An assertional criterion for atomicity

Hesselink, Wim H.

Published in:
Acta informatica

Publication date:
2002

Acta Informatica 38: 343–366 (2002)

An assertional criterion for atomicity

Wim H. Hesselink

Department of Mathematics and Computing Science, Rijksuniversiteit Groningen,
P.O. Box 800, 9700 AV Groningen, The Netherlands
(e-mail: wim@cs.rug.nl, Web: http://www.cs.rug.nl/˜wim)

Received: 10 May 2001 / 6 December 2001

Abstract. A criterion is presented to prove atomicity of read-write objects by means of ghost variables and invariants. The criterion is applied to Bloom's construction of a two-writer atomic register from two one-writer atomic registers and to the algorithm of Vitanyi and Awerbuch for the construction of a read-write object with m readers and writers, based on m^2 read-write objects for one reader and one writer. In both cases, the proof comes down to the verification of a number of invariants. The hand-written proofs of these invariants have been verified with a mechanical theorem prover.

1 Introduction

In this paper we present a criterion for atomicity of read-write objects by means of ghost variables and invariants. Since preservation of a given invariant in a given algorithm is relatively easy to verify or falsify, the criterion makes rigorous, even mechanical, verification easier. The criterion provides guidance to the designer since it introduces the ghost variables with required invariants. It is up to the designer to encode the ghost variables in such a way that the invariants can be preserved. The criterion also reduces the possibility of errors in hand-written proofs: the proof breaks into inevitable cases, and forces one to reason about actions rather than execution traces.

1.1 Atomicity and blocking

Concurrency is introduced for efficient utilization of processing capability. It may lead, however, to undesirable interferences, e.g., when two processes concurrently need exclusive access to some resource. This is the mutual exclusion problem of [7], in which blocking of processes is unavoidable. There are cases, however, where undesirable interferences can be avoided without blocking. When available, such solutions are usually preferred since blocking has always a performance penalty and introduces the danger of deadlock.

Nonblocking methods to avoid undesirable interferences are more difficult to find and to argue about. Indeed, what are "undesirable interferences" and what is the meaning of "nonblocking"? Instead of a negative goal as the avoidance of undesirable interferences, we need a positive goal. This positive goal was first defined in 1979 as serializability [22] or sequential consistency [17]. Later refinements of the theory [12, 20] introduced the terms of linearizability and atomicity.

The term "nonblocking" can also be interpreted in many ways. It is related to fairness (e.g. see [9]). In this paper, we interpret "nonblocking" as wait-free [11]. Informally speaking, a concurrent system is wait-free when every process can achieve its current goal in a bounded number of steps, independently of the (in)activity of other processes.

When we know what we mean by atomicity (linearizability) and nonblocking, the problem becomes to give nonblocking implementations of atomic objects of various types. Now the problem of correctness arises. Indeed, since incorrect solutions of concurrency problems do appear in the literature, the solutions must be verified and must be verifiable for others.

1.2 Verification: assertions or behaviours

There are two methods for the verification of concurrent algorithms. One method, the assertional approach, is to rely on invariants and variant functions, cf. [21]. The alternative, the behavioural approach, is to argue about execution sequences where certain actions precede other actions, cf. [18]. In [15], we introduced the terms synchrony and diachrony to distinguish these approaches.

The behavioural approach is closer to operational intuition and, often, also to the requirements that we want to satisfy. The assertional approach is more convenient for formal, possibly mechanical, verification. The two approaches do not mix conveniently, but they are complementary and, for every nontrivial algorithm, we need the right combination of them.

Indeed, the operational intuition often suggests that certain actions are needed to establish certain properties. The operational intuition is unreliable, however, when it comes to excluding undesirable interferences. Formal treatment based on execution sequences can be quite elegant, cf. [19], but it always requires analysis of all possible execution sequences and offers no structure to exclude some of these. For the latter purpose, we often need invariants but then we are back at the assertional approach. An assertional design method for concurrent algorithms is presented in [8].

In our view, the designer may use all kinds of intuition to come to a reasonable design or design step. Formal specification, analysis and proof in assertional terms can then be used to give the indispensable complementary evidence of correctness. We therefore aim at an assertional criterion for atomicity.

1.3 Grain of atomicity

Every formal verification is based on a mathematical model. In the case of concurrent algorithms where computations of different processes are interleaved nondeterministically, the most critical modelling assumptions are about the grain of atomicity, i.e., the sizes of the chunks that are guaranteed to remain together in all interleavings. It may be easy to prove the correctness of an algorithm under assumption of a coarse grain of atomicity, but this can impose too severe restrictions on the implementation. A fine grain of atomicity is easier to implement, but it may make it harder to prove the correctness of the algorithm.

The solution is to apply hierarchy: use fine grain atomicity to implement atomic commands of a coarser grain of atomicity. In other words, composite commands are accepted as atomic when they are behaviourally equivalent to atomic commands. This idea was proposed in [17, 22] under the names of sequential consistency and serializability. In [12], the formalization was sharpened to linearizability, which is a property of the accessed data objects. Lynch [20] introduces the term *atomic* for linearizability since there is no observable difference.

When constructing an atomic data object with a given specification, two ingredients must be combined: a sequential implementation of the required functional behaviour and a set of primitive atomic data objects to control the concurrency. The papers [11, 13], e.g., describe implementations of an arbitrary atomic data object, given a sequential implementation of its functional behaviour, and using as primitives read-write registers, consensus registers and a compare and swap register. In the present paper, we restrict ourselves to the construction of atomic read-write registers and we only use read-write registers with bounds on the numbers of readers and writers.

1.4 The applications

This investigation was triggered by Groote's remark in [10] that he did not know an elegant way to prove the correctness of Bloom's construction of a two-writer atomic register from two one-writer atomic registers. Indeed, Bloom's original proof in [3] is complicated, as well as behavioural. After some analysis, we constructed a simpler and assertional proof.

Inspired by the proof of Bloom's algorithm in [20] Sect. 13.4.4, which is behavioural, we here present a general assertional atomicity criterion for read-write objects. This criterion is then used to prove Bloom's algorithm [3] and the algorithm of Vitanyi–Awerbuch [24]. In both cases, a comparison with the behavioural proofs in Lynch's book [20] is in order. Our assertional proofs remain closer to the actual code and require verifications that are more easily formalized for a mechanical theorem prover. The behavioural proofs of [20] are more abstract, more conceptual, and better suited to interest and convince a human audience.

Bloom's algorithm is the construction of a two-writer atomic register for an arbitrary number of readers from two one-writer atomic registers, by means of one additional bit to express recentness. The algorithm of Vitanyi and Awerbuch is an implementation of a read-write atomic object with m ports that can both read and write, given m^2 registers, each for a single writer and a single reader. It needs unbounded integers for the reading ports to choose the most recent value.

1.5 Mechanical verification

In mathematics, handwritten proofs have served well for ages. Why then do we need mechanical theorem proving for concurrency? In our view, the reason is that, broadly speaking, in concurrency the combinatorial complexity is higher than in mathematics, although the conceptual complexity is lower. Even short code fragments may require dull case distinctions that must be handled carefully but can be dealt with effectively by a machine.

