Proving Atomicity: An Assertional Approach

Gregory Chockler Nancy Lynch Sayan Mitra Joshua Tauber

MIT Computer Science and Artificial Intelligence Lab
Cambridge, MA 02139
{grishac,lynch,mitras,josh}@theory.csail.mit.edu

Abstract

Atomicity (or *linearizability*) is a commonly used consistency criterion for distributed services and objects. Although atomic object implementations are abundant, proving that algorithms achieve atomicity has turned out to be a challenging problem. In this paper, we initiate the study of systematic ways of verifying distributed implementations of atomic objects, beginning with read/write objects (registers). Our general approach is to replace the existing operational reasoning about events and partial orders with assertional reasoning about invariants and simulation relations. To this end, we define an abstract state machine that captures the atomicity property and prove correctness of the object implementations by establishing a simulation mapping between the implementation and the specification automata. We demonstrate the generality of our specification by showing that it is implemented by three different read/write register constructions (the message-passing register emulation of Attiya, Bar-Noy and Dolev, its optimized version based on real time, and the shared memory register construction of Vitanyi and Awerbuch), and by a general atomic object implementation based on the Lamport's replicated state machine algorithm.

Contact author:	Gregory Chockler
Address:	MIT CSAIL, the Stata Center
	32 Vassar St. (32-G696)
	Cambridge, MA 02139
E-mail:	grishac@csail.mit.edu
Phone:	617-253-9302
Track:	Regular
Eligible student paper:	Yes (Sayan Mitra is a student.)

1 Introduction

Many distributed and network-based services can be modeled as shared objects accessible to (possibly remote) clients through well-defined interfaces. Atomicity [13, 18] (also known as *linearizability* [7]) is a desirable property for such objects as it allows clients to perceive the operations invoked in each run as occurring in some sequential order. This perception makes it easier to understand the behavior of a system using distributed services. To achieve high availability in a distributed system and to tolerate failures, atomic services are typically implemented by distributed algorithms. Many distributed algorithms, using a range of techniques, have been proposed for implementing atomic objects; see, for example, [14, 12, 32, 24, 29, 28, 7, 31, 11, 16, 15, 19, 20, 2].

Although atomic object implementations are abundant, proving that algorithms achieve atomicity has turned out to be a challenging problem. Most existing proofs for such algorithms are long, subtle, and difficult to understand and check. As evidence of the difficulty, we note that several published proofs for implementations of atomic registers in the shared memory model have later been shown to be incorrect. We believe that a fundamental reason for the difficulty of these proofs is their style: they are based on detailed, not-very-systematic, reasoning about events and their ordering. Useful structure in such proofs is often provided by lemmas about partial orders of operations on objects, for example, Proposition 3 of [13] (for single-writer read/write objects) and Lemma 13.16 of [18] (for multi-writer read/write objects). These lemmas provide sufficient conditions for correctness of atomic read/write object implementations, based on a list of properties that a partial ordering of operations must satisfy. However, showing that these properties hold still requires detailed, ad hoc reasoning about events (see, e.g., [19, 20]).

In this paper, we initiate the study of systematic ways of verifying atomicity of distributed implementations, beginning with read/write objects (registers). Our general approach is to replace operational reasoning about events and partial orders with assertional reasoning about invariants and simulation relations. The assertional methods differ from the traditional operational arguments in two important ways. First, the system properties are stated precisely in terms of predicates over the system state components. Second, assertional proofs can be checked by examining individual state transitions of the algorithm without reasoning about entire executions. As such they lend themselves to mechanization, i.e., the process of checking a proof can be carried out using automated tools, such as theorem provers.

Our approach to carrying out assertional atomicity proofs is first to define an abstract state machine that captures the atomicity property and then, prove correctness of the object implementations by establishing a simulation mapping between the implementation and the specification automata. To this end, we define an abstract state machine, which we call the *Partial-Order Machine (PO-Machine)*, which records information about operations and their orders in its state. The PO-Machine expresses the common behavior of many existing atomic register implementations, in which client operation requests are gradually ordered relative to other operation requests until all the necessary ordering constraints are achieved. The ordering constructed is, in the limit, guaranteed to be a partial order of the requested operations that satisfies the conditions listed in Lemma 13.16 of [18]. In fact, the order constructed directly by the PO-Machine satifies slightly weaker constraints, but we prove that these are still sufficient to guarantee atomicity (see Section 2.2).

More specifically, the PO-Machine maintains a directed graph representing the ordering constraints among the submitted operations. As the PO-Machine executes, it adds new edges to the graph thus introducing new ordering constraints. The PO-Machine maintains a set of *ordered* operations whose prefix in the graph is fixed. It guarantees that all the ordered write operations are totally ordered, and each ordered read operation is immediately preceded by at most one ordered write operation. An ordered operation can later become *completed*. One subtle point about the

PO-Machine is that a read operation that is about to complete, must ensure that its immediately preceding write operation is also completed. This captures the essence of the *helping* mechanism found in many atomic register implementations. A completed operation is allowed to return a response. The response returned by a read operation is the value written by the last preceding (in the partial order) write operation, or the initial value if no such write exists.

We use the PO-Machine as a formal specification for distributed algorithms that implement atomic memory. We demonstrate the generality of this specification by showing that it is implemented by three different read/write register implementations: the message-passing register emulation of Attiya, Bar-Noy and Dolev (ABD) [4] (extended to handle multiple writers as in [20]) an optimized version of [4] that takes advantage of synchronized clocks at writers to eliminate the first phase of the write implementation (this optimization first appeared in [2]), and the unbounded version of the shared memory construction of a multi-writer/multi-reader register from single-writer/single-reader registers of [32]. We also discuss informally how to use the PO-Machine as a specification for a general (i.e., not necessarily read/write) atomic object implementation based on the replicated state machine protocol of Lamport [12].

We specify the PO-Machine and the algorithms formally using the I/O Automata (IOA)[17] and Timed IOA [9, 8] models, in fact, using formal specification languages that have been defined for these models. The IOA/TIOA specification languages lead to very stylized assertional proofs for invariants and simulation relations that can be partially automated using theorem provers. Moreover, the same IOA specifications can be used by the IOA compiler [27, 26] to produce executable Java code. We have used our automated translator to generate descriptions of PO-Machine and ABD in the language of the PVS theorem prover [23]. We used PVS to substantially increase the level of detail and assurance of some of our previous hand proofs. In fact, the use of PVS helped us to discover subtle bugs in our hand proofs. Automatic translation enabled us to easily tweak the simulation relations and rerun the proof scripts. We demonstrated the usefulness of the state machine style of specification by compiling and executing our IOA automata for ABD. Thus, a single formal representation of the algorithm can be used for specification, verification, and execution.

