Acta Informatica (2007) 44:123–151
DOI 10.1007/s00236-007-0044-1

ORIGINAL ARTICLE

A criterion for atomicity revisited

Wim H. Hesselink

Received: 6 July 2006 / Accepted: 12 March 2007 / Published online: 11 April 2007

Abstract Concurrent and reactive programs are specified by their behaviours in the presence of a nondeterministic environment. In a natural way, this gives a specification (*ARW*) of an atomic variable in the style of Abadi and Lamport. Several implementations of atomic variables by lower level primitives are known. A few years ago, we formulated a criterion to prove the correctness of such implementations. The proof of correctness of the criterion itself was based on Lynch's definition of atomicity by serialization points. Here, this criterion is reformulated as a specification *HRW* in the formal sense. Simulations from *HRW* to *ARW* and vice versa are constructed. These now serve as a constructive proof of correctness of the criterion. Eternity variables are used in the simulation from *HRW* to *ARW*. We propose so-called gliding simulations to deal with the problems that appear when occasionally the concrete implementation needs fewer steps than the abstract specification.

1 Introduction

Concurrent and reactive programs are preferably specified by their behaviours during execution in the presence of a nondeterministic environment. Indeed, even when termination is desirable, their preconditions and postconditions are usually not very informative.

These programs are designed in a hierarchical way, in which the atomic steps at a certain level are implemented by several steps at a lower level. This makes the use of the word "atomic" at the higher level questionable. There are two ways to justify it. One can either *define* the high-level atomicity by means of serialization points [16], or take the refinement approach and prove that all visible behaviours of the implementation are visible behaviours of the high-level program [1].

Before committing to either of these approaches, let us concentrate on the atomicity of read–write variables. There are several algorithms that implement read–write variables by

W. H. Hesselink (✉)
Department of Mathematics and Computing Science,
University of Groningen, P. O. Box 800, 9700 AV Groningen, The Netherlands
e-mail: w.h.hesselink@rug.nl
URL: http://www.cs.rug.nl/∼wim

means of lower-level primitives, e.g. by Vitányi and Awerbuch [22], Bloom [2], Vidyasankar [23], and Haldar and Subramanian [12]. In all these cases, the correctness is far from obvious. The proposers of the algorithms apply different methods to argue the correctness of their algorithms.

In [3], we used the definition of atomicity by serialization points to develop an assertional criterion for atomicity of read–write variables. The criterion worked by adding history variables to the processes in a certain way, and then proving some invariants for the resulting system, see Theorem 1 in Sect. 3.5 below. In [3], we applied the criterion to verify the algorithms of Bloom and Vitanyi-Awerbuch. In [5], we applied it to the algorithm of Haldar and Subramanian. It can also be used for Vidyasankar's algorithm.

Although successful, we now regard the criterion as an unsatisfactory recipe, and we prefer the refinement approach over the definition of atomicity by serialization points. We therefore rephrase the criterion as an assertion that the "natural" specification (*ARW*) of an atomic variable is implemented by a specification *HRW* that encodes the clauses of the criterion. In principle, this implementation relation could be proved by reduction to Theorem 1, or by rephrasing its proof. From such a proof, however, not much new would be learned.

This paper is therefore devoted to refinements proofs that *HRW* implements *ARW*, and vice versa. Such proofs are challenges to the refinement methods developed in [1] and elsewhere. Indeed, it seems that the method of [1] is not strong enough, since it only allows finite invisible nondeterminism, whereas *HRW* uses infinite invisible nondeterminism. Instead of the prophecy variables of [1], we therefore apply the eternity variables introduced in [6,7].

The refinement relations between *HRW* and *ARW* and vice versa thus seem to require some kind of prescient variables, prophecy or eternity. These relations also show that sometimes the concrete specification does fewer steps than the abstract specification. We propose *gliding simulations* to deal with the technical complications that arise in this way. In our opinion, these gliding simulations are stronger and more convenient than the solutions proposed in [1,14].

The present case study thus serves as an excellent illustration of advanced cases of refinement relations. The specifications are quite small. Yet, the verifications of the various refinement relations are complicated enough to justify the application of a proof assistant. We verified all results with the theorem prover PVS of [21].

1.1 Overview

In Sect. 1.2, we sketch the position of refinement calculus in the field of concurrency verification. Read–write atomicity is dealt with in Sect. 1.3.

In Sect. 2, we present our version of the specification and implementation formalism of [1]. In Sect. 3, we specialize to specifications of reactive systems. Here we introduce protocols to distinguish the responsibilities of the system and its environment. Then we zoom in on read–write systems and present the specifications *ARW* and *HRW* mentioned above.

In Sect. 4, we present simulations, the main tool for proving implementation relations. After the well-known refinement mappings and forward simulations, we present eternity extensions and gliding simulations. The latter type of simulations is needed when the implementing specification takes fewer steps than the abstract one. In our case, this occurs between intermediate specifications to move computation steps to their scheduled positions. In Sect. 5, we construct a simulation from *HRW* to *ARW* as a composition of six simpler simulation relations.

In Sect. 6, a second gliding simulation is used to prove that, conversely, *ARW* implements *HRW*. In other words, the two specifications are equivalent. The verification with the theorem prover PVS is briefly discussed in Sect. 7. Conclusions are drawn in Sect. 8.

1.2 Specifying and verifying concurrent and reactive programs

There are two schools in the way behaviours of concurrent and reactive programs are described. In the process algebra school of Milner [18] and Hoare [11], the behaviours are characterized as sequences or trees of transitions. This is good for communication protocols, but less well suited for algorithms with complicated states and state transitions. In this article, we follow the school of Abadi and Lamport [1,13], Manna and Pnueli [19,20], in which behaviours are characterized as infinite sequences of states.

Since the complete state usually contains too many variables, we have to be explicit about the variables that are relevant for the specification. These are called the visible variables. The combination of all visible variables is the visible state. In this way, we also get visible behaviours: infinite sequences of visible states.

One specification implements another if and only if all visible behaviours of the first specification are also visible behaviours of the second specification. Although they may change roles, we refer to the implementing specification as the *concrete* one, and to the implemented specification as the *abstract* one.

Arguing about behaviours of a given specification is very inconvenient and error prone. This is the reason that methods have been developed to prove implementation relations, in which consideration of the behaviours is eliminated as much as possible. These methods are based on concepts like invariants and simulation.

The idea of refinement calculus is to treat specifications, algorithms and programs under the same heading, viz. as specifications of behaviours, the main difference being that programs are closer to executable code than general specifications.

1.3 Read–write atomicity

One of the central concepts in concurrency is the grain of atomicity. We need to know that a certain object or component acts according to its specification in a single indivisible step, although we know that internally this step may be implemented in a complicated way by many steps, and that the component may be serving different requests concurrently. The easiest case of such an object is the read–write object.

In her book [16], Lynch defines atomicity by means of serialization points. Roughly speaking, an operation is defined to be *atomic* iff, logically, it takes place at some undetermined moment between the start and the end of the operation. In particular, the operation is allowed to take some time, finite but arbitrary. This definition was used in [3] to develop an assertional criterion for atomicity of read–write objects.

Yet, the definition of atomicity of [16] seems too complicated and verbose for such an elementary concept as atomicity. In this paper, therefore, we give an elementary specification *ARW* of an atomic read–write register in the style of [1]. The criterion for atomicity mentioned above suggests a more complicated specification *HRW*. The soundness of the criterion is the theorem that *HRW* implements *ARW*. The criterion is useful since for some specifications it is easier to prove that they implement *HRW* than that they implement *ARW*.

2 Specifications

In this section, we present our formalism for specifications, a syntactic variation of [1], also inspired by TLA [13]. 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.

A binary relation on a set X is identified with the set of pairs that satisfy the relation; this is subset of the Cartesian product $X \times X = X^2$. We write $\mathbf{1}$ for the identity relation of X. If A is a binary relation on X and Q is a predicate on X, its *weakest precondition* is defined by $wp(A, Q) = \{x \mid \forall\, y : (x, y) \in A \;\Rightarrow\; y \in Q\}$. A special case is $disabled(A) = wp(A, \emptyset)$.

2.2 Temporal formulas

We write X^ω for the set of 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 drop 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 infinite sequences, its complement in X^ω is denoted $\neg P = (X^\omega \setminus P)$. The sets $\Box P$ (always P), and $\Diamond P$ (sometime P) are defined by

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

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 A on X, we define the subset $[\![\, A \,]\!]_2$ of X^ω to consist of the sequences that start with an A-transition. So we have

$$\begin{aligned} xs \in [\![\, U \,]\!] &\;\equiv\; xs(0) \in U, \\ xs \in [\![\, A \,]\!]_2 &\;\equiv\; (xs(0), xs(1)) \in A. \end{aligned}$$

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 behaviours 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 conjunction and disjunction of properties is a property. $[\![\, U \,]\!]$ is a property for every $U \subseteq X$. If A is a reflexive relation on X, then $\Box\, [\![\, A \,]\!]_2$ 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 and programs

As announced, we use a syntactic variation of the formalism of [1]. A *specification* is defined to be a tuple $K = (X, Y, N, P)$ where X is the state space, $Y \subseteq X$ is the set of initial states, $N \subseteq X^2$ is the next-state relation and P is the supplementary property [1]. Relation N is required to be reflexive in order to allow stutterings. P is a subset of the set X^ω of the infinite sequences of states, which is required to be a property.

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

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

The property rules imply that $Beh(K)$ is always a property. For a specification $K = (X, Y, N, P)$, we write $states(K) = X$, $start(K) = Y$, $step(K) = N$, $prop(K) = P$.

When presenting a specification, we use a program-like notation, where 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 at 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

(W) **whenever**
[] $U_i \ \rightarrow \ A_i$;
end

denotes a next-state relation that is the union of the identity relation $\mathbf{1}$ with the sets $A_i \cap (U_i \times X)$. So, it is a nondeterminate choice between the guarded commands $U_i \rightarrow A_i$, which are taken atomically and repeatedly. 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 blocks 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 suplementary properties as weak as possible. They are mainly used to express progress conditions.

2.4 Visibility and implementation

A *visible specification* is a pair (K, v) where K is a specification and v is a function from $states(K)$ to some set of observations. Then v is called the *observation function*. The sequences $v \circ xs$ where xs ranges over the behaviours of K, are called *observed behaviours*. A sequence of observations is called a *visible behaviour* of (K, v) if it has a stuttering that is an observed behaviour. So, vs is a visible behaviour iff $vs \preceq v \circ xs$ for some behaviour xs. Not all visible behaviours are observed behaviours, see the example below.

Let (K, v) and (L, w) be visible specifications with observation functions to the same set of observations. Then (K, v) is said to *implement* (L, w) iff every visible behaviour of (K, v) is a visible behaviour of (L, w), cf. [1].

Example Let $m \in \mathbb{N}$ be positive. Consider the specification $K(m)$ given by

var j : *Nat* := 0 ;
whenever
[] *true* → j := (j + 1) **mod** *m* ;
end ;
prop: j changes infinitely often.

Note that we keep the stuttering steps implicit. We have $\mathit{states}(K(m)) = \mathbb{N}$, $\mathit{start}(K(m)) = \{0\}$, relation $\mathit{step}(K(m))$ consists of the pairs (j, k) with $k = (j+1)$ **mod** m or $j = k$ (stuttering). The supplementary property is $\mathit{prop}(K(m)) = \Box\Diamond[\![\, \mathtt{j} \neq \mathtt{j}^+ \,]\!]_2$, where $\mathtt{j}^+$ refers to the value of j in the next state.

Taking $m = 5$, the behaviours of $K(5)$ are the stutterings of $vs = (01234)^\omega$. The other executions are the stutterings of the infinite sequences $(01234)^n 0 \ldots k^\omega$ with $n, k \in \mathbb{N}$ and $k < 5$. These are not behaviours since eventually j is constant.