In concurrency, handwritten proofs have also the drawback that, when the program is modified only marginally, the whole proof is in jeopardy. This is not the case with mechanical proofs. If the old proof is applied to the new program, the prover automatically indicates where the old proof needs modification. It is our experience that, when the modification of the program is correct and not too big, a moderate modification of the proof may be sufficient.

After our work in [13–15], we now have a prelude [16] that defines the semantics of concurrency with shared variables in less than 120 lines for the theorem prover NQTHM of [4, 5]. This prelude can be used only for the assertional approach. Indeed, it mainly defines a function that, given a concurrent program and a list of shared variables, determines the possible atomic steps, i.e., how the global state is modified when any of the processes executes a single atomic command. For a specific program, we then let the prover verify a number of lemmas that specify how each variable is modified by an atomic command. After this, we use the prover to analyse whether proposed invariants are preserved. As shown in [14], progress can also be verified.

In this system, we model nondeterminacy in the following way. We use an auxiliary private variable *oracle*, which is a pair. Every nondeterministic choice is based on the first component of *oracle*. After each inspection, *oracle* is updated by means of the undefined function. The value of *oracle* is not allowed in the invariants. Since the second component of *oracle* remains hidden, arbitrary choice sequences can be generated in this way. Since *oracle* is a private variable, its usage can be combined atomically with actions on shared variables.

A side-effect of our work with NQTHM on concurrency proofs is that it has taught us sharper modes of reasoning about invariants.

1.6 Overview

In Sect. 2, we define atomicity of concurrent data objects, specialize to read-write objects, and then present and prove our criterion for atomicity of the latter, followed by a brief comparison with Lynch's atomicity criterion.

In Sect. 3, we describe Bloom's algorithm, transform it so as to apply our atomicity criterion, prove the atomicity criterion by means of a number of invariants, and give an indication how this proof is supplied to the theorem prover.

Section 4 contains the treatment of the algorithm of Vitanyi and Awerbuch along the same lines. This algorithm is a more straightforward illustration of the criterion, in the sense that its treatment requires less creativity. Section 5 contains concluding remarks.

2 Atomicity of concurrent objects

A concurrent data object is an automaton that holds a value, which can be accessed and modified via a number of ports. Atomicity of an object means that the object regarded as a black box cannot be distinguished from an object in which the operations take place instantaneously, even though invocations and responses may require some time. It follows that the implementer of an atomic object has two responsibilities: correct functional behaviour and atomicity.

In this paper, we treat atomicity of read-write objects. A read-write object is an object that only allows the value to be read or to be replaced by another value. Since we like to treat actual protocols by means of assertional reasoning, we present an assertional criterion for atomicity of read-write objects and we apply it to two of the examples in [20]. We shall prove the validity of our criterion by relating it to the formal definition of atomicity. We therefore start with the formal definitions of concurrent data objects and their atomicity. We use a terminology close to those of [12, 13, 20].

2.1 General definitions

A *variable type* $\mathcal{T}$ is a tuple $\mathcal{T} = \langle V, \mathit{Inv}, \mathit{Res}, v_0, f \rangle$ where V, *Inv* and *Res* are sets, v_0 is an element of V, and f is a function $f : V \times \mathit{Inv} \to V \times \mathit{Res}$. An *object* of type $\mathcal{T}$ is an automaton that holds a current value $v \in V$, which initially equals v_0. The set *Inv* holds the possible invocations of objects of type $\mathcal{T}$, the set *Res* is the set of responses. The effects of the invocations on the current value and the responses are determined by the transition function f in the following way. If an object of type $\mathcal{T}$ holds current value $v \in V$ and is invoked by $u \in \mathit{Inv}$, it gets a new value $w \in V$ and responds with $r \in \mathit{Res}$, as determined by $f(v, u) = (w, r)$.

The object is called *concurrent* if it can be accessed concurrently over a finite number of ports in such a way that an invocation over some port is eventually answered by a response over the same port. The port cannot be used for a new invocation before this response has come.

The observable behaviour of the object is determined by its set of executions. Executions are defined in the following way. Let us define *communication* to mean invocation or response. An *execution* of the object is a finite or infinite sequence e of pairs (q, u) with ports q and communications u. An execution e is *well-formed* iff, for every port q, the subsequence of e of the pairs with first component q alternates between invocation and response and starts with an invocation. The last invocation of q need not (yet) have a corresponding response.

Since invocations and responses over different ports may interleave, we have to specify the relation between invocations and responses carefully. The concurrent object is called *atomic* iff all its executions are *legal*, where, informally speaking, an execution is legal if its responses can be justified by postulating interleaved transitions of the object. Each transition must take place atomically at some moment between invocation and response. This is formalized as follows.

An *operation* is a triple $\langle u, w, r \rangle$ where u is an invocation, w is a value, and r is a response. We regard w and r as the new value and response resulting from invocation u. A *history* is a sequence of pairs (q, z) where each q is a port and each z is a communication or an operation.

If h is a history and p is a port, the *local history* h_p is the subsequence of h of the pairs with first component p, from which the (now redundant) first components p have been removed. A local history h_p is *well-formed* iff every response r in it is immediately preceded by some operation $\langle u, w, r\rangle$ and every operation $\langle u, w, r\rangle$ in it is immediately preceded by the invocation u and every invocation (except for the very first invocation) is immediately preceded by some response. So, the last invocation of p need not (yet) have a corresponding operation and the last operation of p need not (yet) have a corresponding response. A history h is *well-formed* iff its local histories h_q, for all ports q, are well-formed.

A history h *fits* an execution e iff e is obtained from h by removing all pairs (q, z) where z is an operation. Informally speaking, the operations can be removed since they are not observable, but they have to take place at some moment between invocation and response.

It remains to express that the object respects its specification as given by transition function f. For this purpose, we define the *operation history* h' of h to be the sequence of subsequent operations of history h; this sequence is obtained by first removing from h all pairs (q, u) with communications u, and then removing the port components. An operation history h' with elements $\langle u_i, w_i, r_i\rangle$ where i ranges over $0 \le i < m$, is defined to be *legal* iff $f(w_{i-1}, u_i) = (w_i, r_i)$ for all i, where $w_{-1} = v_0$ by convention.

A history h is defined to be *legal* iff its operation history h' is legal. An execution e is defined to be *legal* iff there exists a well-formed legal history h that fits it. A concurrent data object is defined to be *atomic* iff it is guaranteed that every occurring execution of it is legal.

Example. Assume each of the ports q_0, q_1, q_2, q_3 submits one invocation. The invocation of q_1 is treated before the invocation of q_0, but only q_0 receives the response. The execution e has the form: (q_0, u_0), (q_1, u_1), (q_2, u_2), (q_3, u_3), (q_0, r_1). The history h can have the form: (q_0, u_0), (q_1, u_1), $(q_1, \langle u_1, w_0, r_0\rangle)$, (q_2, u_2), $(q_0, \langle u_0, w_1, r_1\rangle)$, (q_3, u_3), (q_0, r_1). The corresponding operation history h' is $\langle u_1, w_0, r_0\rangle$, $\langle u_0, w_1, r_1\rangle$. The histories h and h' are legal iff $f(v_0, u_1) = (w_0, r_0)$ and $f(w_0, u_0) = (w_1, r_1)$. The local history h_{q_1} of port q_1 is u_1, $\langle u_1, w_0, r_0\rangle$.