Other related work: Our use of a partial order automaton as an abstract specification was inspired by prior work of Fekete et al. on specifying the behavior of an Eventually Serializable Data Service [6]. Their specification used a (different) partial-order machine, which expresses weaker consistency requirements than atomicity. The algorithm studied in [6], based on an earlier algorithm of Liskov [10], was shown to achieve this weaker form of consistency.

2 Preliminary Definitions

2.1 The read/write service

A read/write object (a *register*) type consists of the following components: (1) an arbitrary set of values V with an initial value v_0, (2) the set of operations of the form $write(v)$, $v \in V$, and $read$, (3) the set of responses are ack and $v \in V$, and (4) the sequential specification f such that $f(w, write(v)) = (v, ack)$ and $f(w, read) = (w, w)$.

A read/write service implements a shared read/write register. To access the service, a client issues an *operation descriptor* consisting of a location identifier loc, and an operation identifier id. In addition, the write operation descriptor also contains a value val. We often refer to an oeprations descriptor x simply as *operation* x, and denote its various components by $x.loc$, $x.id$, and $x.val$. We denote by $\mathcal{O}_w$ and $\mathcal{O}_r$ the sets of the write and the read operations respectively, and by $\mathcal{O} = \mathcal{O}_w \cup \mathcal{O}_r$ the set of all operations. For a set $X \subseteq \mathcal{O}$, we denote by $X.id = \{x.id : x \in X\}$ the set of identifiers of operations in X.

Clients use the actions of the form $\mathsf{request}(x)$, $x \in \mathcal{O}$, and $\mathsf{response}(x, v)$, $x \in \mathcal{O}$, $v \in V \cup \{ack\}$, to issue operation requests and receive responses respectively. Given a sequence β of the $\mathsf{request}$ and $\mathsf{response}$ actions, an requested operation x is said to be complete in β if β contains $\mathsf{response}(x, v)$ for some $v \in V \cup \{ack\}$ which we call the *return* value of x.

We say that β is *well-formed* if there exists a function *cause* mapping the set of the response events to the set of the request events in β so that the following is satisfied: (1) For each response event $e = \mathsf{response}(x, *)$, $cause(e) = \mathsf{request}(x)$ (i.e., responses are not spuriously generated); and (2) *cause* is one-to-one (i.e., responses are not duplicated)[1].

The following definition will be used throughout the paper: Let Π be a set of read and write operations, and R be a binary relation over Π. For an operation $\pi \in \Pi$ we define $\textit{last-prec-writes}(\pi, R) = \{\omega \in \mathcal{O}_w : (\omega, \pi) \in R \wedge \nexists \omega' \in \mathcal{O}_w : (\omega, \omega') \in R \wedge (\omega', \pi) \in R\}$.

2.2 Atomicity

Atomicity (or linearizability) is specified as a property satisfied by the object implementation traces. It is typically defined in terms of the existence of *serialization points* for operations so that shrinking the operations to occur at their serialization points results in a valid sequential execution of the read/write register (see, e.g., Chapter 13 of [18], Chapter 9 of [5], or [7]). For our purposes in this paper, it is enough to give a sufficient condition for proving atomicity which as we prove in the full paper is equivalent to the one in Lemma 13.16 of [18].

Let β be a well-formed sequence of the actions of the read/write service interface that contains no incomplete operations, and Π be the set of operations requested in β. We say that β satisfies the *Weak Partial Order (Weak-PO)* property if there exists an irreflexive partial ordering $\prec$ of all the operations in Π, satisfying the conditions in Figure 1.

1. If the response event for π precedes the request event for ϕ in β, then $\phi \not\prec \pi$.
2. For any two write operations π and ϕ in Π, either $\pi \prec \phi$ or $\phi \prec \pi$.
3. If π is a write operation in Π and ϕ is a read operation in Π whose request event follows the response event for π, then $\pi \prec \phi$.
4. If π is a read operation in Π and ϕ is a read operation whose request event follows the response event for π, then for each $\omega \in \textit{last-prec-writes}(\pi, \prec)$, $\omega \prec \phi$.
5. Let π be a read operation in Π, and v be the value returned by π. If $\textit{last-prec-writes}(\pi, \prec) \neq \emptyset$, then $v = \omega.val$ for some $\omega \in \textit{last-prec-writes}(\pi, \prec)$. Otherwise, $v = v_0$.

Figure 1: The Weak Partial Order Constraints

Lemma 2.1 *If β is well-formed and satisfies Weak-PO, then β satisfies atomicity.*

3 The PO-Machine

In this section we define the Partial-Order Machine. First, we formally specify the environment assumptions of the read/write service. This environment is represented by a single automaton, called $Users$ (see Figure 2). The $Users$ automaton contains a single variable *requested* to keep track of the ids of requested operations, in order to avoid repeats. An implementation of the

[1]Note that our notion of well formedness is weaker than that found in the literature as it allows requests from the same location to be issued concurrently.

environment would not have such a variable, but would use some other mechanism to ensure unique operation ids (e.g., client id and a counter).

The PO-Machine (see Figure 3) maintains a partial order in its state, represented by variables *vertices* and *edges*. Vertices correspond to requested operations, and edges to ordering relationships that have been determined for these operations. When a request arrives, it is put into *vertices*; later, it becomes classified as *ordered*, then *completed*, and finally, *responded*. Edges may be added at any time from an ordered operation to an unordered one. An unordered operation π may become ordered at any time after it has acquired incoming edges from all operations that completed before π began; when a write operation π becomes ordered, new edges are inserted to ensure that π is ordered with respect to all previously-ordered write operations.

An ordered operation may becomes completed at any time; when a read operation completes, it also forces its immediately preceding write operations to complete. This captures the essence of the "helping" mechanism found in many atomic register implementations.

The partial ordering constructed by PO-Machine, which is a transitive closure of $(vertices, edges)$, is maintained in the derived variable *dag*. As PO-Machine executes, the *dag* keeps growing as new edges are being added. In Section A of the appendix, we prove that the limit of *dag* under successive inclusion satisfies Weak-PO. Since every trace of PO-Machine is obviously well-formed, by Lemma 2.1, PO-Machine implements an atomic register.

