comparison with other formalisms

The UNITY book of [10] postulates weak fairness for all separate instructions. Based on this, the book develops an interesting logic for progress that can also be transferred to our setting, cf. [18, section 7]. In our setting, weak fairness for all separate instructions is one of the possible choices for the supplementary property.

In the fair transition systems of [36], the supplementary property is given by specifying sets of weakly fair and strongly fair transitions. This is usually enough for abstract and concrete specifications, but it is occasionally not flexible enough for intermediate specifications. The temporal logic of this book is much more expressive than ours: it also contains **until** and **next** operators and *past temporal operators*. The book also uses a more expressive programming syntax.

Most of the formalisms proposed only allow state spaces that are spanned by programming variables. In our formalism, the state space can be an arbitrary set. This enables us to consider auxiliary specifications with state spaces that are subsets of spaces spanned by programming variables. In [31,36], programming variables are called *flexible* variables and they require a separate logic. For us, programming variables are just components of the state, cf. [15], and are, strictly speaking, not variables at all. In this way, we do not need a separate logic. This does not remove problems, but only places them on a different level.

TLA of [31] is essentially untyped. In our view, the choice between typed and untyped formalisms is mainly a matter of taste, but our preference is to distinguish the various state spaces involved in refinement relations, and not to put all variables together in one universe.

Following [1,31], we do allow non-machine-closed specifications [16,20]. Such specifications are occasionally useful as intermediate specifications in refinement proofs. See Section 5.3 for an easy example.

The action systems' formalism of [5] and the abstract state machines of [44] accommodate abortion and explicit termination. The paper [5] also treats parallel composition, which for our formalism is still a matter of future work. The paper [44] introduces a variation of our eternity extension and proves a semantic completeness result similar to Theorem 6.1(a) above.

In the fields of labelled transition systems and process algebras, there are numerous related notions of simulations and bisimulations, e.g., [40,43,26,35,12,46,4,13,41]. Usually, the emphasis is on the transitions, which are communications. With us the emphasis is on the states, while the transitions are only modifications of the state. These formalisms usually allow no or only very restricted liveness properties. Some of them focus on bisimulation rather than refinement. Attie [4] comes closest, but restricts the supplementary property to "complemented pairs liveness", i.e., formulas of the form $\forall\, i : (\Box\Diamond R_i \;\Rightarrow\; \Box\Diamond G_i)$, a form of strong fairness. Manolios [37] also comes close to our approach. He investigates refinement of (unlabelled) transition systems without supplementary properties. This means that he only treats minimal fairness.

8. On using proof assistants

In the course of several verification projects of concurrent algorithms, we were forced to use theorem provers, first NQTHM [9], later PVS [42], primarily for the administration of proof obligations. Our main proof scripts are available at [24].

In each of our case studies [16,18,20,21,3], the complexity of the many case distinctions in the proofs is such that hand-written proofs are not reliable enough. When developing proof scripts for inspection by other readers, we felt forced to emphasize the specifications that were to be proved, and to separate the treatment of the meta-theory from the application at hand. On the other hand, using a prover to prove the correctness of concurrent algorithms turns out to be an excellent learning school for the intricacies of these algorithms. The adoption of refinement was forced upon us when we needed prophecies about the developing transactions in the serializable database interface of [16].

Moore taught me in 1986 that one must never unnecessarily introduce axioms into a theorem prover. We have therefore always also encoded and proved the complete meta-theory with the prover. In particular, the theories developed in [17,19,23] are all proved with PVS. This reliance on the classical logic, as encoded in a general theorem prover as PVS, is at the basis of our reluctance to rely on special purpose logics. To quote from [38, p. 37], "In our opinion, it is better to avoid modifying the logic if at all possible, because there are many temptations to modify the logic, and it would be very difficult to keep them compatible". The benefits of treating the meta-theory with the prover were not as large as for the case studies. The main lesson learned here was that elegance is profitable because clumsy proof efforts take much more time than elegant ones. The prover played the role of a never tiring antagonist, always skeptical and yet always prepared to be convinced.