Summarizing, the object is atomic iff all its executions are legal. An execution is legal iff it can be merged with a legal operation history, decorated with port names, to a well-formed history.

The definition of atomicity in [20] uses serialization points instead of pairs (q, z) where z is an operation, as above. It is equivalent to the present one since the values of q and z can be reconstructed from the other information. The definitions of linearizability in [12, 13] differ in other aspects, but are also equivalent.

Remark. An execution is called *sequential* iff it is well-formed and every invocation in it is immediately followed by the corresponding response, possibly except for the very last invocation. A concurrent object is called *sequentially correct* iff every sequential execution of it is legal. Sequential correctness is much weaker than atomicity, but it is also useful. An object that is merely sequentially correct, can be used by concurrent processes under mutual exclusion.

2.2 Atomic read-write objects

We now restrict our attention to a read-write variable type for values of type V. For such a type, we have only write commands and read commands. We model the write command $v := x$ by means of an invocation (Write, x) with the response *Ack*. We model a read command of the value v by means of an invocation *Read* answered by v. We now have that the set *Inv* of invocations is the disjoint union $(\{\mathit{Write}\} \times V) \cup \{\mathit{Read}\}$ and the set *Res* of responses is $\{\mathit{Ack}\} \cup V$. The transitions are specified by function f with $f(v, (\mathit{Write}, x)) = (x, \mathit{Ack})$ and $f(v, \mathit{Read}) = (v, v)$.

We turn to the question of proving atomicity for a concurrent read-write object, i.e., a concurrent object of a read-write variable type. In view of our preference for the assertional approach, we aim at a criterion in terms of states and invariants. Since the state often holds not enough information, we extend the state with additional variables that play no role in the algorithm but only serve in the proof. Such variables are called ghost variables [6], auxiliary variables [21] or history variables [1]. We prefer the first term, since "auxiliary" often has a general connotation and "history" suggests a specific role. Since ghost variables are conceptual only, arbitrary atomic commands can be extended with actions on ghost variables without danger to the atomicity.

We regard a port as a process or thread that executes the operations it participates in. The invocation of an operation takes place when the port starts the execution. The response coincides with the termination of the operation. The ports communicate via shared variables. They may also have some private variables. We use the general convention that shared variables are in type writer font and private variables are slanted. In predicates over the total state, we write $x.p$ for the value of private variable x of port p. Like ordinary variables, ghost variables can be shared or private.

We now give an assertional criterion for atomicity of a concurrent read-write object. The idea is to prove the atomicity (or linearizability) of the object by extending its implementation with actions on ghost variables in such a way that the order of the operations is sufficiently determined.

Setting. In order to prove atomicity, we provide every port with private integer ghost variables *start* and *sqn* (sequence number). We use `masq` to denote the maximal number *sqn* of the completed operations. More precisely, `masq` is a shared ghost variable with an arbitrary initial value t_0. Every port updates `masq` at the end of every operation by

$$\mathtt{masq} := \max(sqn, \mathtt{masq}) \ .$$

In every operation of a port, it updates its private variables *start* and *sqn* precisely once as described now. Every operation of a port starts by copying the current value of `masq` to *start*.

We assume that during every write operation, before the actual writing, the writing port determines some number for *sqn* and attaches this number as a kind of time stamp to the value to be written. In order to express that writers always choose different numbers for *sqn*, we introduce a shared ghost variable `snlist` of the type list of integers with $\mathtt{snlist} = [t_0]$ initially. Whenever a writer chooses a number for *sqn*, it appends this number to `snlist`. The freedom of writers in their choices of *sqn* will only be limited by the conditions in Theorem CRIT below.

Every port that copies a value, also copies the number attached. When a reading port interprets a value as the value read, it copies the attached number to its private variable *sqn*. The initial value v_0 of the implemented object is tagged with the initial number t_0. Since the connection between values and attached numbers is preserved by copying, we have that, when a port encounters a value (x, t), then $(x, t) = (v_0, t_0)$ or there is a writer that has written (x, t).

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

Proof. An object is atomic iff all its executions are legal. We therefore consider an arbitrary execution of the object, i.e., a sequence of invocations and responses resulting from the actions of a number of ports on the object. We have to prove that this execution is legal. The execution is well-formed since each port can execute at most one operation at a time: it needs to wait for a response before it can invoke again.

In order to prove that the execution is legal, we have to form a fitting legal history. We shall use the order of the numbers `masq` and *sqn* for this purpose. We first tag all communications with a number. Every invocation is tagged with the value of `masq` that is assigned to *start* at the moment of the invocation. Every response is tagged with the value of `masq` written at the end of the operation. Since `masq` is incremented only, the tags are ascending (i.e., non-decreasing) along the execution.

We now have to determine the operations and to form a fitting history by placing the operations in the execution. We first determine which operations to add, and tag these operations for adequate positioning later on. For every writing invocation, we add an operation to the history, even if the execution does not contain the corresponding response. For a reading invocation we only add an operation to the history when the execution contains the response.

For every writing invocation $u = (\mathit{Write}, x)$ of a port q, we introduce an operation $\omega = \langle u, x, \mathit{Ack}\rangle$ and we tag the pair (q, ω) with the number *sqn* chosen by writer q. For every reading response v with attached number t, say by port q, we introduce the operation $\omega = \langle \mathit{Read}, v, v\rangle$ and we tag the pair (q, ω) with the tag t. This determines the operations that have to be added to get a history. It remains to determine the order.

We first insert all reading operations into the execution in such a way that the attached numbers remain ascending and that every reading operation is placed between the corresponding invocation and response. This is possible because of the assumption $\mathit{start}.p \leq \mathit{sqn}.p$ and the final updates of `masq`.

We then insert all writing operations, in such a way that the attached time stamps remain ascending *and* that every write operation precedes all other operations tagged with the same number. This is possible since `snlist` never has multiple occurrences and, hence, different write operations have different tags. Since a writer always chooses $\mathit{sqn} > \mathit{start}$, the operation comes after the invocation. It comes before the response because of the final update of `masq`. This implies that the resulting history is well-formed. The history fits the execution by construction.

The resulting history is legal because of the assumption that, whenever a reader reads (x, t), then $(x, t) = (v_0, t_0)$ or there is a writer that has written (x, t). In the first case, the read operation takes place before all write operations of the history. In the second case, the latest write operation of the history has written (x, t). This concludes the proof of the theorem.

Remarks. A verifier who wants to apply Theorem CRIT to a given algorithm, has only to invent a prescription for the writers' choice of *sqn* and then to verify the three assumptions of the theorem. When the verifier is also the designer of the algorithm, he or she can use the assumptions of the theorem as guiding principles for the design.

The atomicity criterion Lemma 13.16 of [20] generates more complicated proof obligations than Theorem CRIT. It is also more general in the sense that it can be used to prove Theorem CRIT, but we do not describe that proof since it is more difficult than proving Theorem CRIT from scratch.

If writing occurs in the last atomic action of the write operation, `masq` is always the highest number that can be read by a reader. In that case, `masq` need not be updated in the final actions of readers. Below, this applies to Bloom's algorithm but not to the algorithm of Vitanyi and Awerbuch.

The proof of atomicity of the handshake register of Tromp [23] in [15] and the snapshot algorithm of [20] 13.4.5 can also be cast in the present setting.