Signature:

Input:
$\mathsf{response}(x, v)$, $x \in \mathcal{O}$, $v \in V \cup \{ack\}$, $i \in \mathcal{I}$

Output:
$\mathsf{request}(x)$, $x \in \mathcal{O}$

State:

requested, a subset of $\mathcal{O}$, initially empty

Transitions:

```
Input response(x, v)
Effect:
    None

Output request(x)
Precondition:
    $x.id \notin requested.id$
Effect:
    $requested := requested \cup \{x\}$
```

Figure 2: The *Users* automaton

4 The Attiya, Bar-Noy and Dolev Algorithm

In this section, we present a distributed wait-free implementation of an atomic multi-writer/multi-reader register based on the well-known message-passing emulation of Attiya, Bar-Noy, and Dolev (ABD) [4]. We prove correctness of ABD by showing that ABD implements PO-Machine, which by Theorem A.3 implies that ABD implements an atomic register.

4.1 The ABD Algorithm

In the original ABD protocol, each process is responsible for handling client operation requests and maintaining the local copy of the register value. Here, we present a generalized version of the protocol where we let the two roles in the original protocol be performed by two classes of agents: *clients* and *replicas*. We also use a separate client to handle each user request thus allowing the actual clients to handle any number of requests and in whatever order. In addition, our implementation supports multiple writers using the technique of [20].

Signature:

Input:
  request(x), $x \in \mathcal{O}$

Output:
  response(x, v), $x \in \mathcal{O}$, $v \in V \cup \{ack\}$

Internal:
  add-edge(x, y), $x \in \mathcal{O}_w$, $y \in \mathcal{O}_w \cup \mathcal{O}_r$
  order(x), $x \in \mathcal{O}$
  complete(x), $x \in \mathcal{O}$

State:

$\mathit{vertices} \subseteq \mathcal{O}$, iniitally empty
$\mathit{ordered} \subseteq \mathcal{O}$, initially empty
$\mathit{completed} \subseteq \mathcal{O}$, initially empty
$\mathit{responded} \subseteq \mathcal{O}$, initially empty
$\mathit{edges} \subseteq \mathcal{O} \times \mathcal{O}$, initially empty
prec is a partial function from $\mathcal{O}$ to subsets of $\mathcal{O}$, initially empty

Derived vars:
dag, the transitive closure of $(\mathit{vertices}, \mathit{edges})$
For $x \in \mathcal{O}_r$, $\mathit{last\text{-}writes}(x) = \mathit{last\text{-}prec\text{-}writes}(x, \mathit{dag})$

Transitions:

```
Input request($x$)
Effect:
    $vertices := vertices \cup \{x\}$
    $prec(x) := completed \cap \mathcal{O}_w$

Internal add-edge(x, y)
Precondition:
    $y \in vertices - ordered$
    $x \in ordered$
Effect:
    $edges := edges \cup \{(x, y)\}$

Internal order(x), $x \in \mathcal{O}_w$
Precondition:
    $x \in vertices - ordered$
    $\forall y \in prec(x) : (y, x) \in dag$
Effect:
    $edges := edges \cup \{(x, y) : y \in ordered \cap \mathcal{O}_w \wedge$
                                    $(y, x) \notin dag\}$
    $ordered := ordered \cup \{x\}$

Internal order(x), $x \in \mathcal{O}_r$
Precondition:
    $x \in vertices - ordered$
    $\forall y \in prec(x) : (y, x) \in dag$
Effect:
    $ordered := ordered \cup \{x\}$

Internal complete(x)
Precondition:
    $x \in ordered - completed$
Effect:
    $completed := completed \cup \{x\}$
    if $x \in \mathcal{O}_r$ then
      $\forall y \in last\text{-}writes(x)$ do
        $completed := completed \cup \{y\}$

Output response($x, ack$), $x \in \mathcal{O}_w$
Precondition:
    $x \in completed - responded$
Effect:
    $responded := responded \cup \{x\}$

Output response($x, v_0$), $x \in \mathcal{O}_r$
Precondition:
    $x \in completed - responded$
    $last\text{-}writes(x) = \emptyset$
Effect:
    $responded := responded \cup \{x\}$

Output response($x, v$), $x \in \mathcal{O}_r$
Precondition:
    $x \in completed - responded$
    $last\text{-}writes(x) \neq \emptyset$
    $v = w.val : w \in last\text{-}writes(x)$
Effect:
    $responded := responded \cup \{x\}$
```

Figure 3: PO-Machine

Let P be a finite set of replicas. We define a *quorum system* $\mathcal{Q}$ on P to be the union of a set of *write quorums* $\mathcal{Q}_w$ and the set of *read quorums* $\mathcal{Q}_r$. $\mathcal{Q}_w$ and $\mathcal{Q}_r$ are sets of subsets of P such that for each $Q_w \in \mathcal{Q}_w$ and $Q_r \in \mathcal{Q}_r$, $Q_w \cap Q_r \neq \emptyset$. The ABD implementation is the composition of the Users automaton in Figure 2, the client automata for handling write requests C_x, for each $x \in \mathcal{O}_w$ (see Figure 6), the client automata for handling read requests C_y, for each $y \in \mathcal{O}_r$ (see Figure 5), and the replica automata R_p, for each $p \in P$ (see Figure 4. Clients interact with replicas using reliable point-to-point channels. We do not present the specification for the channel automata as their functionality is obvious.

The value stored at each replica is associated with a *tag*. Tags are chosen from the set $\mathcal{T} = \mathbb{N}^{\geq 0} \times \mathcal{O}.id$. For $tag \in \mathcal{T}$, we use the notation $tag.sn$ and $tag.id$ to refer to its various components. The tags are ordered lexicographically, with $(0, i_0)$ being the smallest tag.

The request handling at clients involves two rounds of quorum accesses, called the *read* phase and the *write* phase respectively, such that a read quorum is contacted during the read phase, and a write quorum is contacted during the write phase. A client keeps track of the request progress through the phases using the variable *status*.

More specifically, to handle a write request x, the client C_x (see Figure 6) performs a read phase to determine the highest tag t associated with the values stored at some read quorum. It then performs a write phase to store the value v associated with tag $(t.sn+1, x.id)$ at a write quorum. It then responds with *ack*. To handle a read request y, client C_y (see Figure 5) first performs a read phase to determine the value v associated with the highest tag t among those associated with the values stored at some read quorum. It then performs a write phase to guarantee that the pair (t, v) is stored at a write quorum. It then responds with v.