We now take the observation function $v(j) = j$ **div** 2. The observed behaviours are the stutterings of $(00112)^\omega$. The visible behaviours are the stutterings of $(012)^\omega$. Therefore $(K(5), v)$ has the same visible behaviours as $(K(3), id)$, where *id* is the identity function. It follows that $(K(5), v)$ and $(K(3), id)$ implement each other. □

2.5 Machine closure, invariants, and inductive subsets

A specification K is called *machine closed* [1] iff every finite prefix of any initial execution can be extended to a behaviour. The first case of a nonmachine closed specification was in [14]. Below, we encounter specifications that are nonmachine closed in our treatment of prophecies and eternity variables. In a nonmachine closed specification, we need to distinguish between states that are reachable and those that occur in behaviours. This is the reason for the following nonstandard definitions.

A state x of K is said to be *occurring* iff $x = xs(n)$ for some behaviour xs of K and some number n. In this note, we define a subset J of $\mathit{states}(K)$ to be an *invariant* of K iff J contains all occurring states. A subset J is called *inductive* for K if $\mathit{start}(K) \subseteq J$ and $J \subseteq \mathit{wp}(N, J)$. It is easy to verify that every inductive subset of $\mathit{states}(K)$ is an invariant (in the present sense).

Example Consider the specification

var j : *Int* := 0 ;
whenever
[] j = 0 → **choose** j **with** j > 0 ;
[] *true* → j := j − 2 ;
end ;
prop: j = 0 holds infinitely often.

A command of the form **choose** v **with** Q has the obvious meaning of a nondeterministic choice of v in the type of v, constrained by predicate Q. We omit the clause "**with** Q" when Q is identically true.

This specification is not machine closed. All odd numbers are reachable, but not occurring because of the supplementary property. The predicates $\mathtt{j} \geq 0$ and j **mod** 2 = 0 are invariants of this specification, but neither is inductive. □

3 Specification of reactive systems

When we want to specify a reactive system in our state-based linear-time formalism, the natural way is to specify the environment the system has to communicate with as a very nondeterminate component of the system, which is not to be implemented.

This requirement needs to be formalized in such a way that the implementer cannot cheat us by letting the system perform some actions that must be reserved to the environment. For instance, in the administration of a bank, the system is not allowed to generate requests for money transfer; such requests form a privilege of the authorized clients. Indeed, a system that invents values to be written may be regarded as a malicious virus. We now provide a formalization of the separation of responsibilities between environment and system.

3.1 Protocols and reactive specifications

We define a *protocol* to be a tuple $E = (X_1, Y_1, N_e, N_s, v_1)$ such that X_1 is a set (the state space), $Y_1 \subseteq X_1$ (the set of initial states), and N_e and N_s two binary relations over X_1, and $v_1 : X_1 \to Obs$ is an observation function. Relation N_e is required to be reflexive; it is called the *environment relation* and is often denoted $env(E)$. Relation N_s is called the *system relation* and often denoted $sys(E)$. In concurrent systems, both environment and system are themselves again split in component relations (processes) $env.p$ and $sys.p$.

The idea is that X_1 is the state space as far as relevant for the environment and every implementation of the system. N_e defines the steps of the environment. N_s restricts the possibilities of the system as liberal as possible. In particular, it serves to prohibit the system from taking actions that are reserved for the environment. The protocol does not yet specify the behaviours of the system in its environment. This is the reason that a supplementary property is missing.

In the reactive specification to be defined below, we add specification variables to describe the allowed behaviours. These specification variables span a component X_2 of the state space, and have initial values in a subset $Y_2 \subseteq X_2$. We postulate an environment condition (EC) to express that the environment can do everything allowed by N_e and does not change the specification variables, and a protocol constraint (PC) to express that the system keeps the protocol.

A *reactive specification over protocol* $E = (X_1, Y_1, N_e, N_s, v_1)$ is defined to be a visible specification (K, v) with $K = (X, Y, N, P)$ where X is a Cartesian product $X_1 \times X_2$ and $Y = Y_1 \times Y_2$ (so that $Y_2 \subseteq X_2$). The step relation N is required to satisfy the conditions

$$\begin{array}{ll} \text{(EC)} & N_e \times \mathbf{1} \subseteq N\ , \\ \text{(PC)} & (x, y) \in N_i \ \Rightarrow\ (x_1, y_1) \in N_s, \\ \text{where} & N_e \times \mathbf{1} = \{(x, y) \mid (x_1, y_1) \in N_e \wedge x_2 = y_2\}\ , N_i = N \setminus (N_e \times \mathbf{1}). \end{array}$$

Here, we use the convention that $x \in X_1 \times X_2$ always satisfies $x = (x_1, x_2)$. The supplementary property P is just a property over state space X. The observation function v is given by $v(x) = v_1(x_1)$.

Usually, the components X_1 and X_2 are each spanned by shared and private variables. The sets Y_1 and Y_2 indicate their sets of initial states. The steps in $N_e \times \mathbf{1}$ are the environment steps of K, as restricted by the *environment condition* (EC). The steps in the *implementation relation* N_i are the system steps, which need to be in accordance with the *protocol constraint* (PC).

Given protocol E, the reactive specification is completely determined by X_2, Y_2, N_i, and P. Therefore, in practice, the reactive specification is given by declaring the *additional*

variables that span X_2, with their initial values as given by Y_2, the *system step relation* N_i, and the supplementary property P.

Example Consider a system where the environment can tick (write t), and ask a question (write q) about the history. The system acknowledges the ticks by writing n, and answers the questions by choosing between n and y. The state of the system is just the latest symbol written. The state space X_1 therefore is the set {t, q, n, y}. The environment can only tick or ask a question when the system has responded. We therefore choose the initial set $Y_1 = \{\mathtt{n}\}$ and specify the environment relation N_e to consist of the pairs (x, y) with $x = y$, or $x \in \{\mathtt{n}, \mathtt{y}\}$ and $y \in \{\mathtt{t}, \mathtt{q}\}$. The system only reacts to steps of the environment. Therefore the system relation N_s consists of the three pairs (t, n), (q, n), and (q, y). This concludes the protocol.

We can now give a reactive specification for a system that answers y if the number of ticks after the previous question is at least 7, and n otherwise. For this purpose, we introduce a specification variable $k \in \mathbb{N}$ to count the ticks. So we take $X_2 = \mathbb{N}$ with initial set $Y_2 = \{0\}$. We then get to the implementation relation

$$\begin{aligned} N_i = {} & \{((\mathtt{t}, k), (\mathtt{n}, k+1)) \mid k \in \mathbb{N}\} \\ & \cup \{((\mathtt{q}, k), (\mathtt{n}, 0)) \mid k < 7\} \cup \{((\mathtt{q}, k), (\mathtt{y}, 0)) \mid k \geq 7\} \,. \end{aligned}$$

We choose the supplementary property $P = \Box\Diamond[\![\{\mathtt{n}, \mathtt{y}\}]\!]$. This means that the system always eventually answers.

Of course, the infinite state space X_2 is not necessary for implementations. The reactive specification can be implemented by means of a three-bit counter.

Now consider an alternative implementation *Clock*, in which the system also creates and answers its own ticks t and questions q, e.g., when the environment is idle for some time. Every behaviour of the combination of *Clock* with the environment is a behaviour of the reactive specification. Yet, this is not the intention, and indeed this is ruled out by the protocol constraint (PC). □

3.2 Concurrency

We use specifications to model concurrent systems with processes that act on shared variables and also have private variables. Let *Process* be the set of processes or process identifiers. In this setting, a state of the system is given by the values of all variables and the state space X is the set of all states.

We declare shared variables with the keyword **var** and write them in type writer font. Private variables are slanted and declared with the keyword **privar**. Outside of the programs, a private variable v of process p is denoted by $v.p$. Indeed, formally, a private variable is treated as a (modifiable) function from process identifiers to values. The type *PrivState* of the private states is spanned by the private variables, the type *SharedState* is spanned by the shared variables. Then global state space X is the Cartesian product

$$X = \mathit{SharedState} \times (\mathit{Process} \to \mathit{PrivState}) \,.$$

3.3 Read–write systems

In the remainder of this section, we illustrate the formalism by specifying an atomic register in a system with several concurrent processes that can read and write the register. We first give the protocol.

Let *Item* be the set of values to be written and read. Every process p has a private variable $\mathit{arg}.p$ for the value to be written, a variable $\mathit{result}.p$ for the value to be read, and a location

pointer $pc.p$. As announced above, the variables *result*, *arg*, and *pc* are slanted to indicate that they are private.

privar *arg*, *result* : *Item* := *item0* ;
privar *pc* : *Nat* := 0 ;
var `out` : *Item* := *item0* .

Since the values to be written must not be invented by the writing process, and since the need to read should not be generated by the reading process, we model an environment that calls the reading and writing routines. Since the environment can use the values read only after the read action has completed, we model this usage by a separate action of copying *result* to the shared variable `out`. The system for process p is allowed to modify $result.p$ and $pc.p$, but only if $pc.p \neq 0$. The protocol for process p is thus the following nondeterministic choice.

$env.p$: [] $pc = 0 \rightarrow$ **choose** *arg* ; $pc := 20$; // write request
[] $pc = 0 \rightarrow$ $pc := 50$; // read request
[] $pc = 0 \rightarrow$ `out` := *result* ;
$sys.p$: [] $pc \neq 0 \rightarrow$ **choose** *result* , *pc* .

We choose *arg* and `out` to be the visible variables. We regard *result* as not visible since a process can only use the value read after completion of the read action. Indeed, the value read may be outdated when it becomes available. This concludes the protocol.

3.4 The atomic read–write system

The atomic read–write system is a reactive specification over the protocol, which uses a single shared variable `reg` to transfer the value that has to be written and to be read. The system step relation determines how processes p read from and write to `reg`:

var `reg` : *Item* := *item0* ;
`Wr`.p : [] $pc = 20 \rightarrow$ `reg` := *arg* ; $pc := 0$;
`Rd`.p : [] $pc = 50 \rightarrow$ *result* := `reg` ; $pc := 0$;
prop: *TERM* : $(\forall\ p \in Process : \Box\Diamond[\![\ pc.p = 0\ ]\!])$.

The protocol constraint is satisfied, since `Wr`.p and `Rd`.p restrict to the protocol variables $arg.q$, $result.q$, $pc.q$, and `out` as special cases of $sys.p$. Indeed, restricted in this way, they only modify $result.p$ and $pc.p$, and that only when $pc.p \neq 0$.

We further specify the progress property *TERM* that writing and reading always terminates, i.e., $\Box\Diamond[\![\ pc.p = 0\ ]\!]$ for all processes p.

The state space AX of the atomic read–write system consists of the two shared variables `reg` and `out` and the private variables *arg*, *result*, and *pc*. Here *arg* and *result* are functions from the set of processes (process identifiers) to *Item*, whereas *pc* is a function from processes to natural numbers. Initially, `reg` and `out` and all values of *result* and *arg* are equal to *item0*, while all values of *pc* are 0. This concludes the description of the reactive specification *ARW*.

Every implementation of *ARW* is supposed to have the same protocol, but to supply implementations for the shared variable `reg` and the commands `Wr` and `Rd`, possibly consisting of several atomic commands and a different state space. We give an example in the next subsection.

3.5 A criterion for atomicity

In [3], we developed an assertional criterion for atomicity. We showed that, in order to prove atomicity of a read–write object, it suffices to add history variables to the processes in a certain way and then prove some invariants.

We first present this criterion and reformulate it as a specification *HRW* in the present formalism. Correctness of the criterion is then the assertion that *HRW* implements specification *ARW* of Sect. 3.4. The proof of this implementation relation is postponed to Sect. 5.