References

[1] M. Abadi, L. Lamport, The existence of refinement mappings, Theor. Comput. Sci. 82 (1991) 253–284.
[2] G.R. Andrews, Concurrent Programming, Principles and Practice, Addison-Wesley, Menlo Park, etc, 1991.
[3] A.A. Aravind, W.H. Hesselink, A queue based mutual exclusion algorithm, Acta Inf. 46 (2009) 73–86.
[4] P.C. Attie, Liveness-preserving simulation relations, in: ACM Symposium on the Principles of Distributed Computing, PODC'99, 1999, pp. 63–72.
[5] R.J. Back, J. von Wright, Trace refinement of action systems, in: CONCUR '94, Concurrency Theory, 5th International Conference, Uppsala, Sweden, August 22-25, 1994, Proceedings, in: LNCS, vol. 836, Springer, 1994, pp. 367–384.
[6] R.J.R. Back, A method for refining atomicity in parallel algorithms, in: PARLE'89 Parallel Architectures and Languages Europe, in: LNCS, vol. 366, Springer, New York, 1989, pp. 199–216.
[7] R.J.R. Back, K. Sere, Stepwise refinement for action systems, in: J.L.A. van de Snepscheut (Ed.), Mathematics of Program Construction, in: LNCS, vol. 375, Springer, New York, 1989, pp. 115–138.
[8] A. Birrell, J.V. Guttag, J. Horning, R. Levin, Synchronization primitives for a multiprocessor: A formal specification, Oper. Syst. Rev. 21 (1987) 94–102.
[9] R.S. Boyer, J.S. Moore, A Computational Logic Handbook, Academic Press, Boston, 1997.
[10] K.M. Chandy, J. Misra, Parallel Program Design, A Foundation, Addison-Wesley, 1988.
[11] E. Cohen, L. Lamport, Reduction in TLA, in: D. Sangiorgi, R. de Simone (Eds.), CONCUR '98, in: LNCS, vol. 1466, Springer, New York, 1998, pp. 317–331.
[12] R. de Nicola, F. Vaandrager, Three logics for branching time bisimulation, J. ACM 42 (1995) 458–487.
[13] D. Griffioen, F. Vaandrager, A theory of normed simulations, ACM Trans. Comput. Log. 5 (2003) 1–33.
[14] J. He, C.A.R. Hoare, J.W. Sanders, Data refinement refined, in: B. Robinet, R. Wilhelm (Eds.), ESOP 86, in: LNCS, vol. 213, Springer, New York, 1986, pp. 187–196.
[15] W.H. Hesselink, Programs, Recursion and Unbounded Choice, Predicate Transformation Semantics and Transformation Rules, in: Cambridge Tracts in Theoret. Comput. Sci., vol. 27, Cambridge University Press, Cambridge, 1992.
[16] W.H. Hesselink, Using eternity variables to specify and prove a serializable database interface, Sci. Comput. Program. 51 (2004) 47–85.
[17] W.H. Hesselink, Eternity variables to prove simulation of specifications, ACM Trans. Comput. Log. 6 (2005) 175–201.
[18] W.H. Hesselink, Refinement verification of the lazy caching algorithm, Acta Inform. 43 (2006) 195–222.
[19] W.H. Hesselink, Splitting forward simulations to cope with liveness, Acta Inform. 42 (2006) 583–602.
[20] W.H. Hesselink, A criterion for atomicity revisited, Acta Inform. 44 (2007) 123–151.
[21] W.H. Hesselink, A challenge for atomicity verification, Sci. Comput. Program. 71 (2008) 57–72.
[22] W.H. Hesselink, Simulation refinement for concurrency verification, Electron. Notes Theor. Comput. Sci. 214 (2008) 3–23.
[23] W.H. Hesselink, Universal extensions to simulate specifications, Inform. Comput. 206 (2008) 108–128.
[24] W.H. Hesselink, Mechanical verification projects, 2009. Available at: www.cs.rug.nl/~wim/mechver/index.html.
[25] B. Jonsson, Simulations between specifications of distributed systems, in: J.C.M. Baeten, J.F. Groote (Eds.), CONCUR '91, in: LNCS, vol. 527, Springer, New York, 1991, pp. 346–360.
[26] M.B. Josephs, A state-based approach to communicating processes, Distrib. Comput. 3 (1988) 9–18.
[27] S. Kleiman, D. Shah, B. Smaalders, Programming with Threads, Prentice Hall, 1996.
[28] P. Ladkin, L. Lamport, B. Olivier, D. Roegel, Lazy caching in TLA, Distrib. Comput. 12 (1999) 151–174.
[29] L. Lamport, A simple approach to specifying concurrent systems, Commun. ACM 32 (1989) 32–45.
[30] L. Lamport, Critique of the lake arrowhead three, Distrib. Comput. 6 (1992) 65–71.
[31] L. Lamport, The temporal logic of actions, ACM Trans. Program. Lang. Syst. 16 (1994) 872–923.
[32] L. Lamport, Processes are in the eye of the beholder, Theor. Comput. Sci. 179 (1997) 333–351.
[33] L. Lamport, F.B. Schneider, Pretending atomicity, Research Report 44, Digital Equipment Corporation, SRC, 1989.
[34] R.J. Lipton, Reduction: A method of proving properties of parallel programs, Commun. ACM 18 (1975) 717–721.
[35] N. Lynch, F. Vaandrager, Forward and backward simulations, part I: Untimed systems, Inform. and Comput. 121 (1995) 214–233.
[36] Z. Manna, A. Pnueli, Temporal verification of reactive systems: Safety, Springer, New York, 1995.
[37] P. Manolios, A compositional theory of refinement for branching time, in: D. Geist, E. Tronci (Eds.), Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, Proceedings, in: LNCS, vol. 2860, Springer, 2003, pp. 304–318.
[38] J. McCarthy, Circumscription — a form of non-monotonic reasoning, Artificial Intelligence 13 (1980) 27–39.
[39] R. Milner, An algebraic definition of simulation between programs, in: Proc. 2nd Int. Joint Conf. on Artificial Intelligence, British Comp. Soc, 1971, pp. 481–489.
[40] R. Milner, A Calculus of Communicating Systems, in: LNCS, vol. 92, Springer, 1980.
[41] S. Nejati, A. Gurfinkel, M. Chechik, Stuttering abstraction for model checking, in: Third IEEE International Conference on Software Engineering and Formal Methods, SEFM 2005, 2005, pp. 311–320.
[42] S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringer-Calvert, PVS version 2.4, system guide, prover guide, PVS Language Reference, 2001. http://pvs.csl.sri.com.
[43] D. Park, Concurrency and automata on infinite sequences, in: Theoret. Comput. Sci., in: LNCS, vol. 104, Springer, 1981, pp. 167–183.
[44] G. Schellhorn, Completeness of ASM refinement, Electron. Notes Theor. Comput. Sci. 214 (2008) 25–49.
[45] F.B. Schneider, Introduction, Distrib. Comput. 6 (1) (1992) 1–3.
[46] R.J. van Glabbeek, W.P. Weijland, Branching time and abstraction in bisimulation semantics, J. ACM 43 (1996) 555–600.