Signature:

Input:

$\mathsf{receive}(m)_{x,p}$, $x \in \mathcal{O}$, $m \in \{r, w\} \cup (\mathcal{T} \times V)$

Output:

$\mathsf{send}(m)_{p,x}$, $x \in \mathcal{O}$, $p \in R$, $m \in \{ack\} \cup (\mathcal{T} \times V)$

State:

$val \in V$, initially v_0

$tag \in \mathcal{T}$, initially $(0, i_0)$

For each $x \in \mathcal{O}$: $\textit{resp-buffer}_x \in seqof(\{ack\} \cup \mathcal{N}^{\geq 0} \cup (\mathcal{T} \times V))$, initially λ

Transitions:

```
Input $\mathsf{receive}(r)_{x,p}$
Effect:
    append $\langle val, tag \rangle$ to $\textit{resp-buffer}_x$

Input $\mathsf{receive}(w)_{x,p}$
Effect:
    append $\langle tag.sn \rangle$ to $\textit{resp-buffer}_i$

Input $\mathsf{receive}(t, v)_{x,p}$
Effect:
    if $t > tag$ then
        $tag := t$
        $val := v$
    append $\langle ack \rangle$ to $\textit{resp-buffer}_x$

Output $\mathsf{send}(m)_{p,x}$
Precondition:
    $\textit{resp-buffer}_x \neq \lambda$
    $m = head(\textit{resp-buffer}_x)$
Effect:
    delete head of $\textit{resp-buffer}_x$
```

Figure 4: Replica automaton R_p, $p \in P$

4.2 Correctness of ABD

We now prove that ABD implements an atomic register. Our strategy is to show that ABD implements PO-Machine by exhibiting a forward simulation relation from ABD to PO-Machine.

It is convenient for the correctness proof to define several derived state variables for the ABD automaton. These are summarized in Figure 8 (we use subscript x to refer to the state components of C_x). Among these variables, the most interesting one is *min-tag* that is used to keep track of the lowest possible tag that could ever be determined by a client at the end of the read phase.

Signature:

```
Input:
  request$(x)$
  receive$(m)_{p,x}$, $p \in P$, $m \in \{ack\} \cup (\mathcal{T} \times V)$

Output:
  response$(x, v)$,  $v \in V$
  send$(m)_{x,p}$,  $p \in P$, $m \in \{r\} \cup (\mathcal{T} \times V)$

Internal:
  rq-collected$(q)_x$,  $q \in \mathcal{Q}_r$
  wq-collected$(q)_x$,  $q \in \mathcal{Q}_w$
```

State:

$status \in Phase$, initially $idle$

$val \in V$, initially undefined

$tag \in \mathcal{T}$, initially $(0, i_0)$

$read\text{-}resp \in P$, initially empty

$write\text{-}resp \in P$, initilly empty

for each $p \in P$: $req\text{-}buffer_p \in seqof(\{r\} \cup (\mathcal{T} \times V))$, initially λ

Transitions:

```
Input request$(x)$
Effect:
    $status := p$
    for each $p \in P$:
        append $\langle r \rangle$ to $req\text{-}buffer_p$

Input receive$(v, t)_{p,x}$
Effect:
    $read\text{-}resp := read\text{-}resp \cup \{p\}$
    if $status = p \wedge t > tag$ then
            $val := v$
            $tag := t$

Internal rq-collected$(q)_x$
Precondition:
    $status = p$
    $read\text{-}resp \supseteq q$
Effect:
    $status := s$
    for each $p \in P$:
        append $\langle tag, val \rangle$ to $req\text{-}buffer_p$

Input receive$(ack)_{p,x}$
Effect:
    $write\text{-}resp := write\text{-}resp \cup \{p\}$

Internal wq-collected$(q)_x$
Precondition:
    $status = s$
    $write\text{-}resp \supseteq q$
Effect:
    $status := c$

Output response$(x, v)$
Precondition:
    $status = c$
    $val = v$
Effect:
    $status := r$

Output send$(m)_{x,p}$
Precondition:
    $req\text{-}buffer_p \neq \lambda$
    $m = head(req\text{-}buffer_p)$
Effect:
    delete head of $req\text{-}buffer_p$
```

Figure 5: Client automaton for read requests: C_x, $x \in \mathcal{O}_r$.

To facilitate the simulation proof we first state a number of invariants that express useful facts about the individual transitions and states of the ABD automaton (see Figure 10.6 in the appendix). For instance, the key property of *min-tag* used throughout the simulation proof is that it is not decreasing (see Figure 10.13 in the appendix). Another important invariant says that for each completed request x, there exists a write quorum whose members have tags which are at least as big as the final tag of x. For lack of space the proofs of the invariants are omitted from the extended abstract.

The simulation relation f from ABD to PO-Machine is shown in Figure 7. The following lemma

Signature:

Input:
  $\mathsf{request}(x)_i$
  $\mathsf{receive}(m)_{p,x}$, $p \in P$, $m \in \{ack\} \cup \mathcal{N}^{\geq 0}$

Output:
  $\mathsf{response}(x, v)$, $v \in \{ack\}$
  $\mathsf{send}(m)_{x,p}$, $m \in \{w\} \cup (\mathcal{T} \times V)$

Internal:
  $\mathsf{rq\text{-}collected}(q)_x$, $q \in \mathcal{Q}_r$
  $\mathsf{wq\text{-}collected}(q)_x$, $q \in \mathcal{Q}_w$

State:

$status \in Phase$, initially $idle$

$tag \in \mathcal{T}$, initially $(0, x.id)$

$read\text{-}resp \subseteq P$, initially empty

$write\text{-}resp \subseteq P$, initilly empty

for each $p \in P$: $req\text{-}buffer_p \in seqof(\{w\} \cup (\mathcal{T} \times V))$, initially λ

Transitions:

```
Input request(x)
Effect:
    $status := p$
    for each $p \in P$:
        append $\langle w \rangle$ to $req\text{-}buffer_p$

Input receive(sn)_{p,x}, $sn \in \mathcal{N}^{\geq 0}$
Effect:
    $read\text{-}resp := read\text{-}resp \cup \{p\}$
    if $status = p \ \wedge \ sn > tag.sn$ then
        $tag.sn := sn$

Internal rq-collected(q)_x
Precondition:
    $status = p$
    $read\text{-}resp \supseteq q$
Effect:
    $status := s$
    $tag.sn := tag.sn + 1$
    for each $p \in P$:
        append $\langle tag, x.val \rangle$ to $req\text{-}buffer_p$

Input receive(ack)_{p,x}
Effect:
    $write\text{-}resp := write\text{-}resp \cup \{p\}$

Internal wq-collected(q)_x
Precondition:
    $status = s$
    $write\text{-}resp \supseteq q$
Effect:
    $status := c$

Output response(x, ack)
Precondition:
    $status = c$

Effect:
    $status := r$

Output send(m)_{x,p}
Precondition:
    $m = head(req\text{-}buffer_p)$
Effect:
    delete head of $req\text{-}buffer_p$
```