It is not hard to prove that the type `integer` of the ghost variables *start*, *sqn*, and `masq` can be replaced by an arbitrary type with a linear order. In particular, one may use reals or lexically ordered strings.

3 Verification of Bloom's algorithm

In this Section, Theorem CRIT is used to prove atomicity of Bloom's register, cf. [3]. The problem solved by Bloom's algorithm is to construct, i.e., to simulate, an atomic register that can be modified by two writers and can be read by n readers, given two atomic registers that can be modified by one writer and can be read by $n+1$ readers.

Bloom solves this problem as follows. The two writing ports, called writers, are numbered 0 and 1. Each writer (say q) has its own one-writer atomic register $\mathtt{Reg}[q]$, which has one bit more than the register to be simulated. This additional bit (d) is used to indicate which of the two registers contains the current value (v) of the simulated register. We use *vw* for the value to be written and a private variable *vr* for the value to be read. We use the name *self* for the acting process. All ports have some additional private variables (e.g. d, x). We use the operator $\oplus$ to denote addition modulo 2. The writers and readers are given by the following code.

Write (*vw*) :
  read (d, x) **from** $\mathtt{Reg}[1 - \mathit{self}]$
  write $(d \oplus \mathit{self}, \mathit{vw})$ **to** $\mathtt{Reg}[\mathit{self}]$
  return *Ack* .

Read :
  read (d_0, x_0) **from** $\mathtt{Reg}[0]$
  read (d_1, x_1) **from** $\mathtt{Reg}[1]$
  read (d, vr) **from** $\mathtt{Reg}[d_0 \oplus d_1]$
  return *vr* .

The commands *Write* and *Read* are clearly wait-free since the code contains no loops or blocking commands. Note that when a port reads a pair from a register, it always uses only one component and ignores the other component of the pair.

The expression $d \oplus \mathit{self}$ in the writers' code is explained as follows. Since $d \oplus (d \oplus q) = (d \oplus d) \oplus q = q$, we have that, if the processes do not interfere, writer q establishes the postcondition $q = d_0 \oplus d_1$ where d_0, d_1 are the additional bits of the two registers. The readers use this property to determine which register to read. This shows that the object is at least sequentially correct. Note that the initial values of the additional bits are irrelevant for this.

If the processes do interfere, however, correctness is far from obvious. We proceed with the analysis in the following way. In 3.1, we transform the program to our notation, make some initial observations and establish the first invariant. In 3.2, we turn to the application of our atomicity criterion. We introduce ghost variables in the program and express the proof obligations in three invariants. Preservation of these invariants is proved by means of some auxiliary invariants in 3.3.

3.1 Initial transformation

For the ease of notation, the registers `Reg` are split in registers `dir` for the tag bits, and registers `val` for the values, according to the declarations

$$\begin{array}{l} \mathtt{val} : \ \textbf{array}\ \mathit{bit}\ \textbf{of}\ \mathit{value}\ , \\ \mathtt{dir} : \ \textbf{array}\ \mathit{bit}\ \textbf{of}\ \mathit{bit}\ , \end{array}$$

where $\mathit{bit} = \{0, 1\}$.

As is well known, actions on private variables can be combined atomically with actions on shared variables, cf. [2] Theorem 6.26. Since we want to verify the invariants mechanically, we introduce explicit program locations. The locations are numbered from 20 or 30 for easy finding in the code for the theorem prover. Each number stands for one atomic instruction. For the ease of the verification, we combine atomic commands whenever possible.

We need one private variable *loc* for both writers and readers. We thus represent Bloom's code as follows.

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{Write}\ (\mathit{vw}) :} \\ 20 & \mathit{loc} := \ \mathtt{dir}[1 - \mathit{self}] \oplus \mathit{self}\ ; \\ 21 & \mathtt{val}[\mathit{self}] := \ \mathit{vw}\ ; \\ & \mathtt{dir}[\mathit{self}] := \ \mathit{loc}\ ; \\ 22 & \textbf{goto}\ 20\ . \end{array}$$

In action 20, the writer determines the value of the additional bit *loc* that stands for the expression $d \oplus \mathit{self}$ in Bloom's code. Action 21 represents the write action to $\mathtt{Reg}[\mathit{self}]$ and is therefore regarded as a single atomic command. The final command is chosen to model that a writing port can write again. Note that, when it does so, it may use a fresh value *vw* to write. In our NQTHM modelling, *vw* is updated nondeterministically with the first component of *oracle*, see Sect. 1.5.

In order to show that the order of the first two read actions of the readers is irrelevant, we give each reader a private variable *pr* to indicate where to read first. The value of *pr* is chosen nondeterministically, again by means of *oracle*.

Read :
30 $\quad$ $loc := \mathtt{dir}[1 - pr]$;
31 $\quad$ $loc := loc \oplus \mathtt{dir}[pr]$;
32 $\quad$ $vr := \mathtt{val}[loc]$;
33 $\quad$ **choose** *pr* **in** $\{0, 1\}$;
goto 30 .

In order to give some feeling for the protocol, we start with a bottom-up analysis. Recall that the value of a private variable x of process q is denoted $x.q$. In particular, $pc.q$ is the program location of process q.

We first investigate what is read by a reader that performs the actions 30, 31, 32, when no writer has an interleaving action 21. In that case, the reader reads the value at index $loc = \mathtt{dir}[0] \oplus \mathtt{dir}[1]$. Anthropomorphically speaking, such a fast reader acts as if $\mathtt{dir}[0] \oplus \mathtt{dir}[1]$ is the *latest writer* of the register. We therefore define the state function $\mathtt{LaWr}$ by

$$\mathtt{LaWr} = \mathtt{dir}[0] \oplus \mathtt{dir}[1] .$$

When a writer $q = \mathtt{LaWr}$ executes action 20, it establishes $pc.q = 21$ and $loc.q = \mathtt{dir}[1 - q] \oplus \mathtt{LaWr} = \mathtt{dir}[q]$. It turns out that this property is an invariant of the system:

$$\text{(Bloom)} \quad q = \mathtt{LaWr} \quad \wedge \quad pc.q = 21 \quad \Rightarrow \quad loc.q = \mathtt{dir}[q] .$$

This is shown as follows. Apart from action 20 by q itself (as treated just now), the only threat to predicate (Bloom) is when a port $p \neq q$ executes 21 and thus modifies $\mathtt{LaWr}$. It modifies $\mathtt{LaWr}$ only if $loc.p \neq \mathtt{dir}[p]$. Predicate (Bloom) therefore implies that $p \neq \mathtt{LaWr}$ initially. Since p modifies $\mathtt{LaWr}$, it becomes itself equal to $\mathtt{LaWr}$ and then has $pc.p = 22$. This shows that, indeed, (Bloom) is preserved.

Remark. It is not true that, conversely, $q \neq \mathtt{LaWr}$ and $pc.q = 21$ implies $loc.q \neq \mathtt{dir}[q]$. In fact, if $q = \mathtt{LaWr}$ and $pc.q = 21$, the other writer may modify $\mathtt{LaWr}$, but it cannot modify $loc.q$ or $\mathtt{dir}[q]$.