According to [3], every process should get private integer history variables *start* and *sqn*, and there should be a shared history variable `masq`, initialized in an arbitrary way. Furthermore, there is a shared history variable `snlist` that holds a list of integer, which is initially empty.

Every process updates `masq` at the end of every operation by setting `masq` := max(`masq`, *sqn*). In every operation of a process, it sets its private variables *start* and *sqn* precisely once as described now. Whenever a process starts an operation, it sets its *start* := `masq`. In every writing operation, a writing process chooses a value for *sqn*, attaches it to the item to be written and adds it to the list `snlist`. Every process that copies an item, also copies the number attached to it. When a reading process interprets a value as the item read, it sets its *sqn* equal to the attached number. Given this setting, we proved

Theorem 1 *Assume that, for every process p, every write action of p has the postcondition start.p* $<$ *sqn.p and every read action of p has the postcondition start.p* $\leq$ *sqn.p. Assume that* `snlist` *always remains without multiple occurrences. Then the object is atomic.*

We now reformulate this informal description as a reactive specification over the read–write protocol of Sect. 3.3.

The conditions imply that every execution constructs a partial function from sequence numbers to items. We introduce a shared history variable `hist` to hold this partial function. Its domain corresponds to the list `snlist` mentioned above. We use $\perp$ for the value of `hist` outside its domain. Since *start* is used elsewhere in the theory, we replace it here by the identifier *first*. This leads to the specification *HRW* over the protocol of Sect. 3.3 with the declaration

var `hist` : $\mathbb{N} \to$ *Item* $\cup \{\perp\}$;
var `masq` : $\mathbb{N}$;
privar *first*, *sqn* : $\mathbb{N}$;
initially `hist(masq)` = *item0* $\wedge$ $(\forall\, n : n \neq$ `masq` $\Rightarrow$ `hist`$(n) = \perp)$.

The system always first reads *first* := `masq`, then chooses *sqn*, concludes with `masq` := max(`masq`, *sqn*), and returns control to the environment. Furthermore, a writing process puts its argument at `hist`(*sqn*).

```
Wr.p :   []  pc = 20  →  first := masq ;  pc++
         []  pc = 21  →  choose sqn with first < sqn ∧ hist(sqn) = ⊥ ;
                         hist(sqn) := arg ;  pc++
         []  pc = 22  →  masq := max(masq, sqn) ;  pc++
         []  pc = 23  →  pc := 0 .
```

A reading process reads its result from `hist`(*sqn*).

```
Rd.p :   []  pc = 50  →  first := masq ;  pc++
         []  pc = 51  →  choose sqn with first ≤ sqn ∧ hist(sqn) ≠ ⊥ ;
                         result := hist(sqn) ;  pc++
         []  pc = 52  →  masq := max(masq, sqn) ;  pc++
         []  pc = 53  →  pc := 0
prop:    TERM .
```

This concludes the step relation of *HRW*. For the sake of symmetry, one may want to replace $first < sqn$ in command 21 by $first \leq sqn$. This is allowed because of the easy invariant $\texttt{hist}(first) \neq \bot$. The progress property is the same as for specification *ARW*.

Now Theorem 1 is formalized as

Theorem 2 *Specification HRW implements ARW.*

It seems likely that the proof of Theorem 2 can be obtained by reformulating the proof of Theorem 1 in [3]. This would result in an ad hoc proof without additional value. We prefer to regard Theorem 2 as a challenge for the theory of refinements or simulations of [1,7]. In the next section, we develop and extend this theory in such a way that we can prove Theorem 2 in Sect. 5.

4 Simulations

The easiest way to prove implementation relations between different specifications is by means of refinement mappings. It is well known, however, that refinement mappings are often too specific for this purpose. Usually, we also need to extend the state space with history variables [1], prophecy variables [1], or eternity variables [7]. All these methods are unified as (strict) simulations [4].

4.1 Simulations

Whereas the concept of implementation is defined in terms of the relation between observable behaviours of the two specifications involved, a simulation is a relation between the two state spaces that satisfies certain conditions on the observations and the behaviours.

For specifications K and L, let F be a binary relation F between $states(K)$ and $states(L)$. We write F_ω for the relation between infinite sequences given by

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

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

A relation F between the state spaces of K and L is called a *simulation* (notation $F : K \rightarrow\!\!\!\!\triangleright\!\!\triangleright L$) if for every $xs \in Beh(K)$ there is $ys \in Beh(L)$ and a sequence xt with $xs \preceq xt$ and $(xt, ys) \in F_\omega$. Here, xt is a possibly slowed-down version of the concrete behaviour xs that matches the abstract behaviour ys step by step. Since relation $\preceq$ is reflexive, every strict simulation is a simulation.

For visible specifications (K, v) and (L, w), a relation F between the state spaces of K and L is called *nondisturbing* if F respects the observations in the sense that $v(x) = w(y)$ for all pairs $(x, y) \in F$. The weakest nondisturbing relation is *observational equivalence* $OE = \{(x, y) \mid v(x) = w(y)\}$.

These definitions are justified by the following easy result.

Lemma 1 *Let $F : K \rightarrow\!\!\!\!\triangleright\!\!\triangleright L$ be an nondisturbing simulation. Then (K, v) is an implementation of (L, w). Conversely, if (K, v) is a implementation of (L, w), the relation $OE = \{(x, y) \mid v(x) = w(y)\}$ is a nondisturbing simulation $K \rightarrow\!\!\!\!\triangleright\!\!\triangleright L$.*

We are therefore interested in simulations only when they are nondisturbing. The verification whether some relation is nondisturbing, is usually trivial, but it requires explicit observation functions.

The need for nonstrict simulations is shown in the next example.

Example Consider the specifications $K(m)$ and $K(n)$ of Sect. 2.3 with $m < n$. Give both specifications the binary observation function v with $v(j) = \min(j, 1)$. Then ($K(m)$ and $K(n)$ have the same visible behaviours, namely all stutterings of $(01)^\omega$. Therefore $K(m)$ implements $K(n)$. Yet there is no nondisturbing strict simulation $K(m) \rightarrowtriangle K(n)$. □

In the remainder of this paper, we usually forget about the observations. Of course, our results are only useful when observations are possible and when the simulations are nondisturbing.

If F is a relation between sets X and Y and G is a relation between Y and Z, the *relational composition* $(F; G)$ is the relation between X and Z given by $(F; G) = \{(x, z) \mid \exists\, y \in Y : (x, y) \in F \wedge (y, z) \in G\}$.

Lemma 2 *Let K, L, M be specifications and let $F : K \rightarrowtriangle\!\!\!\rightarrow L$ and $G : L \rightarrowtriangle\!\!\!\rightarrow M$ be simulations. Then $(F; G)$ is a simulation $K \rightarrowtriangle\!\!\!\rightarrow M$. Moreover, if K, L, M are visible specifications and F and G are nondisturbing, then $(F; G)$ is nondisturbing. If the simulations F and G are strict, then $(F; G)$ is strict.*

This Lemma allows use to construct simulations step by step. We first present the easy steps: invariant restriction, refinement mapping, and forward simulation. The first one is so obvious, it is usually overlooked. The other two are well known.

4.2 Invariant restrictions

Let J be a subset of the state space of a specification K. We can then define the specification $L = \mathit{res}(K, J)$ by $\mathit{states}(L) = J$, $\mathit{start}(L) = J \cap \mathit{start}(K)$, $\mathit{step}(L) = J^2 \cap \mathit{step}(K)$, and $\mathit{prop}(L) = J^\omega \cap \mathit{prop}(K)$. The identity function from J into $\mathit{states}(K)$ clearly defines a refinement mapping $\mathit{res}(K, J) \rightarrowtriangle K$.

Assume that J is an invariant of K, cf. Sect. 2.5. Then every behaviour xs of K remains within J. Therefore, the identity relation in $\mathit{states}(K) \times J$ defines a strict simulation $K \rightarrowtriangle \mathit{res}(K, J)$. This simulation is called the *invariant restriction*.

4.3 Refinement mappings and functions

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) $f \circ xs \in \mathit{prop}(L)$ for every $xs \in \mathit{Beh}(K)$.

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 invariant. In the Lemmas 8 and 12 below, we shall encounter refinement functions that are not refinement mappings.

Example Consider the specifications $K(7)$ and $K(4)$ of Sect. 2.4. Let $f : \mathbb{N} \to \mathbb{N}$ be given by $f(j) = \min(j, 3)$. Then f is a refinement mapping from $K(7)$ to $K(4)$. □

Example To see the difference between refinement mappings and refinement functions, consider the specification $L(m)$ for $m > 1$ obtained from $K(m)$ by replacing the assignment to $\mathtt{j}$ by $\mathtt{j} := (\mathtt{j} = m - 1 ? \, 0 : \mathtt{j} + 1)$. Now $L(m)$ has a step from m to $m + 1$, whereas $K(m)$ has a step from m to 1. It follows that the identity function is not a refinement mapping from $K(m)$ to $L(m)$, or vice versa. In both specifications, we have the invariant $\mathtt{j} < m$. Using this one can easily show that the identity function is a refinement function from $K(m)$ to $L(m)$, and vice versa. □

4.4 Forward simulations

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 [10,15,17], defined as follows.

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 behaviour xs of K, we have that $(xs, ys) \in F_\omega$ implies $ys \in$ *prop*(L).

Roughly speaking, condition (F0) is a matter of consistent initialization, condition (F1) says that the steps of K are faithfully represented by L, and condition (F2) says that no additional progress conditions are imposed. The conditions (F0) and (F1) go back to [17], condition (F2) is added in [1].

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

Lemma 3 (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 $\{(x, y) \mid x \in J \wedge f(x) = y\}$ 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 $F : K \twoheadrightarrow L$.*

In view of this Lemma, we use the notation $f : K \twoheadrightarrow L$ also for a refinement function from K to L. A refinement function is called *nondisturbing* iff its graph is nondisturbing.

Example Consider the specifications $K(4)$ and $K(8)$ of Sect. 2.4. Let $F \subseteq \mathbb{N} \times \mathbb{N}$ be given by $F = \{(x, y) \mid x = y \ \mathbf{mod}\ 4\}$. Then F is a forward simulation $K(4) \twoheadrightarrow K(8)$. There is no refinement function f from $K(4)$ to $K(8)$. Indeed, $J = \{0, 1, 2, 3\}$ is the set of reachable states of $K(4)$, and hence the smallest invariant. Every function $f : J \to \mathbb{N}$ with $(f(x), f(x')) \in$ *step*$(K(8))$ whenever $(x, x') \in$ *step*$(K(4))$, is constant since a cycle of eight steps cannot be traversed in four steps. Therefore condition (f1f) cannot be combined with condition (f2). □

4.5 Eternity extensions

Since the prophecy variables of [1] only allow finite nondeterminism, we introduced eternity variables in [4,6]. These variables allow infinite nondeterminism, but they are immutable: the

(infinite) nondeterministic choice is done upon initialization. Thus, extending a specification with an eternity variable is like allowing the specification a single initial guess about the complete future, i.e., about the behaviour that will evolve. The formalism implies that the nondeterminism is angelic: the guess is always right. Soundness only requires satisfiability: every behaviour must allow a correct guess. The formalization is as follows.

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

$$\text{(BR)} \qquad xs \in Beh(K) \;\Rightarrow\; (\exists\, m : \forall\, n : (xs(n), m) \in R)\ .$$

If R is a behaviour restriction of K at M, we define the *eternity extension* $W = et(K, R)$ as the specification W given by

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

Here fst is the function that returns the first component of a pair. It is clear that $step(W)$ is reflexive and that $prop(W)$ is a property. Therefore W is a specification. The component $m \in M$ is called an eternity variable since it does not change during the entire behaviour.