Figure 6: Client automaton for write requests: C_x, $x \in \mathcal{O}_w$.

is proven in the full version of the paper:

Lemma 4.1 *f is a forward simulation from ABD to PO-Machine.*

5 Timed ABD

In this section, we present an optimized version of the ABD protocol that takes advantage of perfectly synchronized clocks at the writers to eliminate the read phase of the write implementation. This optimization is based on the ideas first introduced in [2].

f is the relation over $states(PO - Machine) \times states(ABD)$ such that each $(s, u) \in f$ iff:

- $u.requested = s.requested$
- $u.ordered = s.ordered$
- $u.vertices = s.pending$
- $u.responded = s.responded$
- $u.completed = s.completed \cup \bigcup_{r \in \mathcal{O}_r \cap s.completed} s.last\text{-}writes(r)$
- For all $x \in u.vertices$, if $y \in u.prec(x)$, then $s.tag_y \leq s.min\text{-}tag(x)$
- $u.dag \subseteq s.ordered \times s.ordered$
- For all $x, y \in \mathcal{O}_w \cap u.ordered$, if $(x, y) \in u.dag$, then $s.tag_x < s.tag_y$
- For all $x \in \mathcal{O}_w \cap u.ordered$ and $y \in \mathcal{O}_r \cap u.ordered$, $(x, y) \in u.edges$ iff $s.tag_x = s.tag_y$

Figure 7: Forward simulation from ABD to PO-Machine

- $pending = \{x \in \mathcal{O} : status_x \geq p\}$
- $completed = \{x \in \mathcal{O} : status_x \geq c\}$
- $ordered = \{x \in \mathcal{O} : status_x \geq s\}$
- $responded = \{x \in \mathcal{O} : status_x \geq r\}$
- For $r \in \mathcal{O}_r$: $last\text{-}writes(r) = \{w \in \mathcal{O}_w \cap ordered : s.tag_w = s.tag_r\}$
- For $x \in \mathcal{O}$, $p \in P$:

$$new\text{-}tag(x, p) = \begin{cases} t, \text{ if } \exists v \in V : \langle v, t \rangle \in resp\text{-}buffer_{p,x} \cup channel_{p,x} \\ (sn, x.id), \text{ if } \langle sn \rangle \in resp\text{-}buffer_{p,x} \cup channel_{p,x} \\ tag_p, \text{ otherwise} \end{cases}$$

- For $x \in \mathcal{O}$:

$$min\text{-}tag(x) = \begin{cases} \max[tag_x, \min_{Q \in \mathcal{Q}_r} \max\{new\text{-}tag(x, p) : p \in Q \setminus read\text{-}resp_x\}], \\ \qquad \text{if } \forall Q \in \mathcal{Q}_r,\ read\text{-}resp \subset Q \\ tag_x, \text{otherwise} \end{cases}$$

Figure 8: Derived variables for the ABD automaton

The optimized client algorithm works as follows: To write a value, the writer first takes its current clock reading, and then delays its execution until its clock exceeds the initial reading. The second clock reading is used as the tag with which the client performs the write phase. The code of the writer appears in Figure 11 of Section C in the appendix. The reader and the replica codes are the same as in ABD. In the full paper, we describe the optimized version of ABD and prove its correctness using the Timed I/O Automata (TIOA) model of [9, 8].

We prove that Timed-ABD implements an atomic register by exhibiting a forward simulation from Timed-ABD to PO-Machine. Since Timed-ABD and ABD automata are closely related, most of the ABD invariants carry over to Timed-ABD. The simulation relation along with several additional invariants involving the state components specific to Timed-ABDappears in Section C of the appendix.

6 The Vitanyi and Awerbuch Algorithm

In this section, we present a distributed wait-free implementation of an atomic multi-writer/multi-reader register based on the unbounded version of the well-known shared memory algorithm by Vitanyi and Awerbuch (VA) [32] , and argue that VA implements PO-Machine.

Our implementation of the VA algorithm is obtained through the following simple modification of the ABD automaton in Section 4. We re-use the ABD replica automata (see Figure 4) to implement the base single-writer/single-reader registers of the VA algorithm. (Note that although the replicas support more powerful read-modify-write objects, our implementation does not rely on this extra power.) We modify the ABD client to access replicas using fixed read and write quorums where the write quorums correspond to rows and the read qourums correspond to rows in the original VA algorithm. We also modify the Users automaton in a straightforward way (see Figure 2) to ensure that there is at most one outstanding request for each location (this models the more stringent well formedness assumption typically stipulated by shared memory algorithms).

For lack of space, we omit the detailed code of the VA automaton from the extended abstract. This can be found in the full version of the paper.

The correctness proof of VA is almost identical to that of ABD in Section 4.2. In particular, it is easy to see that the simulation from ABD to PO-Machine in Figure 7 is also a forward simulation from VA to PO-Machine. Alternatively, one could easily establish the VA correctness by showing that it implements ABD. In this case, the one-to-one mapping from the VA state components to the ABD state components is a forward simulation from VA to ABD. This shows that the VA and the ABD algorithms are closely related; namely, the former is a special case of the latter.

7 Replicated State Machine

In this section, we present an implementation of an arbitrary object using the Replicated State Machine methodology [12] and argue informally that our implementation simulates a modified version of the PO-Machine, called TO-Machine, that eventually totally orders all the requested operations.

Our RSM-based object implementation (RSM) is based on Lamport's original algorithm [12]. As in the other algorithms presented in this paper, there is a set of replicas P, each holding a copy of the object state. Each client is assigned to an arbitrary replica with which it communicates through a reliable (not necessary FIFO) channel. In addition, each replica can communicate with all the other replicas using pairwise point-to-point reliable FIFO channels.