3.2 The main analysis

We turn to the proof of the protocol. In view of Theorem CRIT, we give every port a private ghost variable *sqn* to hold a number. We introduce a shared ghost variable $\mathtt{time}$ and we let the sequence number of a writer be obtained by the action

$$\mathtt{time}\ \mathtt{++}\ ; \quad sqn := \mathtt{time}\ ;$$
$$\mathtt{snlist} := sqn : \mathtt{snlist}\ .$$

Here, we use the operator `++` for incrementation and : for adding an element to a list. In the concluding write action 21, the sequence number is tagged as a time stamp to the value written. For this purpose, we introduce a shared ghost variable `tag` for the time stamps, according to the declaration

$$\texttt{tag} : \ \textbf{array}\ \mathit{bit}\ \textbf{of}\ \mathit{integer}\ .$$

We then extend action 21 with

$$\texttt{tag}[\mathit{self}] := \ \mathit{sqn}\ ;$$
$$\texttt{masq} := \max(\mathit{sqn}, \texttt{masq})\ .$$

We use the analysis of Sect. 3.1 to decide at which moment a writer gets its sequence number. If writer `LaWr` executes 20 and the other writer then modifies `LaWr` by executing 21, we must justify the behaviour of fast readers by giving the second writer a later sequence number than the first one. We therefore give a writer its new sequence number at action 20 if it then equals `LaWr`. Otherwise, the sequence number is obtained in action 21. The question whether the writer equals `LaWr` can be encoded by the test $\mathit{loc} = \texttt{dir}[\mathit{self}]$ after the assignment to *loc* in 20. We thus get the following extended code for the writers.

```
Write (vw) :
20    start := masq ;
      loc := dir[1 − self] ⊕ self ;
      if loc = dir[self] then
         time ++ ;   sqn := time ;
         snlist := sqn : snlist fi ;
21    if loc ≠ dir[self] then
         time ++ ;   sqn := time ;
         snlist := sqn : snlist fi ;
      val[self] := vw ;
      dir[self] := loc ;
      tag[self] := sqn ;
      masq := max(sqn, masq) ;
22    goto 20 .
```

Since $\mathit{loc}.q$ and $\texttt{dir}[q]$ are modified only by writer q itself, every write action obtains precisely one sequence number. Note the update of the ghost variable `masq` according to the setting of Theorem CRIT.

When a reader starts reading, its private ghost variable *start* becomes a copy of `masq`. When the reader executes 32, the private ghost variable *sqn* records the time stamp of the value that is read. The program for the readers therefore becomes

Read :
30 $\quad$ *start* := `masq` ;
$\qquad$ *loc* := `dir`$[1 - pr]$;
31 $\quad$ *loc* := *loc* $\oplus$ `dir`$[pr]$;
32 $\quad$ *vr* := `val`$[loc]$; $\quad$ *sqn* := `tag`$[loc]$;
$\qquad$ `masq` := max(*sqn*, `masq`) ;
33 $\quad$ **choose** *pr* **in** $\{0, 1\}$; $\quad$ **goto** 30 .

At this point one easily verifies the setting of Theorem CRIT. In particular, whenever a reader reads a pair (x, t) in instruction 32, there has been a writer that wrote the same pair in instruction 21. This follows from the atomicity of the instructions 21 and 32 and the observation that the arrays `val` and `tag` are modified only in 21.

Remark. This atomicity might have been more apparent when we had represented the pair of arrays `val`, `tag` by an array of pairs. The present set-up was chosen since `tag` is a ghost variable whereas `val` is an actual variable.

According to Theorem CRIT, it now suffices to prove the invariants

(Iq0) $\quad pc.q = 22 \quad \Rightarrow \quad start.q < sqn.q$,
(Iq1) $\quad pc.q = 33 \quad \Rightarrow \quad start.q \leq sqn.q$,
(Iq2) $\quad$ *IsSet* (`snlist`) ,

where predicate *IsSet* determines whether its argument is a list without multiple occurrences.

3.3 The verification

In this subsection we prove that the predicates (Iq0), (Iq1), (Iq2) are invariants of the system. This requires the invention of a number of other invariants. We can assume that all invariants hold as a precondition for each atomic step and then have to prove that they hold in the postcondition. Often this requires detailed case distinctions. The proof given below matches the formal proof [16] that has been verified with the theorem prover NQTHM. One may notice that, for a theorem prover, boring trivialities and subtle case distinctions are not far apart.

The method used is as follows. We start with the invariants postulated, here (Iq0), (Iq1), and (Iq2). For each invariant, we then verify whether each of the atomic commands preserves it. When some atomic command may falsify it, we postulate some auxiliary invariants to hold in the precondition of that atomic command that prevent this falsification. These auxiliary invariants should be as weak as possible. Indeed, they must hold initially, and we have to maximize the likelyhood that they in turn are preserved by all atomic actions. When the resulting list contains an invariant that is implied by other invariants, such an invariant can be removed from the list.

In this way, the invariants appear in an unsystematic order. For example, looking ahead, one can see invariants (Jq3) and (Jq6), which can be combined to

$$q \in \{0,1\} \quad \Rightarrow \quad \mathtt{tag}[q] \leq \mathit{sqn}.q \leq \mathtt{time}\ .$$

We separate such invariants since we need them at different points and since the proof of invariance is easier when they are separated.

A predicate P is said to be *threatened* by a command A iff it is not true that A started with precondition P always has postcondition P. If P is a predicate threatened by a command A, we need more information than P alone to prove its invariance, i.e., we have to postulate some other invariant Q such that A started with precondition $P \wedge Q$ always has postcondition P.

Since *pc*, *start*, and *sqn* are private variables, predicate (Iq0) is threatened only when a writing port q executes 21. If it does so, it preserves (Iq0) if and only if we also have the invariants

(Jq0) $\quad \mathit{pc}.q = 21 \quad \wedge \quad \mathit{loc}.q = \mathit{dir}[q] \quad \Rightarrow \quad \mathit{start}.q < \mathit{sqn}.q\ ,$
(Kq0) $\quad \mathit{pc}.q = 21 \quad \wedge \quad \mathit{loc}.q \neq \mathit{dir}[q] \quad \Rightarrow \quad \mathit{start}.q \leq \mathtt{time}\ .$

We first note that (Kq0) is implied by postulating the slightly stronger invariants

(Jq1) $\quad \mathit{pc}.q \in \{21, 31, 32\} \quad \Rightarrow \quad \mathit{start}.q \leq \mathtt{masq}\ ;$
(Jq2) $\quad \mathtt{masq} \leq \mathtt{time}\ .$

Predicate (Jq0) is threatened by command 20, but preserved because of (Jq2). Since *start* is set to $\mathtt{masq}$ in 20 and 30 and $\mathtt{masq}$ is incremented only, predicate (Jq1) is an invariant. Predicate (Jq2) is threatened only by 21 and 32. It is preserved at these points because of the obvious invariants for writers

(Jq3) $\quad q \in \{0,1\} \quad \Rightarrow \quad \mathit{sqn}.q \leq \mathtt{time}\ ;$
(Jq4) $\quad q \in \{0,1\} \quad \Rightarrow \quad \mathtt{tag}[q] \leq \mathtt{masq}\ .$

This concludes the proof of invariance of (Iq0).