It is easy to verify that $fst : states(W) \to states(K)$ is a refinement mapping. Let relation cvf between $states(K)$ and $states(W)$ be defined as the relational converse of fst. We now prove soundness (cf. [7]):

Theorem 3 *Let R be a behaviour restriction of K at M. Then relation cvf is a strict simulation $K \twoheadrightarrow W$.*

Proof Let $xs \in Beh(K)$. We have to construct $ys \in Beh(W)$ with $(xs, ys) \in cvf_\omega$. By (BR), we can choose m with $(xs(n), m) \in R$ for all $n \in \mathbb{N}$. Define $ys(i) = (xs(i), m)$. A trivial verification shows that the sequence ys constructed in this way is a behaviour of W with $(xs, ys) \in cvf_\omega$. This proves that cvf is a strict simulation. □

The strict simulation $cvf : K \twoheadrightarrow et(K, R)$ of Theorem 3 is called the *eternity extension* of K corresponding to behaviour restriction R.

In general, condition (BR) is a heavy proof obligation. It requires to invent a relation R and then, for every behaviour, it requires to invent an element m. In our practice [6,7], m is always some kind of limit of some abstraction of the states in xs, and relation R expresses this fact. Usually, $et(K, R)$ is not machine closed since it has steps that would eventually lead to violations of R.

4.6 Stutterings

Up to this point, all our simulations were strict. In order to construct nonstrict simulations, we need to formalize stuttering.

We define a *stutter function* to be a function $\mathbb{N} \to \mathbb{N}$ that is monotonic and surjective. This easily implies that the composition of stutter functions is a stutter function. It can be proved that a function $g : \mathbb{N} \to \mathbb{N}$ is a stutter function iff it satisfies $g(0) = 0$, and $g(i + 1) - g(i) \in \{0, 1\}$ for all indices i, and g is unbounded, i.e. for every number n there is an index i with $g(i) \geq n$.

The stuttering relation is now defined by $xs \preceq ys$ if and and only if $xs \circ g = ys$ for some stutter function g. Note that, indeed, even if all elements of xs differ, ys stutters when g stutters (i.e. is not injective). Since the identity function is a stutter function, and the composition of stutter functions is a stutter function, relation $\preceq$ is reflexive and transitive.

4.7 Gliding simulations

Gliding simulations form a non-strict generalization of the forward simulations of Sect. 4.4. For the sake of greater flexibility, relation F is momentarily replaced by two ternary relations T and W, where T "looks ahead" in K and W "looks ahead" in L. The simulation F will be defined in terms of T, whereas W takes cares of the additional stutterings in L. The requirements are phrased in such a way that the construction and its proof are as simple as possible. In the next section, we give a version that is geared more towards the applications.

Let K and L be specifications with $X = \mathit{states}(K)$ and $Y = \mathit{states}(L)$. In order to transfer an execution of K to L, we consider ternary relations $T \subseteq X^2 \times Y$ and $W \subseteq X \times Y^2$. Consider the conditions:

(T0) For every pair $(x, x') \in \mathit{step}(K)$ with $x \in \mathit{start}(K)$, there is $y \in \mathit{start}(L)$ with $(x, x', y) \in T$.

(T1) For every $(x, x', y) \in T$, it holds that there is $y' \in Y$ with $(x, x', y') \in T$ and $(x, y, y') \in W$, or that for every x'' with $(x', x'') \in \mathit{step}(K)$ there is y' with $(y, y') \in \mathit{step}(L)$ and $(x', x'', y') \in T$.

(T2) For every $x \in \mathit{states}(K)$, the binary relation $W[x] = \{(y, y') \mid (x, y, y') \in W\}$ satisfies that $W[x] \subseteq \mathit{step}(L)$ and that there are no infinite sequences with steps in $W[x]$.

We distinguish the two alternatives offered in (T1) as follows. A step is called *gliding step* if it follows the first alternative, and a *computation step* if it follows the second alternative. So, in a gliding step, the concrete states x and x' are unchanged and the abstract state y is replaced by y' with $(y, y') \in W[x]$. In a computation step, the triple (x, x', y) is replaced by (x', x'', y').

Condition (T2) serves to ensure that all gliding steps are steps of L and that every sequence of gliding steps is terminated by a computation step.

Lemma 4 *Let T and W be given such that* (T0), (T1), and (T2) *hold. Let xs be an initial execution of K. Then there is an initial execution ys of L and a stutter function g such that, for all n*:

(*) $(xs(g(n)), xs(g(n)+1), ys(n)) \in T$,

(**) **if** $\exists\, z : (xs(g(n)), ys(n), z) \in W \ \wedge\ (xs(g(n)), xs(g(n)+1), z) \in T$
then $(xs(g(n)), ys(n), ys(n+1)) \in W \ \wedge\ g(n) = g(n+1)$
else $g(n+1) = g(n) + 1$ **end** .

Proof This is proved by an inductive simultaneous construction of ys and g, which satisfies conditions (*) and (**).

We first use condition (T0) to choose $y_0 \in \mathit{start}(L)$ with $(xs(0), xs(1), y_0) \in T$. Then define $ys(0) = y_0$ and $g(0) = 0$. Condition (*) for $n = 0$ follows from (T1). Next, assume that $y = ys(n)$ and $k = g(n)$ are constructed such that (*) holds. Put $x = xs(k)$ and $x' = xs(k+1)$ and $x'' = xs(k+2)$.

If the guard of (**) holds, we choose $ys(n+1) = z$ for some witness z of this guard, and we define $g(n+1) = k$. Otherwise, we define $g(n+1) = k+1$, use the second

alternative offered in condition (T1), and choose $ys(n+1) = y'$ with $(y, y') \in \mathit{step}(L)$ and $(x', x'', y') \in T$. In either case, it follows that (**) holds for n, and that (*) holds for $n+1$.

Sequence ys is an initial execution of L since it begins in $\mathit{start}(L)$ and always takes steps in L, because of (T2). Condition (**) implies that $g(i+1) - g(i) \in \{0, 1\}$. It remains to prove that function g is unbounded. Assume that g bounded. Since it is ascending, it becomes eventually constant, say $g(n) = k$ for all $n \geq i$. Then condition (**) implies that $(ys(n), ys(n+1)) \in W[xs(k)]$ for all $n \geq i$. This contradicts the second clause of (T2), thus proving that g is unbounded. Since $g(0) = 0$, it follows that g is a stutter function. □

Provisionally, we define relation F between the state spaces of K and L to be a *straight gliding simulation* if there are ternary relations $T \subseteq X^2 \times Y$ and $W \subseteq X \times Y^2$ that satisfy the above conditions (T0), (T1), and (T2), and F satisfies condition (F2) of Sect. 4.4, and:
(T3) For every $(x, x', y) \in T$, we have $(x, y) \in F$.

Lemma 5 *Let F be an straight gliding simulation between specifications K and L. Then F is a simulation $K \rightarrowtriangle L$.*

Proof Let xs be a behaviour of K. We apply Lemma 4 with xs as initial execution. Lemma 4 yields an initial execution ys and a stutter function g, such that $(xs \circ g, ys) \in F_\omega$ because of (T3). Since g is a stutter function, $xt = xs \circ g$ satisfies $xs \preceq xt$. Since xs is a behaviour of K, sequence xt is also a behaviour of K. Therefore, condition (F2) of Sect. 4.4 implies that ys is a behaviour of L. This proves that F is a simulation. □

4.8 General gliding simulations

Unfortunately, in Lemma 13 below, we have a case where condition (F2) of Sect. 4.4 is not valid. We therefore weaken it by strengthening its antecedent. Given a sequence xs, condition (F2) allows the construction of ys in an arbitrary way provided $(xs, ys) \in F_\omega$. The construction in Lemma 4, however, is more precise and chooses gliding steps in (**) whenever these are enabled. By choosing the **then** part of (**) whenever possible, the guard of (**) is forced to become false in a finite number of steps because of condition (T2). The fact that the guard of (**) becomes false in a finite number of steps is used in the concept of W-immediacy to be introduced now.

Let a pair of infinite sequences xs, ys be called *W-immediate* if, for every k, there exists n with $k \leq n$ and $xs(k) = xs(n)$ and

$$\neg\, (\exists\, z : (xs(n), ys(n), z) \in W \ \wedge\ (xs(n), xs(n+1), z) \in T).$$

We now postulate the condition:
(T4) For every initial execution ys of L and every behaviour xs of K such that $(xs, ys) \in F_\omega$ and that the pair xs, ys is W-immediate, we have $ys \in \mathit{prop}(L)$.

It is clear that condition (F2) implies (T4).

Relation F between the state spaces of K and L is defined to be a *gliding simulation* if there are a ternary relations $T \subseteq X^2 \times Y$ and $W \subseteq X \times Y^2$ that satisfy the above conditions (T0), (T1), (T2), (T3), and (T4).

Using condition (**) in Lemma 4, the proof of Lemma 5 is easily adapted to yield the following stronger result:

Theorem 4 *Let F be a gliding simulation between specifications K and L. Then F is a simulation $K \rightarrowtriangle L$.*

Remark 1 The definition of W-immediacy is provisional. It is strong enough for the present purposes, but there is room for variations and strengthenings.

4.9 Forms of ternary relations

To prove that a given relation is a gliding simulation requires the invention of two ternary relations. In our two applications, relation T is of the form

$$T = \{(x, x', y) \mid (x, y) \in F \wedge (x, x') \in \mathit{step}(K) \wedge (x', y) \in H\}$$

where F is the simulation relation and H is another binary relation between the state spaces of K and L. If T is of this form, condition (T3) clearly holds.

In these applications, relation W is independent of its first argument. So, in view of (T2), it is of the form $(x, y, y') \in W \equiv (y, y') \in W'$ for some subrelation $W' \subseteq \mathit{step}(L)$ that does not allow infinite sequences of W' steps. The greater generality of W is in anticipation of other applications.

The special forms of T and W do not really reduce the complexity of the proof obligations (T0), (T1), (T4).

4.10 Summary of simulation concepts

We now summarize the simulation concepts defined above, give some idea about how to use them, and discuss some related concepts from the literature.

Simulations are introduced to prove implementation between specifications. Simulations must have no effect on the observations. This is expressed by the condition that the simulation should be *nondisturbing*, see Lemma 1. A simulation is *strict* if every concrete step corresponds to at most one abstract step. It was noted already in [1] that cases occur where the abstract specification needs more steps than the concrete one. We come back to this at (2) below.

The primary examples of simulations are *refinement mappings or functions*, and *forward simulations*. When we remove or rename variables, we form a refinement mapping or function. When we extend a specification with history variables, we get a forward simulation. In a formal treatment, one also needs the *invariant restrictions* of Sect. 4.2, but these are conceptually so innocent that they are often overlooked.

If these methods fail, we try to form an intuitive understanding of the corresponding states and steps, and to identify the obstacle we encounter. There are two main kinds of obstacles: (1) the abstract specification makes a choice that is delayed in the concrete specification, (2) the abstract specification needs more steps than the concrete specification for a certain subtask.

(1) When the abstract specification makes a nondeterministic choice earlier than the concrete specification, one needs "prophecies". In such cases, we use the *eternity extensions* of Sect. 4.5. The same role can be played by *backward simulations* or extensions with *prophecy variables*, see [1,7]. Eternity extensions have the advantage that they do not rely on finite nondeterminacy, as opposed to backward simulations and prophecy variables.

(2) When the abstract specification needs more steps than the concrete one, the matching of the steps becomes more complicated. There are several kinds of simulations that deal with this. The prophecy variables of [1] can deal with it. The paper [14] introduces *stuttering variables* for this purpose. In Sects. 4.7, 4.8, and 4.9, we introduced *gliding simulations* for it. These are more powerful but also more complicated than stuttering

variables. The *splitting simulations* of [9] play the same role but only for specific specifications.

One can select a smaller repertoire of simulations that is semantically complete [1,7,8]. The simulations presented here are chosen with the aim to provide intermediate specifications as naturally as possible.