The client's algorithm is simple: it immediately forwards each requested operation x to the replica it is assigned to. The (more interesting) replica algorithm is as follows: Each replica p maintains an integer timestamp ts, initially 0, and a message buffer M, initially empty. Whenever replica p receives an operation x from a client, it increments ts, forms a message $m = \langle p, ts, x \rangle$, inserts m into M, and then, broadcasts m to the other replicas. When p receives a message $m = \langle q, ts, y \rangle$, it inserts m into M, and sets ts to $\max(ts, m.ts) + 1$. At any point in the protocol, if replica p has a message $\langle r, ts, y \rangle \in M$ such that for all $m \in M$ where $m.sender \neq p$, $ts < m.ts \vee (ts = m.ts \wedge r < m.sender)$, then p applies y to p's local state copy, and removes m from M. Then, if the operation originated at a client assigned to p, p sends the return value back to this client.

One may prove correctness of RSM by showing that it implements the *Total-Order Machine (TO-Machine)* which is obtained by modifying the PO-Machine to enforce the total ordering for *all* operations. To see that RSM implements TO-Machine, note that in RSM, each operation received by a replica from a client follows the same sequence of stages as that undergone by each newly requested operation in TO-Machine. Namely, a replica p assigns each newly received operation x a timestamp, which determines the set of operations ordered *before* x (namely, the operations with a smaller timestamp). This behavior of RSM simulates a sequence of $\mathsf{add\text{-}edge}(*, x)$ actions of TO-Machine. Once the local timestamps of all the other replicas become larger than p's local timestamp, the set of the operations *after* x is determined, and therefore x becomes totally ordered. This simulates $\mathsf{order}(x)$. Whenever x becomes the smallest timestamped operation in p's message buffer, it is applied to p's local copy of the object state and removed from M. This sequence of actions corresponds to inserting x into *completed* in TO-Machine. Finally, x's response in RSM corresponds to x's response in TO-Machine.

8 Automated Tool Support

Using the IOA/TIOA formal languages for specification allows us to apply the tools that have recently been developed for the framework.
Theorem Proving: We have used the PVS theorem prover to check the proofs of some of the invariants and the simulation relation for ABD. IOA specifications of the PO-Machine and the composed ABD algorithm were translated to the language of PVS using the TAME libraries [3] and the TIOA to PVS translator. The simulation proof entails proving nine subgoals (corresponding to the conjuncts in Figure 7), for each action type of the ABD automaton. Some of the easy subgoals are discharged automatically by PVS's decision procedures; subgoals that have almost identical proofs can be proved quickly by reusing PVS proof scripts of previously proved subgoals, while harder subgoals require detailed user interaction in the form of application of the correct invariant lemmas, and suggesting the correct rewrite rules to the prover. While constructing the PVS proofs we found several minor gaps and a few bugs in the handwritten proofs. The PVS prover made it possible to quickly re-check proofs after modifying the specifications.
Code Generation: The target environment of the IOA compiler is a collection of workstations each running a Java JVM. The compiler only supports systems cast in "node-channel" form where a single automaton runs on each machine. In our ABD specification, each of an arbitrarily large set of requests corresponds to an individual automaton. To handle that set of request automata on a finite set of machines, we partition the requests among machines (*a priori*) and combine all the request automata in each partition into a single node automaton. Since IOA is an inherently nondeterministic language, in any state, many actions may be enabled. At runtime, Java must select enabled parameterized transitions to execute. So, we annotate ABD with a schedule block that yields an enabled parameterized transition as a function of the automaton state.

9 Future Work

Our work with four algorithms so far suggests to us that our PO-Machine (or small variants) may be general enough to capture many of the existing atomic register algorithms. We plan to use these methods to study a wider variety of algorithms, starting with bounded-timestamp-based constructions such as those in [30]; proofs for bounded-timestamp algorithms have been notoriously difficult and bug-prone. An interesting challenge will be to extend the framework to capture implementations that are not explicitly based on timestamps, for example, the construction that creates atomic bits from safe bits [28]). Another interesting direction deals with adapting the PO-Machine to capture weaker register semantics, such as safe registers, regular registers (including the multi-writer regular registers of Welch [25]), and sequentially consistent registers. There is an increased recent interest in these semantics as they capture the guarantees provided by many Byzantine-resilient storage systems [21, 22, 1] based on Byzantine quorums [21].

Finally, we are interested in identifying common patterns behind many diverse implementations of atomic objects. This will make it easier to understand and compare different algorithms. We expect that such patterns should be expressible in terms of common specification automata.

References

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: Optimal resilience with byzantine shared memory. In *Proceedings of the 23st ACM Symposium on Principles of Distributed Computing*

(PODC'04), pages 226–235, St John's Newfoundland, Canada, July 2004.

[2] Shlomi Dolev amd Seth Gilbert amd Nancy A. Lynch and Alex A. Shvartsmanand Jennifer L. Welch. GeoQuorums: Implementing atomic memory in ad hoc networks.

[3] Myla Archer. TAME: PVS Strategies for special purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1/4), February 2001.

[4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[5] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill Publishing Company, UK, 1998.

[6] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 300–309, Philadelphia, PA, May 1996.

[7] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[8] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed I/O automata. Technical Report MIT/LCS/TR-917a, MIT Laboratory for Computer Science, 2004. Available at `http://theory.lcs.mit.edu/tds/reflist.html`.

[9] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time system. In *RTSS 2003: The 24th IEEE International Real-Time Systems Symposium*, Cancun,Mexico, December 2003.

[10] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Science*, 10(4):360–391, 1992.

[11] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. Earlier version in Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] Leslie Lamport. On interprocess communication: Part I and II. *Dist. Comput.*, 1:77–101, 1986.

[14] Leslie Lamport. On interprocess communication, Part II: Algorithms. *Distributed Computing*, 1(2):86–101, April 1986.

[15] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):18–25, December 2001.

[16] Butler Lampson. The ABCD's of paxos. In *Proceedings of the Twentieth Annual ACM symposium on Principles of Distributed Computing*, Newport, RI, August 2001.

[17] N. A. Lynch and M.R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[18] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[19] Nancy Lynch and Alex Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 272–281, Seattle, Washington, USA, June 1997. IEEE.

[20] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In D. Malkhi, editor, *Distributed Computing (Proceedings of the 16th International Symposium on DIStributed Computing (DISC), Toulouse, France, October 2002)*, volume 2508 of *Lecture Notes in Computer Science*, pages 173–190. Springer-Verlag, 2002. Also, Technical Report MIT-LCS-TR-856.

[21] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Journal of Distributed Computing*, 11(4):203–213, 1998.