Since *pc*, *start*, and *sqn* are private variables, predicate (Iq1) is threatened only by action 32. It is preserved by 32 because of the new postulate

(Jq5) $\quad \mathit{pc}.q = 32 \quad \Rightarrow \quad \mathit{start}.q \leq \mathtt{tag}[\mathit{loc}.q]\ .$

Predicate (Jq5) is threatened by 21 and 31. It is preserved when p executes 21 with $\mathit{loc}.p \neq \mathtt{dir}[p]$ because of (Jq1) and (Jq2). It is preserved by p at 21 with $\mathit{loc}.p = \mathtt{dir}[p]$ because of the new postulate

(Jq6) $\quad q \in \{0,1\} \quad \Rightarrow \quad \mathtt{tag}[q] \leq \mathit{sqn}.q\ .$

Predicate (Jq5) is preserved at 31 because of the new postulate

(Jq7) $\quad \mathit{pc}.q = 31 \quad \Rightarrow \quad \mathit{start}.q \leq \mathtt{tag}[\mathit{loc}.q \oplus \mathtt{dir}[\mathit{pr}.q]]\ .$

Since $\mathtt{tag}[q]$ and $sqn.q$ are modified only by writer q, predicate (Jq6) is threatened only by action 20. It is preserved because of (Jq2) and (Jq4).

Predicate (Jq7) is threatened only by the actions 21 and 30. Recall that $\mathtt{LaWr} = \mathtt{dir}[0] \oplus \mathtt{dir}[1]$. Preservation of (Jq7) at 30 now follows from the new invariant

$$\text{(Jq8)} \qquad \mathtt{masq} = \mathtt{tag}[\mathtt{LaWr}] \,,$$

which, as a justification of the acronym $\mathtt{LaWr}$, expresses that the time stamp of $\mathtt{LaWr}$ is the highest time stamp.

Preservation of (Jq7) when writer p executes 21 is complicated, since both $\mathtt{tag}$ and $\mathtt{dir}$ can be modified by 21. It is shown as follows. If $\mathtt{dir}$ is not modified, i.e., if $loc.p = \mathtt{dir}[p]$, it suffices to use (Jq6). If $\mathtt{dir}$ is modified, let Y be the new value of $loc.q \oplus \mathtt{dir}[pr.q]$. If $p = Y$, preservation of (Jq7) follows from (Jq1) and (Jq2). Otherwise, we use the invariant (Bloom) verified in Sect. 3.1. This invariant implies that $p = 1 - \mathtt{LaWr}$. Therefore, $Y = \mathtt{LaWr}$ and preservation of (Jq7) follows from (Jq1) and (Jq8).

Predicate (Jq8) is threatened only at 21 and 32. It is preserved at 32 since (Jq4) implies that $\mathtt{masq}$ is not modified in 32. Preservation of (Jq8) when a writer p executes 21 is shown as follows. If $p = \mathtt{LaWr}$ then (Bloom) implies that $loc.p = \mathtt{dir}[p]$. Therefore $\mathtt{LaWr}$ remains p and preservation of (Jq8) follows from (Jq6). If $p \neq \mathtt{LaWr}$ and $loc.p \neq \mathtt{dir}[p]$, preservation of (Jq8) follows from (Jq2). In the remaining case, with $p \neq \mathtt{LaWr}$ and $loc.p = \mathtt{dir}[p]$, we use the new postulate that $\mathtt{LaWr}$ is the only writer that can have $sqn.q > \mathtt{masq}$:

$$\text{(Jq9)} \qquad q \in \{0,1\} \quad \wedge \quad \mathtt{masq} < sqn.q \quad \Rightarrow \quad q = \mathtt{LaWr} \,.$$

Predicate (Jq9) seems to be threatened by the actions 20 and 21. If p executes 20 and increments $sqn.p$, it becomes $\mathtt{LaWr}$, so that (Jq9) is preserved. If q executes 21, it sets $\mathtt{masq} \geq sqn.q$. Finally, if $p \neq q$ executes 21, it preserves (Jq9) because of (Jq3) applied to q. This concludes the proof of invariance of (Iq1).

The invariance of (Iq2) easily follows from the obvious invariant

$$\text{(Jq10)} \qquad x \in \mathtt{snlist} \quad \Rightarrow \quad x \leq \mathtt{time} \,.$$

It remains to initialize the variables such that all invariants hold. For the ghost variables $\mathtt{time}$ and $\mathtt{masq}$, we take the initial values $t_0 = 1$. For the two writers, q, we specify initially $pc.q = 20$ and $\mathtt{tag}[q] = sqn.q = t_0$. For the readers, it suffices to specify that $pc = 30$ initially.

Remark. The initialization of $sqn.q$ of the writers is needed because the invariants (Jq3), (Jq6), and (Jq9) are stronger than necessary. A stricter analysis shows that these inequalities are needed only when $pc.q = 21$ and $loc.q = \mathtt{dir}[q]$.

The above proof uses implicitly that 0 and 1 are the only writing ports. The mechanical proof makes this explicit by requiring the obvious additional invariant

$$pc.q \in \{20, 21, 22\} \quad \equiv \quad q \in \{0, 1\} \; .$$

The mechanical proof also needs the type invariants that *loc* and *pr* are bits.

The mechanical proof `bloom` in [16] is an NQTHM events file, cf. [4, 5]. The method employed is the same as used in [14, 15]. The file `bloom` is the input to the theorem prover. It consists of around 1250 lines. After a call of the prelude for concurrency that was mentioned in Sect. 1.5, the first part of this file (340 lines) contains the program and the analysis of how the variables are modified in the atomic steps. The proofs of the individual invariants require 630 lines. The remainder is taken by the proof that the individual invariants combine to one global invariant (140 lines) and the proof that the global invariant can be initialized (140 lines). This remainder is an administrative check of global consistency.

4 The Vitanyi-Awerbuch algorithm

In this section, we use Theorem CRIT to prove the atomicity of the algorithm of Vitanyi and Awerbuch [24], see also [20], Sect. 13.4.5.

This algorithm is an implementation of a read-write atomic object with m ports that can both read and write. It uses m^2 registers, each for a single writer and a single reader. It is based on the declarations

type
    Port $= [0 \,..\, m-1]$;
    Reg $=$ **record**
        val : *Value* ;
        tag : *Integer* ;
    end ;
var x : **array** *Port*, *Port* **of** *Reg* ;

Register $\mathsf{x}[p, q]$ is a variable that can be read only by port p and written only by port q. All registers are initially equal to (v_0, t_0) where v_0 is the initial value of the abstract object and t_0 is some initial number.

In this algorithm, the fields *tag* are actual variables that must be able to hold arbitrary large integers. These fields serve to hold the tags used in our atomicity criterion. The algorithm also uses private variables that play the roles of the ghost variables *sqn* of the atomicity criterion. These variables are therefore named *sqn* here.