5 Construction of a simulation from *HRW* to *ARW*

In the methodology of program design, one prefers to start with an abstract specification and then to proceed via a number of refinement steps (simulation relations) towards an implementable specification (program). The challenge of this section is different. Given are two simple specifications *ARW* and *HRW*, and the aim is to prove Theorem 2, that *HRW* implements *ARW*.

The idea of the proof is to create a number of intermediate specifications with simulations between them. The selection of intermediate specifications is driven by the aim to resolve the differences between *HRW* and *ARW* one by one. For each intermediate refinement step or simulation, we try to match the states of the two state spaces in such a way that the steps correspond.

5.1 Roadmap for a simulation from *HRW* to *ARW*

We now give an overview of the construction. The starting point is a comparison of the specifications of *HRW* of Sect. 3.5 and specification *ARW* of Sect. 3.4. The first point to observe is that the sequence numbers *sqn* chosen in *HRW* at lines 21 and 51 must be used to order and synchronize the abstract write and read actions of *ARW*. Since these abstract actions may have to occur before execution of the lines 21 and 51, we first have to move the choices of *sqn* to the lines 20 and 50. This is done in our main intermediate specification *PRW*.

When we are looking for a simulation from *HRW* to *PRW*, we regard *HRW* as the concrete specification and *PRW* as the abstract one. The problem is therefore that the abstract specification *PRW* does some nondeterministic choice earlier that the concrete specification *HRW*. This asks for prophecy variables or an eternity extension. We have chosen to use an eternity extension $KRW \rightarrowtriangle ERW$, preceded by a forward simulation $HRW \rightarrowtriangle KRW$ to introduce the history variables needed for prophecy, and followed by a refinement function $ERW \rightarrowtriangle PRW$ to remove the superfluous variables. Composing these by means of Lemma 2 yields a strict simulation from *HRW* to *PRW*.

In *PRW*, the actions have their sequence number from the beginning. The next goal is to reorder the actions according to sequence number. Since the write action with a given sequence number must precede all read actions with this number, we first have to identify the writing process for each sequence number. This is done in specification *QRW*, via a forward simulation $PRW \rightarrowtriangle QRW$. Then the actions can be reordered. Since reordering actions means delaying some actions, we use a gliding simulation from (an invariant restriction of) *QRW* to a specification *TRW*, in which the writers and the readers are synchronized according to the sequence numbers. Finally, a refinement function from *TRW* to *ARW* is used to remove the superfluous variables.

Summarizing, we construct the simulation from *HRW* to *ARW* as a huge composition

$$HRW \rightarrowtriangle KRW \rightarrowtriangle ERW \rightarrowtriangle PRW \rightarrowtriangle QRW \twoheadrightarrow TRW \rightarrowtriangle ARW \ .$$

Apart from *ARW*, all these specifications use almost the same variables and locations, the same progress property *TERM*, and, as far as relevant, the same initializations.

5.2 The prophetic specification *PRW*

The first intermediate goal is the specification *PRW* where the sequence numbers *sqn* are chosen in the lines 20 and 50, rather than in 21 and 51 as in *HRW*. Moving a nondeterministic choice to an earlier point in the code is called a *prophecy* [1]. Specification *PRW* is a variation of *HRW* in which the private variables *first* have been removed. The environment commands and the commands 22, 23, 52, 53 of *HRW* are retained. The commands 20, 21, 50, 51 are replaced. The writing code becomes:

$$
\begin{array}{lll}
\texttt{Wr}.p: & [] \;\; pc = 20 \;\rightarrow & \textbf{choose}\ sqn\ \textbf{with}\ \texttt{masq} < sqn \;\wedge\; \texttt{hist}(sqn) = \bot\ ; \\
 & & \quad \texttt{hist}(sqn) := arg\ ;\ pc := 22 \\
 & [] \;\; pc = 22 \;\rightarrow & \texttt{masq} := \max(\texttt{masq}, sqn)\ ;\ pc{+}{+} \\
 & [] \;\; pc = 23 \;\rightarrow & pc := 0\ .
\end{array}
$$

The reading code becomes:

$$
\begin{array}{lll}
\texttt{Rd}.p: & [] \;\; pc = 50 \;\rightarrow & \textbf{choose}\ sqn\ \textbf{with}\ \texttt{masq} \le sqn\ ;\ pc{+}{+} \\
 & [] \;\; pc = 51 \;\wedge\; \texttt{hist}(sqn) \neq \bot \;\rightarrow & result := \texttt{hist}(sqn)\ ;\ pc{+}{+} \\
 & [] \;\; pc = 52 \;\rightarrow & \texttt{masq} := \max(\texttt{masq}, sqn)\ ;\ pc{+}{+} \\
 & [] \;\; pc = 53 \;\rightarrow & pc := 0\ .
\end{array}
$$

Note that the environment is still there, and is unchanged. Again the progress property is *TERM*.

The challenge to construct a simulation from *HRW* to *PRW* asks for prophecy variables. Note that specification *PRW* is not machine closed (Sect. 2.5): if a reader q chooses an unused sequence number *sqn*, and `masq` increases beyond *sqn*, the reader is blocked, $pc.q$ can never reach 0, and the execution prefix cannot be extended to a behaviour.

5.3 Construction of an eternity record

In this section, we construct the three strict simulations needed to go from *HRW* to *PRW*. The starting point is specification *HRW*. At lines 20 and 50, the processes have to prophesy the value of *sqn* that will be chosen in lines 21 and 51, respectively. We therefore extend *HRW* to a specification *KRW* with private history variables *sqlist* to hold the subsequently chosen values of *sqn*, and *cnt* to hold the number of values chosen. For convenience, we model *sqlist* as a function.

$$
\begin{array}{l}
\textbf{privar}\ cnt : \mathbb{N} := 1\ ; \\
\textbf{privar}\ sqlist : \mathbb{N} \to \mathbb{N}\ ; \\
\textbf{initially}\ sqlist.q(0) = \texttt{masq}\ .
\end{array}
$$

These variables are modified in lines 21 and 51 in the following way

$$
\begin{array}{lll}
[] \;\; pc = 21 \;\rightarrow & \textbf{choose}\ sqn\ \textbf{with}\ first < sqn \;\wedge\; \texttt{hist}(sqn) = \bot\ ; \\
 & \quad sqlist(cnt) := sqn\ ;\ cnt{+}{+}\ ;\ \texttt{hist}(sqn) := arg\ ;\ pc{+}{+} \\
[] \;\; pc = 51 \;\rightarrow & \textbf{choose}\ sqn\ \textbf{with}\ first \le sqn \;\wedge\; \texttt{hist}(sqn) \neq \bot\ ; \\
 & \quad sqlist(cnt) := sqn\ ;\ cnt{+}{+}\ ;\ result := \texttt{hist}(sqn)\ ;\ pc{+}{+}\ .
\end{array}
$$

The remainder of the code remains unchanged. The verification of the following result is straightforward.

Lemma 6 *Let* id *be the relation between the state spaces of HRW and KRW that expresses equality of all common variables. Then* id *is a nondisturbing forward simulation from HRW to KRW.*

We now form an eternity extension of *KRW* with a shared eternity variable `ehist` of type $(\mathbb{N} \to \mathit{Item} \cup \{\bot\})$ and private eternity variables *esql* of the type $(\mathbb{N} \to \mathbb{N})$. The behaviour restriction R expresses that `hist` is always a partial version of `ehist` and that the functions *sqlist* and *esql* agree on the first *cnt* elements for all readers q. We thus define

$$\begin{aligned} R \;\equiv\; & (\forall\, n : \mathtt{hist}(n) = \bot \;\vee\; \mathtt{hist}(n) = \mathtt{ehist}(n)) \\ & \wedge\; (\forall\, q, i : i < \mathit{cnt}.q \;\Rightarrow\; \mathit{sqlist}.q(i) = \mathit{esql}.q(i)) \;. \end{aligned}$$

To show that R is a behaviour restriction (i.e. satisfies (BR)), we consider some behaviour *xs* of *KRW*. Function `hist` is only modified in command 21, and then only at an index where it was not yet defined. This implies that there is a function `ehist` that satisfies the first conjunct of R. If we regard $\mathit{sqlist}.q$ as a list of length *cnt*, it is modified only in steps 21 and 51, and then by extending the list at the end. We can therefore satisfy the second conjunct of R by choosing *esql* as the limit of *sqlist* if *cnt* grows to infinity. If *cnt* stabilizes at some finite value, the remainder of *esql* can be chosen arbitrarily. So, for every execution *xs*, some function `ehist` and some set of functions $\mathit{esql}.q$ exists that combined satisfy R. This proves (BR).

Let *ERW* be the corresponding eternity extension $\mathit{et}(\mathit{KRW}, R)$. By Theorem 3, we have:

Lemma 7 *Let* id *be the relation between the state spaces of KRW and ERW that expresses equality of all common variables. Then* id *is a nondisturbing strict simulation from HRW to KRW.*

At this point the idea is to use $\mathit{esql}(\mathit{cnt})$ as the prophecy of *sqn* when the reader is at 21 or 51 (so that *sqn* has not yet been chosen).

We claim that *ERW* has the invariants

$$\begin{aligned} &\text{(Lq0)} \quad \mathit{pc}.q = 50 \;\Rightarrow\; \mathtt{masq} \le \mathit{esql}.q(\mathit{cnt}.q)\;, \\ &\text{(Lq1)} \quad \mathit{pc}.q = 51 \;\Rightarrow\; \mathit{first}.q \le \mathit{esql}.q(\mathit{cnt}.q)\;, \\ &\text{(Lq2)} \quad \mathit{pc}.q = 20 \;\Rightarrow\; \mathtt{masq} < \mathit{esql}.q(\mathit{cnt}.q) \;\wedge\; \mathtt{hist}(\mathit{esql}.q(\mathit{cnt}.q)) = \bot\;, \\ &\text{(Lq3)} \quad \mathit{pc}.q = 21 \;\Rightarrow\; \mathit{first}.q < \mathit{esql}.q(\mathit{cnt}.q) \;\wedge\; \mathtt{hist}(\mathit{esql}.q(\mathit{cnt}.q)) = \bot\;. \end{aligned}$$

Note that these predicates are not inductive. They are so-called backward invariants [6] Sect. 2.2. The proofs of (Lq1) and (Lq3) use the second conjunct of the behaviour restriction. The proofs of (Lq0) and (Lq2) use (Lq1) and (Lq3), respectively.

We also need the following backward invariants

$$\begin{aligned} &\text{(Lq4)} \quad \mathit{pc}.q = 21 \;\Rightarrow\; \mathtt{ehist}(\mathit{esql}.q(\mathit{cnt}.q)) = \mathit{arg}.q\;, \\ &\text{(Lq5)} \quad \mathit{pc}.q \in \{20, 21\} \;\wedge\; \mathit{pc}.r \in \{20, 21\} \;\wedge\; \mathit{esql}.q(\mathit{cnt}.q) = \mathit{esql}.r(\mathit{cnt}.r) \\ &\qquad\quad \Rightarrow\; q = r\;. \end{aligned}$$

The proof of (Lq4) is based on the first conjunct of the behaviour restriction. The proof of (Lq5) is based on the second conjunct of the behaviour restriction, together with (Lq2) and (Lq3).

For state x of *ERW*, $n \in \mathbb{N}$, and process q, we define $\mathit{histp}(x, n) \in \mathit{Item} \cup \{\bot\}$ and $\mathit{sqnp}(x, q) \in \mathbb{N}$ by the conditional expressions

$$\begin{aligned} &\mathit{histp}(x, n) = (\exists\, q : \mathit{crit}(x, n)\;?\; x.\mathtt{ehist}(n) : x.\mathtt{hist}(n))\;, \text{ where} \\ &\quad \mathit{crit}(x, n) \equiv (\exists\, q : x.\mathit{pc}.q = 21 \wedge x.\mathit{esql}.q(x.\mathit{cnt}.q) = n)\;, \\ &\mathit{sqnp}(x, q) = (x.\mathit{pc}.q \in \{21, 51\}\;?\; x.\mathit{esl}.q(x.\mathit{cnt}.q) : x.\mathit{sqn}.q)\;. \end{aligned}$$

Here $crit(x, q, n)$ expresses that process q is about to write the value $x.\mathtt{ehist}(n)$ into $\mathtt{hist}(n)$.