[22] J. P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *16th International Symposium on Distributed Computing (DISC'02), Toulouse, France*, Lecture Notes in Computer Science, pages 311–325. Springer-Verlag, 2002.

[23] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[24] Gary L. Peterson and James E. Burns. Concurrent reading while writing II: The multi-writer case. In *28th Annual Symposium on Foundations of Computer Science*, pages 383–392, Los Angeles, California, October 1987. IEEE.

[25] Cheng Shao, Evelyn Pierce, and Jennifer L. Welch. Multi-writer consistency conditions for shared memory objects. In *Proceedings of the 17th International Conference on Distributed Computing (DISC)*, pages 106–120, October 2003.

[26] Joshua A. Tauber. *Verifiable Compilation of I/O Automata without Global Synchronization.* PhD thesis, Massachusetts Institute of Technology, Cambridge,MA, September 2004.

[27] Joshua A. Tauber, Nancy A. Lynch, and Michael J. Tsai. Compiling IOA without global synchronization. In *Proceedings of the The 3rd IEEE International Symposium on Network Computing and Applications, (IEEE NCA04)*, pages 121–130, September 2004.

[28] John Tromp. How to construct an atomic variable. In *LNCS 392, Proc. 3rd International Workshop On Distributed Algorithms*, pages 292–302. Springer-Verlag, 1989.

[29] K. Vidyasankar. Concurrent reading while writing revisited. *Distributed Computing*, 4:81–85, 1990.

[30] Paul Vitanyi. Simple wait-free multireader registers. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC'02)*, volume 2508 of *Lecture Notes in Computer Science*, pages 118–132. Springer-Verlag, Berlin, 2002.

[31] Paul M. B. Vitányi. Distributed elections in an Archimedean ring of processors. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pages 542–547, Washington, D.C., April/May 1984.

[32] P.M.B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *27th IEEE Annual Symposium on Foundations of Computer Science*, pages 233–243, 1986.

A Correctness of PO-Machine

The following lemmas hold for every transition (s, π, s') of the PO-Machine:

1. $s.edges \subseteq s'.edges$ and $s.dag \subseteq s'.dag$.
2. If $x \in s.vertices$, then $s.prec(x) = s'.prec(x)$.
3. If $\pi = \mathsf{add\text{-}edge}(y, x)$, then $x \in s'.vertices - s'.ordered$, $s'.ordered = s.ordered$, and $(y, x) \in s'.edges$.
4. If $r \in \mathcal{O}_r \cap s.ordered$, then $s.last\text{-}writes(r) = s'.last\text{-}writes(r)$.
5. If $\pi = \mathsf{request}(x)$, $x \in \mathcal{O}_r$, then for each $y \in s.completed \cap \mathcal{O}_r$, if $\omega \in s.last\text{-}writes(y)$, then $\omega \in s'.prec(x)$.
6. Suppose that $\pi = \mathsf{response}(x, v)$ and $x \in \mathcal{O}_r$. If $s.last\text{-}writes(x) \neq \emptyset$, then $v = \omega.val$ for some $\omega \in s.last\text{-}writes(x)$. Otherwise, $v = v_0$.

The following properties hold in each reachable state s of the PO-Machine:

1. $s.completed \subseteq s.ordered \subseteq s.vertices \subseteq s.requested$.
2. $s.edges \subseteq s.ordered \times s.vertices$
3. For any request x, $s.prec(x) \subseteq s.completed$.
4. For any two write requests $x, y \in \mathcal{O}_w$ such that $x \neq y$, if $x, y \in s.ordered$, then either $(x, y) \in s.dag$ or $(y, x) \in s.dag$.
5. If $x \in s.vertices - s.ordered$, then there does not exists a request y such that $(x, y) \in s.edges$.
6. $s.dag$ is an irreflexive partial order.
7. If $y \in s.prec(x)$, then $(x, y) \notin s.dag$.
8. If $y \in s.prec(x)$ and $y \in \mathcal{O}_w$, then $(y, x) \in s.dag$.
9. **[Helping]** If $\pi \in s.completed$, then for all $\omega \in last\text{-}writes(\pi)$, $\omega \in s.completed$.

Figure 9: PO-invariants

We prove that every trace of the PO-Machine is well-formed and satisfies Weak-PO. Fix β to be a trace of PO-Machine, and Π be the set of all the operations requested in β.

Well-formedness is easy to see since in each state, the set of the responded operations is a subset of the set of the requested operations, and by precondition of $\mathsf{response}$ only the operations that have not yet responded are allowed to respond. Hence, the following holds:

Lemma A.1 *β is well-formed.*

We now show how to order the operations in Π to obtain a partial ordering satisfying the Weak-PO property in Figure 1.

Lemma A.2 *If β is well-formed and does not contain incomplete operations, then β satisfies Weak-PO.*

The above lemmas and Lemma 2.1 imply the following

Theorem A.3 *Each trace of the PO-Machine satisfies atomicity.*

The detailed proof can be found in the full version of the paper.

B Correctness of ABD

The transition and state invariants satisfied by ABD appear in Figure 10.

Transition invariants: Let (s, π, s') be a transition of ABD. The following properties hold:

1. For each $p \in P$, tag_p is non-decreasing (i.e., $s.tag_p \leq s'.tag_p$).
2. For each $x \in \mathcal{O}$, $p \in P$, $new\text{-}tag(x,p)$ is non-decreasing (i.e., $s.new\text{-}tag(x,p) \leq s'.new\text{-}tag(x,p)$).
3. For each $x \in \mathcal{O}$, tag_x is non-decreasing (i.e., $s.tag_x \leq s'.tag_x$).
4. If $x \in s.ordered$, then $s.tag_x = s'.tag_x$.
5. For each reachable state s of ABD and $x \in \mathcal{O}$ such that $\forall Q \in \mathcal{Q}_r$, $s.read\text{-}resp_x \subset Q$: if $\exists Q \in \mathcal{Q}_r$ such that $p \in Q - s.read\text{-}resp_x$, then $s.min\text{-}tag(x) \leq \max(s.tag_x, s.new\text{-}tag(x,p))$.
6. For each $x \in \mathcal{O}$, $min\text{-}tag(x)$ is non-decreasing.
7. if $r \in \mathcal{O}_r \cap s.ordered$, then $s.last\text{-}writes(r) = s'.last\text{-}writes(r)$.