The algorithm works as follows. A writing port that has to write a value *vw*, first reads all tags that it can read and then chooses a number *sqn* bigger than all of them. To ensure that different writers always choose different numbers, the writer keeps *sqn* **mod** m equal to its process identifier *self*. It subsequently writes the pair (vw, sqn) to all available registers. These design decisions could have been inspired directly by Theorem CRIT. Though Vitanyi and Awerbuch clearly did not need it, this is the guidance to the designer that we suggested in the introduction.

```
Write (vw) :
    num := 0 ;
    for all j in Port do
        num := max(num, x[self, j].tag)  od ;
    sqn := (num div m + 1) * m + self ;
    for all i in Port do
        x[i, self] := (vw, sqn)  od ;
    return Ack .
```

A reader reads the record with the highest number and also transfers that record to all its writing registers. At this point, we cannot see this, but the latter activity is needed so that the writing ports can obtain a good estimate of the ghost variable `masq` of the atomicity criterion.

```
Read :
    num := 0 ;
    for all j in Port do
        if num ≤ x[self, j].tag then
            dat := x[self, j] ;
            num := dat.tag  fi od ;
    for all i in Port do
        x[i, self] := dat  od ;
    return dat.val .
```

These implementations of *Write* and *Read* contains no blocking commands or unbounded repetitions. They have a time complexity of order m, the number of ports. Therefore, both writing and reading are wait-free.

As before, one easily verifies the setting of Theorem CRIT. In particular, whenever a reader reads a pair (v, t) in its first **for** loop, it was the initial value (v_0, t_0) or there has been a writer that wrote the pair (v, t) in its second **for** loop.

4.1 Initial transformation

We turn to the verification of the assumptions of Theorem CRIT. For convenience, we represent the array `x` of pairs by a pair of arrays `val` and `tag` in the obvious way. So, now, array `tag` is an actual variable, not a ghost variable as in Sect. 3. Yet, its elements will figure as the tags of the atomicity criterion. The private variables *sqn* of the writing ports are also actual variables. Since we need invariants during the **for** loops, we introduce a private variable *lis* for the set of port numbers that yet have to be treated in the loop.

```
Write (vw) :
20   start := masq ;   num := 0 ;   lis := Port ;
21   if IsEmpty(lis) then goto 22 else
         choose j ∈ lis ;   lis := lis \ {j} ;
         num := max(num, tag[self, j]) ;
         goto 21 fi ;
22   sqn := (num div m + 1) * m + self ;   lis := Port ;
23   if IsEmpty(lis) then goto 24 else
         choose i ∈ lis ;   lis := lis \ {i} ;
         val[i, self] := vw ;
         tag[i, self] := sqn ;
         goto 23 fi ;
24   snlist := sqn : snlist ;
     masq := max(sqn, masq) ;
     goto 20 or 30 .
```

The final **goto** is chosen to model that, after writing or reading, a port may decide to write or read again. In our NQTHM modelling, the choice between 20 and 30 is determined by the *oracle* as explained in 1.5. We could have done the same for the choices of j and i from *lis*, but we did not regard that as worth the effort. Indeed, looking at the proof below, one easily sees that the order of treating the elements of *lis* is irrelevant. For the sake of symmetry, the value *dat* determined by the reader is represented by the pair of private variables (vr, sqn).

```
Read :
30   start := masq ;   num := 0 ;   lis := Port ;
31   if IsEmpty(lis) then goto 32 else
         choose j ∈ lis ;   lis := lis \ {j} ;
         if num ≤ tag[self, j] then
             vr := val[self, j] ;
             num := tag[self, j] fi ;
         goto 31 fi ;
32   lis := Port ;   sqn := num ;
33   if IsEmpty(lis) then goto 34 else
         choose i ∈ lis ;   lis := lis \ {i} ;
         val[i, self] := vr ;
         tag[i, self] := sqn ;
         goto 33 fi ;
34   masq := max(sqn, masq) ;
     goto 20 or 30 .
```

It is easy to see that we have followed the prescriptions of Theorem CRIT with respect to the assignments to *start*, *sqn*, `masq`, and `snlist`.

According to Theorem CRIT, it now suffices to prove the invariants

(Lq0) $\quad pc.q \in \{23, 24\} \quad \Rightarrow \quad start.q < sqn.q$;
(Lq1) $\quad pc.q = 33 \quad \Rightarrow \quad start.q \le sqn.q$
(Lq2) $\quad IsSet\,(\texttt{snlist})$.

We strengthened (Lq0) by including location 24 for the sake of later convenience.

4.2 Verification

We use the same method as for Bloom's algorithm to verify preservation of the invariants.

In view of the commands 22 and 32, preservation of (Lq0) and (Lq1) follows when we also have the invariant

(Mq0) $\quad pc.q \in \{22, 32\} \quad \Rightarrow \quad start.q \le num.q$.

In order to prove preservation of (Mq0) when q executes 21 or 31, we need an invariant that incorporates the tags that are yet to be encountered in that loop. Indeed, preservation of (Mq0) follows from the new invariant

(Mq1) $\quad pc.q \in \{21, 31\} \quad \Rightarrow \quad start.q \le \max\,(num.q,$
$\quad (\texttt{MAX}\ \ j \in lis.q :: \texttt{tag}[q, j]))$.

It is easy to see that (Mq1) is preserved by the commands 21 and 31: it is a kind of loop invariant. Predicate (Mq1) is threatened by the modifications of *start*, *num*, *lis* in 20 and 30 and by the modifications of `tag` in 23 and 33. It is preserved by the former when we postulate the invariant

(Mq2) $\quad \texttt{masq} \le (\texttt{MAX}\ \ j \in Port :: \texttt{tag}[q, j])$.

It is preserved by the latter when the modifications of `tag` are always incrementations, as will follow from the invariant

(Nq0) $\quad pc.q \in \{23, 33\} \quad \wedge \quad i \in lis.q \quad \Rightarrow \quad \texttt{tag}[i, q] \le sqn.q$.

This predicate follows from (Lq0) and (Lq1) when we postulate the invariant

(Mq3) $\quad pc.q \in \{23, 33\} \quad \wedge \quad i \in lis.q \quad \Rightarrow \quad \texttt{tag}[i, q] \le start.q$.

Since $\texttt{tag}[i, q]$ is modified only by port q, preservation of (Mq3) follows from the invariant

(Mq4) $\quad pc.q \in \{21, 22, 31, 32\} \quad \Rightarrow \quad \texttt{tag}[i, q] \le start.q$.

Preservation of (Mq4) follows from the invariant

(Mq5) $\quad pc.q \in \{20, 30\} \quad \Rightarrow \quad \texttt{tag}[i, q] \le \texttt{masq}$.

Preservation of (Mq5) in its turn follows from the invariant

(Mq6) $\quad pc.q \in \{24, 34\} \quad \Rightarrow \quad \mathtt{tag}[i, q] = sqn.q$.

Finally, preservation of (Mq6) follows from the obvious invariant

(Mq7) $\quad pc.q \in \{23, 33\} \quad \wedge \quad i \notin lis.q \quad \Rightarrow \quad \mathtt{tag}[i, q] = sqn.q$.

It remains to prove preservation of (Mq2). This predicate is threatened by the assignments to `masq` and `tag`. It is preserved when port p executes 24 or 34 since $sqn.p = \mathtt{tag}[q, p]$ holds by (Mq6). It is preserved by assignments to `tag` because of (Nq0).

We turn to the invariant (Lq2) that expresses the uniqueness of the sequence numbers. Here we use that each writing port q only uses *sqn* with $sqn.q \ \mathbf{mod}\ m = q$, as expressed in the obvious invariant

(Mq8) $\quad pc.q \in \{23, 24\} \quad \Rightarrow \quad sqn.q \ \mathbf{mod}\ m = q$.