Now, we let $fep(x)$ in the state space of *PRW* be given by:

$$
\begin{array}{l}
fep(x) = (\# \\
\quad \mathtt{out} := x.\mathtt{out}\ ,\ \mathtt{masq} := x.\mathtt{masq}\ ,\ arg := x.arg\ ,\ result := x.result\ , \\
\quad \mathtt{hist} := \lambda\, n : histp(x, n)\ , \\
\quad sqn := \lambda\, q : sqnp(x, q)\ , \\
\quad pc := \lambda\, q : (x.pc.q = 21\ ?\ 22 : x.pc.q) \quad \#)\ ,
\end{array}
$$

where (# and #) are the datatype delimiters of PVS.

Lemma 8 *Function fep is a nondisturbing refinement function ERW* $\rightarrowtriangle$ *PRW.*

Proof The verifications of the conditions (f0) and (f2) of Sect. 4.3 is straightforward. In condition (f1f), we use as invariant the conjunction of the invariants (Lq0) up to (Lq5). Validity of the choice of *sqn* for *fep* in steps 20 and 50 follows from the invariants (Lq2) and (Lq0), respectively. The new value of `hist` after step 20 of *PRW* is correct because of (Lq4) and (Lq5). Here (Lq5) is needed to show that $crit(x, n)$ is false when process p is at 20 and $n = esql.p(x.cnt.p)$. It follows from (Lq4) and behaviour restriction R that command 21 of *ERW* is mapped to skip in *PRW*. The behaviour restriction is also needed at 51. □

5.4 Extension of the prophetic specification

We now proceed from *PRW* towards *ARW*. The first step is to identify the process that writes at sequence number *sqn*. For this purpose, we introduce a shared auxiliary variable `wr` of type $\mathbb{N} \rightarrow$ *Process*. This gives a slight modification at 20.

$$
\begin{array}{lll}
\mathtt{Wr}.p: & [] \quad pc = 20 \ \rightarrow & \textbf{choose}\ sqn\ \textbf{with}\ \mathtt{masq} < sqn\ \wedge\ \mathtt{hist}(sqn) = \bot\ ; \\
 & & \quad \mathtt{hist}(sqn) := arg\ ;\ \mathtt{wr}(sqn) := p\ ;\ pc := 22 \\
 & [] \quad pc = 22 \ \rightarrow & \mathtt{masq} := \max(\mathtt{masq}, sqn)\ ;\ pc{+}{+} \\
 & [] \quad pc = 23 \ \rightarrow & pc := 0\ .
\end{array}
$$

The reading code and the environment are still there, and are unchanged. Again the progress property is *TERM*. Let *QRW* be the corresponding specification. We construct a forward simulation $Fpq : PRW \rightarrowtriangle QRW$ by

$$
\begin{array}{l}
(x, y) \in Fpq \ \equiv \\
\quad x.\mathtt{out} = y.\mathtt{out}\ \wedge\ x.\mathtt{hist} = y.\mathtt{hist}\ \wedge\ x.\mathtt{masq} = y.\mathtt{masq} \\
\wedge\ \ x.sqn = y.sqn\ \wedge\ x.arg = y.arg\ \wedge\ x.result = y.result \\
\wedge\ \ x.pc = y.pc\ .
\end{array}
$$

The verification that *Fpq* is a forward simulation is completely standard.

The relevance of the extension is in the invariant of *QRW* that expresses the identity of the writer when it has written and not yet updated `masq`:

$$
\text{(Kq0)} \qquad \mathtt{masq} < n\ \wedge\ \mathtt{hist}(n) \neq \bot\ \Rightarrow\ pc.(\mathtt{wr}(n)) = 22\ \wedge\ sqn.(\mathtt{wr}(n)) = n\ .
$$

Indeed, it is easy to verify that (Kq0) is inductive. At this point, we need to observe three other invariants of *QRW*:

$$
\begin{array}{ll}
\text{(Kq1)} & pc.q \in \{22, 23\}\ \Rightarrow\ \mathtt{hist}(sqn.q) = arg.q\ \wedge\ \mathtt{wr}(sqn.q) = q\ , \\
\text{(Kq2)} & pc.q > 51\ \Rightarrow\ \mathtt{hist}(sqn.q) = result.q\ , \\
\text{(Kq3)} & pc.q = 53\ \Rightarrow\ sqn.q \leq \mathtt{masq}\ .
\end{array}
$$

Indeed, all three predicates are inductive. In (Kq1) and (Kq2), we do not really need the value of $\mathtt{hist}(sqn.q)$, but only that it differs from $\bot$, so that (Kq0) may be applicable.

We finally observe and prove the backward invariant:

(Kq4) $\quad pc.q = 51 \;\wedge\; \mathtt{hist}(sqn.q) = \bot \;\Rightarrow\; \mathtt{masq} < sqn.q$.

Let *QRWi* be the invariant restriction of specification *QRW* for these invariants. By Sect. 4.2, the identity relation is a strict simulation $QRW \rightarrowtriangle QRWi$. By combining the results of this section with the Lemmas 6, 7, and 8 by means of the Lemmas 2 and 3, we obtain

Lemma 9 *There is a nondisturbing strict simulation $HRW \rightarrowtriangle QRWi$.*

5.5 The timed specification

The next step is that we extend writing with serialization points that correspond to the value of *sqn*. This gives rise to the timed specification *TRW*, where we no longer need array `wr`. In comparison with *PRW*, the only change is that, in *TRW*, the variable `masq` is never incremented by readers, but only by the writer with the appropriate sequence number. The writing code is given by

$$\begin{array}{lll}
\mathtt{Wr}.p: & [] & pc = 20 \;\rightarrow\; \textbf{choose } sqn \textbf{ with } \mathtt{masq} < sqn \;\wedge\; \mathtt{hist}(sqn) = \bot\,; \\
& & \qquad\qquad \mathtt{hist}(sqn) := arg\,;\; pc := 22 \\
& [] & pc = 22 \;\wedge\; \mathtt{masq} < sqn \;\rightarrow\; \mathtt{masq} := sqn\,;\; pc{+}{+} \\
& [] & pc = 23 \;\rightarrow\; pc := 0\,.
\end{array}$$

The reading code becomes:

$$\begin{array}{lll}
\mathtt{Rd}.p: & [] & pc = 50 \;\rightarrow\; \textbf{choose } sqn \textbf{ with } \mathtt{masq} \le sqn\,;\; pc{+}{+} \\
& [] & pc = 51 \;\wedge\; sqn = \mathtt{masq} \;\wedge\; \mathtt{hist}(\mathtt{masq}) \neq \bot \;\rightarrow \\
& & \qquad result := \mathtt{hist}(\mathtt{masq})\,;\; pc := 53 \\
& [] & pc = 53 \;\rightarrow\; pc := 0\,.
\end{array}$$

Just as before, the environment is retained and the progress property is *TERM*.

Specification *TRW* has the invariants:

(Mq0) $\quad pc.q = 22 \;\Rightarrow\; \mathtt{masq} < sqn.q$,
(Mq1) $\quad pc.q = 23 \;\Rightarrow\; sqn.q \le \mathtt{masq}$,
(Mq2) $\quad pc.q = 51 \;\Rightarrow\; \mathtt{masq} \le sqn.q$,
(Mq3) $\quad pc.q = 53 \;\Rightarrow\; \mathtt{hist}(sqn.q) = result.q$,
(Mq4) $\quad \mathtt{hist}(\mathtt{masq}) \neq \bot$.

At this point, we do not need to prove these invariants for *TRW*, but we incorporate them in the simulation relation. Let *Mq* be the boolean function on the state space of *TRW* that expresses, for all processes q, the predicates (Mq0), …, (Mq4).

We construct a gliding simulation $Fqt : QRWi \rightarrowtriangle\!\!\rightarrowtriangle TRW$. Since the variables `out`, `hist`, *arg*, *sqn* play precisely the same roles in the two specifications, *Fqt* expresses equality for these variables. The difference is that, whenever `masq` is incremented in lines 22 or 52 of *QRWi*, several writers may have to execute line 22 of *TRW*. Therefore, `masq` may have to traverse a number of intermediate values. It follows that the two specifications differ in their

treatments of *pc* and masq. We thus propose the simulation relation:

$$
\begin{array}{l}
(x, y) \in \mathit{Fqt} \;\equiv \\
\quad x.\mathtt{masq} \leq y.\mathtt{masq} \;\wedge\; x.\mathtt{out} = y.\mathtt{out} \;\wedge\; x.\mathtt{hist} = y.\mathtt{hist} \\
\quad \wedge\; x.\mathit{arg} = y.\mathit{arg} \;\wedge\; x.\mathit{sqn} = y.\mathit{sqn} \\
\quad \wedge\; (\forall\, q : (x.\mathit{pc}.q = y.\mathit{pc}.q \leq 50 \;\vee\; (x.\mathit{pc}.q = 22 \;\wedge\; y.\mathit{pc}.q = 23) \\
\qquad\qquad \vee\; (x.\mathit{pc}.q > 50 \;\wedge\; y.\mathit{pc}.q \in \{51, 53\})\,) \\
\qquad\quad \wedge\; (x.\mathit{result}.q = y.\mathit{result}.q \;\vee\; x.\mathit{pc}.q > 50)\,) \\
\quad \wedge\; \mathit{Mq}(y)\,.
\end{array}
$$

Lemma 10 *Relation Fqt is a nondisturbing gliding simulation QRWi* $\rightarrow\!\!\triangleright$ *TRW.*

Proof We define binary relations *Hqt* and *Wt* to play the roles of H and W' in Sect. 4.9. Relation *Hqt* is given by

$$(x', y) \in \mathit{Hqt} \;\equiv\; y.\mathtt{masq} \leq x'.\mathtt{masq}\,.$$

In this way, the conjunction $(x, y) \in \mathit{Fqt} \wedge (x', y) \in \mathit{Hqt}$ expresses that $y.\mathtt{masq}$ is between $x.\mathtt{masq}$ and $x'.\mathtt{masq}$. Since relation *Fqt* allows the steps at 22 and 51 to occur at different moments in *QRWi* and *TRW*, we let the gliding steps correspond to the steps of *TRW* at 22 and 51. We thus define $\mathit{Wt} = \bigcup_q \mathit{wt}.q$ where $\mathit{wt}.q$ is the union of the two step relations in *TRW* for the steps of process q at 22 and 51.

Let the ternary relations T and W be defined in terms of *Fqt*, *Hqt* and *Wt* as described in Sect. 4.9. It now remains to verify the conditions (T0), (T1), (T2), and (T4). The verification of (T0) is straightforward. It is clear that *Wt* is a subrelation of the step relation of *TRW*. In every *Wt* step, the number of processes q with $\mathit{pc}.q \in \{22, 51\}$ decreases. This proves condition (T2).

As for (T4), let $(\mathit{xs}, \mathit{ys}) \in \mathit{Fqt}_\omega$ for some behaviour *xs* of *QRWi*. It suffices to prove that *ys* satisfies condition *TERM*. Let a process q and a number n be given. Since *xs* satisfies *TERM*, there is $k \geq n$ with $\mathit{xs}(k).\mathit{pc}.q = 0$. Since $(\mathit{xs}(k), \mathit{ys}(k)) \in \mathit{Fqt}$, this implies that $\mathit{ys}(k).\mathit{pc}.q = 0$. Therefore, *ys* satisfies *TERM*, thus proving (T4).