State Invariants: Let s be a reachable state of ABD:

8. $s.responded \subseteq s.completed \subseteq s.ordered \subseteq s.pending \subseteq s.requested$.
9. $x \in \mathcal{O}$, $x \notin s.requested$ **iff** C_x **is in its initial state in** s.
10. **For each reachable state s of ABD,** $x \in \mathcal{O}$, $p \in P$, **if** $\langle tag, val \rangle$ **is in** $s.req\text{-}buffer_{p,x} \cup s.channel_{x,p}$, **then** $x \in s.ordered$, $s.tag_x = tag$, **and** $s.val_x = val$.
11. $x \in \mathcal{O}$, $p \in P$ **if** $\langle ack \rangle$ **is in** $s.channel_{p,x} \cup s.resp\text{-}buffer_{p,x}$, **then** $s.tag_p \geq s.tag_x$.
12. $x \in \mathcal{O}$, $p \in P$, **if** $p \in s.write\text{-}resp_x$, **then** $s.tag_p \geq s.tag_x$.
13. **For** $x \in \mathcal{O}$, **if** $x \in s.completed$, **then there exists** $Q \in \mathcal{Q}_w$ **such that** $\forall p \in Q$, $s.tag_p \geq s.tag_x$.
14. **For** $x, y \in s.ordered \cap \mathcal{O}_w$, **if** $x \neq y$, **then** $s.tag_x \neq s.tag_y$.
15. **For** $x \in \mathcal{O}_w$, **if** $x \in s.ordered$, **then** $s.tag_x > (0, i_0)$.
16. **For** $x \in \mathcal{O}_r$, **if** $s.tag_x = (0, i_0)$, **then** $s.val_x = v_0$.
17. **For** $x \in \mathcal{O}_r$, **if** $s.tag_x \neq (0, i_0)$, **then there exists** $y \in s.ordered \cap \mathcal{O}_w$ **such that** $s.tag_y = s.tag_x$ **and** $y.val = s.val_x$.

Figure 10: ABD Invariants

C The Timed ABD Implementation

C.1 Invariants

The following invariant says that the local clocks of the writers are perfectly synchronized.

Lemma C.1 *For each reachable state s of Timed-ABD, and $x, y \in \mathcal{O}_w$, $s.clock_x = s.clock_y$.*

The next lemma is a "timed" variant of Invariant 13 in Figure 10 for write requests:

Lemma C.2 *For each reachable state s of Timed-ABD, if $x \in \mathcal{O}_w$, then for all $y \in \mathcal{O}_w$, $s.tag_x.sn \leq s.clock_y$.*

C.2 The simulation

The simulation relation depicted in Figure 12 is a forward simulation from Timed-ABD to Timed-PO. The simulation relation is the same as that used in the simulation proof of ABD with two additional conditions involving the *clock* and *req-time* variables.

Signature:

Input:
  $\mathsf{request}(x)_i$
  $\mathsf{receive}(m)_{p,x}$, $p \in P$, $m \in \{ack\} \cup \mathcal{N}^{\geq 0}$

Internal:
  order_x

Output:
  $\mathsf{response}(x, v)$, $v \in \{ack\}$
  $\mathsf{send}(m)_{x,p}$, $m \in \{w\} \cup (\mathcal{T} \times V)$

Internal:
  $\mathit{wq\text{-}collected}(q)_x$, $q \in \mathcal{Q}_w$

State:

$\mathit{clock} \in \mathbb{R}$, initially 0

Discrete $\mathit{req\text{-}time} \in \mathbb{R}$, initially 0

$\mathit{status} \in \mathit{Phase}$, initially idle

$\mathit{tag} \in \mathcal{T}$, initially $(0, x.id)$

$\mathit{write\text{-}resp} \subseteq P$, initilly empty

for each $p \in P$: $\mathit{req\text{-}buffer}_p \in \mathit{seqof}(\{w\} \cup (\mathcal{T} \times V))$, initially λ

Transitions:

```
Input request(x)
Effect:
    $status := p$
    $req\text{-}time := clock$

Internal order_x
Precondition:
    $clock > req\text{-}time$
    $status = p$
Effect:
    $tag.sn := clock$
    $status := s$
    for each $p \in P$:
        append $\langle tag, x.val \rangle$ to $req\text{-}buffer_p$
```

Trajectories:

evolve
  $d(\mathit{clock}) = 1$
All the other state variables are kept unchanged

```
Input receive(ack)_{p,x}
Effect:
    $write\text{-}resp := write\text{-}resp \cup \{p\}$

Internal wq-collected(q)_x
Precondition:
    $status = s$
    $write\text{-}resp \supseteq q$
Effect:
    $status := c$

Output response(x, ack)
Precondition:
    $status = c$
Effect:
    $status := r$

Output send(m)_{x,p}
Precondition:
    $m = head(req\text{-}buffer_p)$
Effect:
    delete head of $req\text{-}buffer_p$
```

Figure 11: Optimized client automaton for write requests: $\bar{C}_x$, $x \in \mathcal{O}_w$.

Lemma C.3 *The relation f in Figure 12 is a forward simulation from Timed-ABD to Timed-PO.*

f is the relation over $states(Timed-PO) \times states(Timed-ABD)$ such that $(s,u) \in f$ iff:

- $u.requested = s.requested$
- $u.vertices = s.pending$
- $u.ordered = s.ordered$
- $u.completed = s.completed \cup \bigcup_{r \in \mathcal{O}_r \cap s.completed} s.last\text{-}writes(r)$
- $u.responded = s.responded$
- For all $x \in u.vertices \cap \mathcal{O}_r$, if $y \in u.prec(x)$, then $s.tag_y \leq s.min\text{-}tag(x)$
- For all $x \in u.vertices \cap \mathcal{O}_w$, if $y \in u.prec(x)$, then $s.tag_y.sn \leq s.req\text{-}time_x$.
- For all $x \in (u.vertices - u.ordered) \cap \mathcal{O}_w$, $y \in u.ordered \cap \mathcal{O}_w$, if $s.tag_y.sn < s.clock_x$, then $(y,x) \in u.edges$.
- $u.dag \subseteq s.ordered \times s.ordered$
- For all $x, y \in \mathcal{O}_w \cap u.ordered$, if $(x,y) \in u.dag$, then $s.tag_x < s.tag_y$
- For all $x \in \mathcal{O}_w \cap u.ordered$ and $y \in \mathcal{O}_r \cap u.ordered$, $(x,y) \in u.edges$ iff $s.tag_x = s.tag_y$

Figure 12: Forward simulation from Timed-ABD to Timed-PO