In order to prove preservation of (Lq2), it suffices to prove the predicate

(Nq1) $\quad pc.q = 24 \quad \Rightarrow \quad sqn.q \notin \mathtt{snlist}$.

In order to prove (Nq1), we introduce the set

$$\mathsf{SN}(q) = \{x \in \mathtt{snlist} \mid x \ \mathbf{mod}\ m = q\}$$

and postulate that *start*.q is an upper bound of $\mathsf{SN}(q)$:

(Mq9) $\quad x \in \mathsf{SN}(q) \quad \wedge \quad pc.q \in \{21, 22, 23, 24\} \quad \Rightarrow \quad x \leq start.q$.

Predicate (Nq1) is implied by (Mq8), (Mq9), and (Lq0) as is shown in

$$\begin{array}{ll}
 & pc.q = 24 \quad \wedge \quad sqn.q \in \mathtt{snlist} \\
\Rightarrow & \{(\text{Mq8})\} \\
 & pc.q = 24 \quad \wedge \quad sqn.q \in \mathsf{SN}(q) \\
\Rightarrow & \{(\text{Mq9})\} \\
 & pc.q = 24 \quad \wedge \quad sqn.q \leq start.q \\
\Rightarrow & \{(\text{Lq0})\ \} \\
 & false\ .
\end{array}$$

It is here that we use that (Lq0) has been strengthened to cover location 24.

The set $\mathsf{SN}(q)$ is modified only when port q itself executes command 24, but then $pc.q$ becomes 20 or 30. Predicate (Mq9) is therefore threatened only when port q itself executes 20 and thus gets $pc.q = 21$. At that point, preservation of (Mq9) follows from the obvious invariant that `masq` is an upper bound of `snlist`:

(Mq10) $\quad x \in \mathtt{snlist} \quad \Rightarrow \quad x \leq \mathtt{masq}$.

It is easy to see that the invariants can be initialized.

This concludes the verification of the assumptions of Theorem CRIT for the Vitanyi-Awerbuch algorithm and thus proves that the algorithm implements an atomic read-write register.

The mechanical proof `vitanyi` we constructed for this algorithm can be obtained from [16]. The proofs of the invariants are somewhat easier than in `bloom`, but the events file is longer (1482 lines) since it requires arithmetic for command 22 and a quantification in invariant (Mq1). We were able to mechanize our handwritten proof in less than two days since it was almost flawless and we had the arithmetic for command 22 available. The one flaw in our handwritten proof was an insufficient candidate for (Mq10).

5 Concluding remarks

We presented and proved an assertional criterion for atomicity of read-write objects (Theorem CRIT). This criterion enabled us to prove the correctness of Bloom's algorithm for two writers and of the algorithm of Vitanyi and Awerbuch for a bounded number of readers and writers. The proofs are simple enough for straightforward verification with a mechanical theorem prover.

It seems likely that our criterion is strictly weaker than the behavioural criterion Lemma 13.16 of [20]. We believe, however, that it is strong enough for every atomic read-write object that is not specifically designed to be hard to prove.

The proof for Bloom's algorithm is based on the new (but natural) idea to order the write operations as perceived by fast readers and to encode this order by actions on ghost variables. The key to this was the invariant (Bloom), the only invariant for Bloom's algorithm that mentions no ghost variables. In Bloom's proof [3] the order of writing is not defined by fast readers but by the actual infinite execution. This may have been the reason for Groote to suggest in [10] to phrase the proof in terms of prophecy variables (see [1]).

The criterion was even more useful in the case of the algorithm of Vitanyi and Awerbuch. For, in this case, the sequence numbers could be found as actual variables of the algorithm. With our system, we always have to invent the invariants, but in this case that was easy. Conversely, as we have indicated, our criterion could have suggested the design of this algorithm.

It is a fairly straightforward exercise to apply the criterion to prove atomicity of the snapshot algorithm of [20] 13.4.5 or of Tromp's handshake register [15, 23].

Acknowledgements. We are grateful for comments and suggestions of Gao Hui, Jan Jongejan, Jan Eppo Jonker, and three anonymous referees.

References

1. Abadi M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science **82**, 253–284 (1991)
2. Apt, K.R., Olderog, E.-R.: Verification of Sequential and Concurrent Programs. Berlin Heidelberg New York: Springer 1991
3. Bloom, B.: Constructing two-writer atomic registers. IEEE Transactions on Computers **37**, 1506–1514 (1988)
4. Boyer, R.S., Moore, J S.: A Computational Logic Handbook. Boston: Academic Press 1988
5. Boyer, R.S., Moore, J S.: A Computational Logic Handbook, Authorized Excerpts from a Proposed Second Edition, to be obtained by ftp from Computational Logic Inc. Information available at `nqthm-request@cli.com`
6. Clint, M.: Program proving: coroutines. Acta Informatica **2**, 50–63 (1973)
7. Dijkstra, E.W.: Co-operating sequential processes. In: F. Genuys (ed.): Programming Languages (NATO Advanced Study Institute) pp. 43–112. London: Academic Press 1968
8. Feijen, W.H.J., Gasteren, A.J.M. van: On a method of multiprogramming. New York: Springer 1999
9. Francez, N.: Fairness. Berlin Heidelberg New York: Springer 1986
10. Groote, J.F.: We moeten software leren beheersen. Inaugural Address, University of Eindhoven, 1999
11. Herlihy, M.P.: Wait–free synchronization. ACM Trans. on Program. Languages and Systems **13**, 124–149 (1991)
12. Herlihy M.P., Wing J.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. on Program. Languages and Systems **12**, 463–492 (1990)
13. Hesselink, W.H.: Wait–free linearization with a mechanical proof. Distrib Comput **9**, 21–36 (1995)
14. Hesselink, W.H.: Theories for mechanical proofs of imperative programs. Formal Aspects of Computing **9**, 448–468 (1997)
15. Hesselink, W.H.: Invariants for the construction of a handshake register. Information Processing Letters **68**, 173–177 (1998)
16. Hesselink, W.H.: www.cs.rug.nl/˜wim/mechver/imperative/, the NQTHM events files `concprelude`, `bloom`, and `vitanyi`
17. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**, 690–691 (1979)
18. Lamport, L.: On interprocess communication, Parts I and II. Distrib. Comput. **1**, 77–101 (1986)
19. Lamport, L.: How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans. Computers **46**, 779–782 (1997)
20. Lynch, N.A.: Distributed Algorithms. San Francisco: Morgan Kaufman 1996
21. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Informatica **6**, 319–340 (1976)
22. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**, 631–653 (1979)
23. Tromp, J.: How to construct an atomic variable. In: J.-C. Bermond, M. Raynal (Eds.): Distributed Algorithms, Proceedings Nice. Lecture Notes in Comput. Sci., Vol. 392, pp. 292–302. Berlin: Springer 1989
24. Vitányi, P.M.B., Awerbuch, B.: Atomic shared register access by asynchronous hardware. In *27th Annual Symposium on Foundations of Computer Science*, pages 233–243, Toronto, Ontario, Canada, 1986. IEEE, Los Alamitos, Calif. Corrigendum in 28th Annual Symposium on Foundations of Computer Science, page 487, Los Angeles, 1987