It remains to verify condition (T1). Let $(x, x', y) \in T$ be given, i.e., (x, x') is a step of *QRWi*, and $(x, y) \in \mathit{Fqt}$, and $(x', y) \in \mathit{Hqt}$. We need several case distinctions. In our experience, the easiest approach is to start by treating and eliminating the gliding steps.

First, assume that there is a process q with $y.\mathit{pc}.q = 51$ and $y.\mathit{sqn}.q = y.\mathtt{masq}$. Since y satisfies (Mq4), process q can do step 51 in *TRW*. So there is y' with $(y, y') \in \mathit{Wt}$. An easy verification shows that $(x, x', y') \in T$, as required. We may therefore assume that such a process does not exist. Since y satisfies (Mq2), it follows that

$$\forall\, q : y.\mathit{pc}.q = 51 \;\;\Rightarrow\;\; y.\mathtt{masq} < y.\mathit{sqn}.q. \tag{0}$$

The second case is that $y.\mathtt{masq} < x'.\mathtt{masq}$. Since $(x, y) \in \mathit{Fqt}$, it follows that $x.\mathtt{masq} < x'.\mathtt{masq}$. Therefore, the step (x, x') of *QRWi* increases $x.\mathtt{masq}$. This implies that there is a process, say p, that performs step 22 or 52. We then have $x'.\mathtt{masq} = x.\mathit{sqn}.p$. Since x satisfies (Kq1) and (Kq2), it follows that $x.\mathtt{hist}(x'.\mathtt{masq}) \neq \bot$. In other words, we have $x'.\mathtt{masq} \in S$ for the set

$$S = \{n \in \mathbb{N} \mid y.\mathtt{masq} < n \;\wedge\; x.\mathtt{hist}(n) \neq \bot\}\,.$$

Let n_0 be the minimum of S. We then have $x.\mathtt{masq} \leq y.\mathtt{masq} < n_0 \leq x'.\mathtt{masq}$ and $x.\mathtt{hist}(n_0) \neq \bot$. We now use that x satisfies (Kq0) for $n := n_0$. It follows that the process $q_0 = x.\mathtt{wr}(n_0)$ satisfies $x.\mathit{pc}.q_0 = 22$ and $x.\mathit{sqn}.q_0 = n_0$. Since $(x, y) \in \mathit{Fqt}$ and since y satisfies (Mq1), it follows that $y.\mathit{pc}.q_0 = 22$. Therefore *TRW* can perform step 22 at q_0.

Let y' be the resulting state. We then have $(y, y') \in Wt$ and $y'.\mathtt{masq} = n_0$. It follows that $(x', y') \in Hqt$ holds.

We need to verify that y' satisfies the predicates (Mq0) up to (Mq4). Since y' only differs from y in masq and $pc.q_0$, and since masq is incremented, we need only consider (Mq0), (Mq2), and (Mq4). State y' satisfies (Mq4) because of $y'.\mathtt{hist}(y'.\mathtt{masq}) = x.\mathtt{hist}(n_0) \neq \bot$.

As for (Mq0), assume $y'.pc.q = 22$. Then $q \neq q_0$ and $x.pc.q = y.pc.q = 22$. By (Mq0) for y, we then have $y.\mathtt{masq} < y.sqn.q$. By (Kq1) for x, we have $x.\mathtt{hist}(x.sqn.q) \neq \bot$ and $x.\mathtt{wr}(x.sqn.q) = q$. Since $q \neq q_0$, it follows that $x.sqn.q \neq n_0$. On the other hand, since $y.sqn.q = x.sqn.q$, it follows that $y.sqn.q \in S$, and hence $n_0 \leq y.sqn.q$. This proves $y'.\mathtt{masq} < y'.sqn.q$, thus proving that y' satisfies (Mq0).

As for (Mq2), assume $y'.pc.q = 51$ and $y'.sqn.q < y'.\mathtt{masq}$. We then have $q \neq q_0$ and $y.sqn.q = y'.sqn.q < n_0$ and $y.pc.q = y'.pc.q = 51$, so that formula (0) implies $y.\mathtt{masq} < y.sqn.q$. Since n_0 is the minimum of S, it follows that $x.\mathtt{hist}(y.sqn.q) = \bot$. Using $(x, y) \in Fqt$ and (Kq2) for x, we get that $x.pc.q = 51$. We now use that $x'.pc.q = x.pc.q$ and $x'.\mathtt{hist} = x.\mathtt{hist}$ and $x'.sqn.q = x.sqn.q$. Since x' satisfies (Kq4), it follows that $n_0 \leq x'.\mathtt{masq} < x'.sqn.q = y'.sqn.q$, a contradiction.

Using these results, it is easy to verify that $(x, y') \in Fqt$ and, hence, that $(x, x', y') \in T$. This concludes the case of $y.\mathtt{masq} < x'.\mathtt{masq}$.

The third and final case is that $x'.\mathtt{masq} \leq y.\mathtt{masq}$. Since $(x', y) \in Hqt$, it follows that $x'.\mathtt{masq} = y.\mathtt{masq}$. We now make a case distinction over the possible steps (x, x') of *QRWi*. In all these subcases, however, we shall do computation steps. Therefore, let (x', x'') be a next step of *QRWi*. We need to find a step (y, y') of *TRW* such that $(x', x'', y') \in T$.

We first treat the case that (x, x') is a step of some process p at 22 or 52, which may modify masq. Since $x.\mathtt{masq} \leq y.\mathtt{masq} = x'.\mathtt{masq}$, we can choose a skipping step $y' = y$ in *TRW*. In the case $x.pc.p = 22$, we use (Mq0) to infer $y.pc.p = 23$. In either case, it is proved that $(x', y') \in Fqt$. Since the step (x', x'') does not decrease masq, we also have $(x', y) \in Hqt$, so that indeed $(x', x'', y') \in T$.

In all remaining cases, the step (x, x') does not modify masq and we have $x.\mathtt{masq} = x'.\mathtt{masq} = y.\mathtt{masq}$. In the case of a step (x, x') for some process p at 51 or of a skipping step $x = x'$, we choose a skipping step $y' = y$ in *TRW*. In the case that (x, x') is the step in *QRWi* of some process p at 0, 20, 23, 50, or 53, we let (y, y') be the corresponding step of p in *TRW*.

First, consider the case that p does the step at 53. Then (Kq3) implies that $x.sqn.p \leq x.\mathtt{masq}$. It follows that $y.sqn.p \leq y.\mathtt{masq}$. Therefore, formula (0) together with *Fqt* implies that $y.pc.q = 53$. So, indeed, y can do the step at 53. We then use (Kq2) for x and (Mq3) for y to get the equality $x'.result.q = y'.result.q$. In the remaining cases, the verifications required are all straightforward. This concludes the proof of (T1) and, hence, of Lemma 10. □

5.6 The refinement function from *TRW* to *ARW*

Since in *TRW*, every writing process increases masq to its own sequence number and every reading process reads its result when its sequence number equals masq, we identify these steps as the actual writing and reading steps and regard hist(masq) as the implementation of reg.

For the next step, we need that *TRW* has the invariant (Mq4) introduced above and the new invariant

(Mq5) $\quad pc.q = 22 \;\Rightarrow\; \mathtt{hist}(sqn.q) = arg.q$.

Indeed, (Mq5) is inductive and (Mq4) is preserved at 22 because of (Mq5). Let *Mq45* be the subset of the state space of *TRW* where (Mq4) and (Mq5) hold.

Recall from Sect. 3.4 that *AX* is the state space of *ARW*. We define the function $fta : Mq45 \to AX$ by

$$
\begin{array}{l}
fta(x) = (\# \\
\quad \mathtt{out} := x.\mathtt{out}\ ,\ arg := x.arg\ ,\ result := x.result\ , \\
\quad \mathtt{reg} := x.\mathtt{hist}(x.\mathtt{masq}), \\
\quad pc := \lambda\, q : (x.pc.q \in \{20, 22\}\ ?\ 20 : \\
\qquad\qquad x.pc.q \in \{50, 51\}\ ?\ 50 : 0)\ \#)\ .
\end{array}
$$

This function is well defined because of (Mq4). We have:

Lemma 11 *Function fta is a refinement mapping from the invariant restriction for Mq45 of TRW to ARW. Its graph is a nondisturbing strict simulation TRW* $\rightarrowtriangle$ *ARW.*

The verification of this lemma is completely standard. The invariant (Mq5) is needed to show that step 22 in *TRW* corresponds to step 20 in *ARW*. By composing the results of the Lemmas 9, 10, and 11, we obtain:

Theorem 5 *There is a nondisturbing simulation HRW* $\rightarrowtriangle\!\!\rightarrowtriangle$ *ARW.*

By Lemma 1, this proves Theorem 2.

6 Universality of the atomicity criterion?

We conjecture that the atomicity criterion of Theorem 2 is universal in the sense that every specification *XRW* that implements *ARW* has a nondisturbing gliding simulation to *HRW*. For now, we have no idea how to prove this.

In this section, we prove that *ARW* itself has a nondisturbing simulation to *HRW*. This implies that every specification *XRW* that implements *ARW* has a nondisturbing simulation to *HRW*. The question remains, whether *HRW* introduces enough prophecies for all possible implementations of *ARW*.

To construct a gliding simulation *ARW* $\rightarrowtriangle\!\!\rightarrowtriangle$ *HRW*, we consider the question how to transfer a behaviour of *ARW* to *HRW*. The idea is simple enough. Given a behaviour of *ARW*, we let the read and write actions be executed by *HRW* under mutual exclusion.

We therefore define specification *MRW*, which is a variation of *HRW* under mutual exclusion. The state space of *MRW* is *MX*, obtained from the state space *HX* of *HRW* by adding the declaration and initialization

$$\textbf{var}\ \mathtt{mutex} : Process \cup \{\bot\} := \bot\ .$$

The step relation of *MRW* is given by

$$
\begin{array}{lll}
\mathtt{Wr}.p: & [] & pc = 20 \wedge \mathtt{mutex} = \bot \ \to\ \mathtt{mutex} = p\ ;\ first := \mathtt{masq}\ ;\ pc\mathtt{++} \\
& [] & pc = 21 \ \to\ sqn := \mathtt{masq} + 1\ ;\ \mathtt{hist}(\mathtt{masq} + 1) := arg\ ;\ pc\mathtt{++} \\
& [] & pc = 22 \ \to\ \mathtt{masq} := \mathtt{masq} + 1\ ;\ pc\mathtt{++} \\
& [] & pc = 23 \ \to\ \mathtt{mutex} := \bot\ ;\ pc := 0\ . \\
\mathtt{Rd}.p: & [] & pc = 50 \wedge \mathtt{mutex} = \bot \ \to\ \mathtt{mutex} = p\ ;\ first := \mathtt{masq}\ ;\ pc\mathtt{++} \\
& [] & pc = 51 \ \to\ sqn := \mathtt{masq}\ ;\ result := \mathtt{hist}(\mathtt{masq})\ ;\ pc\mathtt{++} \\
& [] & pc = 52 \ \to\ pc\mathtt{++} \\
& [] & pc = 53 \ \to\ \mathtt{mutex} := \bot\ ;\ pc := 0 \\
\textbf{prop:} & & TERM\ .
\end{array}
$$

We first note the mutual exclusion invariant:

(Nq0) $pc.q \in \{0, 20, 50\} \ \vee \ \texttt{mutex} = q$.

Let *fmh* be the projection function from *MX* to *HX* that forgets the value of `mutex`. In order to show that *fmh* is a refinement function, we verify that *MRW* has the invariants

(Nq1) $pc.q \in \{21, 51\} \ \Rightarrow \ first.q = \texttt{masq}$,
(Nq2) $pc.q = 22 \ \Rightarrow \ sqn.q = \texttt{masq} + 1$,
(Nq3) $pc.q = 52 \ \Rightarrow \ sqn.q = \texttt{masq}$,
(Nq4) $\texttt{hist}(\texttt{masq} + 1) \neq \bot \ \Rightarrow \ \texttt{mutex} \neq \bot \ \wedge \ pc.\texttt{mutex} = 22$.

These predicates are preserved under modification of `masq` because of (Nq0). Preservation of (Nq4) at 22 follows from the easy invariant

(Nq5) $\texttt{hist}(n) \neq \bot \ \Rightarrow \ n \leq \texttt{masq} + 1$.

With these invariants, the next lemma is straightforward to verify.

Lemma 12 *Function fmh is a nondisturbing refinement function MRW* $\rightarrowtriangle$ *HRW.*

We turn to the relation between *ARW* and *MRW*. We first observe that *MRW* also has the invariants:

(Nq6) $\texttt{mutex} \neq \bot \ \Rightarrow \ pc.\texttt{mutex} \in \{21, 22, 23, 51, 52, 53\}$,
(Nq7) $\texttt{mutex} \neq \bot \ \wedge \ pc.\texttt{mutex} = 22 \ \Rightarrow \ \texttt{hist}(\texttt{masq} + 1) = arg.\texttt{mutex}$.

Notice that we do not need to prove these invariants. They only serve as predicates, to be incorporated in the simulation relation.

Let *Nq* be the subset of the state space *MX* where the three predicates (Nq0), (Nq6), and (Nq7) hold. Let $fma : Nq \rightarrow AX$ be the function given by

$$
\begin{array}{l}
fma(y) = (\# \\
\quad \texttt{out} := y.\texttt{out} \ , \ arg := y.arg \ , \ result := y.result \ , \\
\quad \texttt{reg} := y.\texttt{hist}(y.\texttt{masq}) \ , \\
\quad pc := pha \circ y.pc \quad \#) \ ,
\end{array}
$$

where *pma* is the location projection given by

$$
pma = \begin{pmatrix} 0 & 20 & 21 & 22 & 23 & 50 & 51 & 52 & 53 \\ 0 & 20 & 20 & 20 & 0 & 50 & 51 & 0 & 0 \end{pmatrix} .
$$

Let *Fam* be the converse relation between *AX* and *MX* given by

$$(x, y) \in Fam \ \equiv \ x = fma(y) \ \wedge \ y \in Nq \ .$$

Lemma 13 *Relation Fam is a nondisturbing gliding simulation ARW* $\rightarrowtriangle\!\!\!\!\rightarrow$ *MRW.*

Proof We choose binary relations *Ham* and *Wm* to play the roles of H and W' described in Sect. 4.9. Relation *Ham* expresses that, when *MRW* has started a read or write procedure of some process q, this is the next step of *ARW*:

$$(x', y) \in Ham \ \equiv \ (\forall \, q : y.pc.q \in \{21, 22, 51\} \ \Rightarrow \ x'.pc.q = 0) \ .$$

In view of function *pma*, we take the steps at the locations 20, 21, 23, 50, 52, and 53 to be gliding steps. We therefore define $Wm = \bigcup_q wm.q$ where $wm.q$ is the union of the six step

relation in *MRW* for the steps of process q at 20, 21, 23, 50, 52, and 53. It is easy to verify that there are no infinite sequences of steps in *Wm*. Therefore, condition (T2) is satisfied. Condition (T0) is also easily verified. Condition (T3) holds by definition.

As for (T1), consider $(x, x', y) \in T$. This means that $(x, y) \in Fam$ and $(x', y) \in Ham$ and (x, x') is a step of *ARW*. First, assume that $y.\texttt{mutex} \neq \bot$. Write $p = y.\texttt{mutex}$. By (Nq6), we have $pc.p \in \{21, 22, 23, 51, 52, 53\}$. First, assume that $y.pc.p \in \{21, 23, 52, 53\}$. Then y can do a gliding step with $(y, y') \in wm.p$ while x and x' remain unchanged. Otherwise, we have $y.pc.p \in \{22, 51\}$. Using *Ham*, we get $x'.pc.p = 0$. For every step (x, x') of *ARW*, we can use the *MRW* step of p at 22 or 51 to get a new triple $(x', x'', y') \in T$. At 22, we need (Nq7) to retain the correspondence $x.\texttt{reg} = y.\texttt{hist}(y.\texttt{masq})$.

It remains to consider the case with $y.\texttt{mutex} = \bot$. Then (Nq0) implies that $y.pc.q \in \{0, 20, 50\}$ for all q, and hence $x.pc.q = y.pc.q$ for all q. We now distinguish according to the step (x, x') of *ARW*. If this is a skip step or a step of the environment, it can be mimicked immediately in *MRW*. It remains that it is a write or read step. Then there is a process q with $x.pc.q = y.pc.q \in \{20, 50\}$ and $x'.pc.q = 0$. In that case, again, y can do a gliding step with $(y, y') \in wm.q$ while x and x' remain unchanged.

It remains to verify condition (T4). Let *ys* be an initial execution of *MRW* and let *xs* be a behaviour of *ARW* such that $(xs, ys) \in Fam_\omega$ and that the pair *xs*, *ys* is *W*-immediate. We have to prove that *ys* satisfies the property *TERM*, that is $\forall\, q : \Box\Diamond[\![\, pc.q = 0 \,]\!]$. Let a process q and a number i be given. We have to prove that there is a $k \geq i$ with $ys(k).pc.q = 0$. Since *xs* is a behaviour of *ARW*, there is a number $n \geq i$ with $xs(n).pc.q = 0$. *W*-immediacy implies that there is $k \geq n$ with $xs(k).pc.q = 0$ and such that there is no state $z \in MX$ with $(z, xs(k+1) \in Ham$ and $(ys(k), z) \in Wm$. Since $(xs(k), ys(k)) \in Fam$, we have $ys(k).pc.q \in \{0, 23, 52, 53\}$. It follows that, if $ys(k).pc.q \neq 0$, then $ys(k)$ can do a gliding step in $wm.q$. This implies that $ys(k).pc.q = 0$. Therefore *ys* satisfies *TERM*. □

Combining the Lemmas 12 and 13, we obtain

Theorem 6 *There is a nondisturbing simulation ARW* $\rightarrow\!\!\triangleright$ *HRW.*

7 The verification with PVS

As announced in the introduction, all results have been verified with the theorem prover PVS. The proof scripts are available at

`www.cs.rug.nl/~wim/mechver/eternity`.

The down-loadable PVS-dumpfile `w359dump` contains a PVS-theory `atomicVar`, which ends with the lemma

```
simHRWtoARW: LEMMA
  simulation?(relHK o cvf(behResHH) o graph(fep) o relPQ
              o idd(qinv) o relQT o (idd(mq45) o graph(fta)),
              histreg, atreg)
```

This asserts that the composition of eight explicitly given relations is a simulation from specification `histreg` (i.e., *HRW*) to specification `atreg` (i.e., *ARW*). Two of these relations are invariant restrictions: `idd`(J) is the identity relation of the invariant restriction for invariant J. Two relations come from functions: `graph`(f) is the relation associated to

function f. The definition of `cvf` is given in Sect. 4.5. The other theory `atomicVarU` ends with the converse lemma

```
simARWtoHRW: LEMMA
   simulation?(relAM o graph(fmh), atreg, histreg)
```

Both theories heavily rely on a number of general theories about the various kinds of refinement and simulation relations. These theories are also included in the dumpfile mentioned. The proofs presented in this paper closely follow the PVS proofs (or vice versa). In order to verify that we proved these two lemmas with PVS, one just has to "undump" the dumpfile and to ask PVS to replay the proof as provided. This takes 5 min on our machinery (a Pentium 4, 2 GHz). Of course, in order to see what has been proved in this way, one has to consult the definitions of `simulation?`, `histreg`, and `atreg` in the proof scripts.

8 Conclusions

Years ago, Groote conjectured that prophecy variables would be needed to prove correctness of Bloom's algorithm. At that time, I had proved Bloom's algorithm by means of the atomicity criterion I am now revisiting, and I did not see the need for prophecy variables. His conjecture is now vindicated in the sense that I do need prophecies in the form of eternity variables to prove the atomicity criterion itself.

The details in the proofs of Theorem 4 and of the Lemmas 8, 10, and 13 are so complicated, that we could only convince ourselves of the proofs by extensive use of the proof assistant PVS [21]. Indeed, there is ample room for improvement and simplification, in particular of the strict simulation $ERW \rightarrowtriangle PRW$ and the gliding simulations $QRWi \rightarrow\!\!\!\!\triangleright\!\!\triangleright TRW$ and $ARW \rightarrow\!\!\!\!\triangleright\!\!\triangleright MRW$. In principle, it is quite possible that there is a way to construct a simulation from *HRW* to *ARW* and vice versa via completely different lists of intermediate specifications.

Acknowledgments We are grateful for the constructive criticisms of an anonymous referee, which led to significant improvements of the presentation.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**, 253–284 (1991)
2. Bloom, B.: Constructing two-writer atomic registers. IEEE Trans. Comput. **37**, 1506–1514 (1988)
3. Hesselink, W.H.: An assertional criterion for atomicity. Acta Inf. **38**, 343–366 (2002)
4. Hesselink, W.H.: Eternity variables to simulate specifications. In: Boiten, E.A., Moeller, B. (eds.) MPC 2002, LNCS, vol. 2386, pp. 117–130. Springer, New York (2002)
5. Hesselink, W.H.: An assertional proof for a construction of an atomic variable. Formal Aspects Comput. **16**, 387–393 (2004)
6. Hesselink, W.H.: Using eternity variables to specify and prove a serializable database interface. Sci. Comput. Program. **51**, 47–85 (2004)
7. Hesselink, W.H.: Eternity variables to prove simulation of specifications. ACM Trans. Comput. Logic **6**, 175–201 (2005)
8. Hesselink, W.H.: Universal extensions to simulate specifications (2005)
9. Hesselink, W.H.: Splitting forward simulations to cope with liveness. Acta Inf. **42**, 583–602 (2006)
10. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 86, LNCS, vol. 213, pp. 187–196. Springer, New York (1986)
11. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Englewood Cliffs (1985)

12. Haldar, S., Subramanian, K.: Space-optimum conflict-free Space-optimum conflict-free construction of 1-writer 1-reader multivalued atomic variable. In: Proceedings of the 8th International Workshop on Distributed Algorithms. LNCS, vol. 857, pp. 116–129. Springer, Heidelberg (1994)
13. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**, 872–923 (1994)
14. Ladkin, P., Lamport, L., Olivier, B., Roegel, D.: Lazy caching in TLA. Distrib. Comput. **12**, 151–174 (1999)
15. Lynch, N., Vaandrager, F.: Forward and backward simulations, part I: untimed systems. Inf. Comput. **121**, 214–233 (1995)
16. Lynch, N.A.: Distributed Algorithms. Morgan Kaufman, San Francisco (1996)
17. Milner, R.: An algebraic definition of simulation between programs. In: Proceedings of the 2nd Internatioanal Joint Conference on Artificial Intelligence, pp. 481–489. British Comp. Soc. (1971)
18. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
19. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New York (1992)
20. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
21. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference (2001) http://pvs.csl.sri.com
22. Vitányi, P.M.B., Awerbuch, B.: Atomic shared register access by Atomic shared register access by asynchronous hardware. In: 27th Annual Symposium on Foundations of Computer Science, pp. 233–243. IEEE, Los Alamitos, Calif., 1986. Corrigendum in 28th Annual Symposium on Foundations of Computer Science, page 487, Los Angeles (1987)
23. Vidyasankar, K.: Concurrent reading while writing revisited. Distrib. Comput. **4**, 81–85 (1990)