Proving Atomicity: An Assertional Approach

Gregory Chockler[1], Nancy Lynch[1], Sayan Mitra[1], and Joshua Tauber[1]

MIT CSAIL, The Stata Center, Bldg.32, 32 Vassar Street, Cambridge MA 02139, USA
{grishac,lynch,mitras,josh}@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. In addition, we show that a simplified version of our specification is implemented by a general atomic object construction based on the Lamport's replicated state machine algorithm.

1 Introduction

Many distributed and network-based services can be modeled as shared objects accessible to (possibly remote) clients through well-defined interfaces. Atomicity [16, 21] (also known as *linearizability* [10]) is a desirable property for such objects as it allows clients using the objects to perceive the operations that occur in each run as occurring atomically, in some sequential order. This perception makes it easier to understand the behavior of a system using distributed services, and so, simplifies the task of system design.

Atomic services could be implemented simply on single server machines. However, to achieve high availability in a distributed system and to tolerate failures, atomic services are typically implemented by distributed algorithms. Many distributed algorithms have been proposed for implementing atomic objects; see,

⋆ This work is supported by MURI–AFOSR SA2796PO 1-0000243658, USAF–AFRL #FA9550-04-1-0121, NSF Grant CCR-0121277, NSF-Texas Engineering Experiment Station Grant 64961-CS, and DARPA F33615-01-C-1896.

for example, [17, 15, 36, 27, 33, 32, 10, 35, 14, 19, 18, 22, 23, 8]. These use a range of techniques to achieve the appearance of total ordering, for example, assigning timestamps and processing operations in timestamp order, or using quorum configurations.

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 shared read/write memory objects 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 [16] (for single-writer read/write objects) and Lemma 13.16 of [21] (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., [22, 23]).

In this paper, we study systematic ways of verifying distributed implementations of atomic objects, 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 interactive 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. The challenge is to find a specification automaton that is general enough to apply to many existing implementations, and at the same time sufficiently close to the actual implementations to simplify the task of finding the mapping. One example of an atomicity specification that turned out to be too abstract for carrying out simulation proofs is the canonical atomic object automaton of Section 13.1.2 of [21]. The canonical object automaton maintains a buffer used to store incoming client requests. Buffered requests can later be applied to the object state, and the generated responses are returned to their originators. Unfortunately, this specification, though simple, does not provide sufficient detail to allow for easy match with concrete implementations.

We therefore, give more detailed specifications. Namely, 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 sufficient conditions for showing atomicity.

We use the PO-Machine as a formal specification for distributed algorithms that implement atomic memory. We show that it is implemented by three different read/write register constructions: the message-passing emulation of Attiya, Bar-Noy, and Dolev (ABD) [3] (extended to handle multiple writers as in [23]), an optimized version of ABD that takes advantage of synchronized clocks at writers [8], and the unbounded version of the shared memory construction of a multi-writer/multi-reader register from single-writer/single-reader registers of [36]. We also show that a slight modification of the PO-Machine, called the *TO-Machine*, can be used to prove atomicity of a general (i.e., not necessarily read/write) object implementation based on the replicated state machine protocol of Lamport [15].

We specify the PO-Machine and the algorithms formally using the I/O Automata (IOA)[20] and Timed IOA [12, 11] 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 [31, 30] to produce executable Java code.

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 [9]. Their specification used a (different) partial-order machine, which expresses weaker consistency requirements than atomicity. The algorithm studied in [9], based on an earlier algorithm of Liskov et al. [13], was shown to achieve this weaker form of consistency.

The only other published simulation-based atomicity proofs we are aware of are those of Bogdanov [5] (replicated state machine), and Doherty et al. (lock-free queue) [7]. The proofs in both these papers are complicated: They involve multiple levels of asbtraction as well as both forward and backward simulations. In contrast, every construction considered in this paper is shown to be atomic by exhibiting a single forward simulation directly from the implementation automaton to a specification automaton.

Another example of using assertional reasoning for proving atomicity is the work by Wang and Stoller [37], which uses static analysis combined with model checking to verify atomicity of code blocks involving lock-free synchronization primitives. A more general discussion of assertional proof techniques can be found in [28].

The rest of the paper is organized as follows: In Section 2, we introduce preliminary definitions and notation used throughout the paper. The sufficient condition for proving atomicity is specified in Section 3. The PO-Machine is described in Section 4. The ABD algorithm is presented and proved correct in Section 5. A time-based version of ABD is discussed in Section 6. Section 7 briefly discusses the proofs of the Vitanyi-Awerbuch's register construction, and of the Lamport's replicated state machine. Section 8 discusses future directions. For lack of space, we only outline intuition and highlight basic ideas underlying the correctness proofs. The detailed proofs can be found in the full version of the paper [6].

2 Preliminary Definitions

We use the I/O Automata (IOA)[20] model to formally specify services, describe algorithms and carry out proofs. An I/O automaton is a non-determenistic state machine whose state can change atomically through a discrete transition labeled by a discrete *action*. The set of the automaton's actions is called the *action signature* of the automaton. The actions can be either *external* or *internal*. The external actions, which can be either *input* or *output*, model interaction with the automaton's environment; and the internal actions model local computation steps. In Section 6, we also use the Timed I/O Automata (TIOA) model [12, 11], which, in addition to discrete transitions, also allows the automata state to evolve by *trajectories*, which describe evolution of the state over time.

We use *forward simulations* to carry out atomicity proofs. Informally, a forward simulation is a relationship between the states of two automata requiring that the transitions of one system can in some sense be mimicked by the other. A precise definition of the simulation formalism can be found in [21].

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 operations 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 each response event to a preceding request event 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 $\textit{last-prec-writes}(\pi, R) = \{\omega \in \mathcal{O}_w : (\omega, \pi) \in R \wedge \nexists \omega' \in \mathcal{O}_w : (\omega, \omega') \in R \wedge (\omega', \pi) \in R\}$.

3 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 [21], Chapter 9 of [4], or [10]). For our purposes in this paper, it is enough to give a sufficient condition for proving atomicity; this condition is equivalent to the one in Lemma 13.16 of [21].

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 *Partial Order* property (henceforth, referred to as *PO*) if there exists an irreflexive partial ordering $\prec$ of all the operations in Π, satisfying the following:

Property 1 (PO Constraints)

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, $v = v_0$.*

The following lemma is proved in [6]:

Lemma 1. *β satisfies PO iff there exists an irreflexive partial ordering of all the operations in Π, satisfying the (more restrictive) constraints of Lemma 13.16 of [21].*

From the above result and Lemma 13.16 of [21], we obtain:

Lemma 2. *If β is well-formed and satisfies PO, then β satisfies atomicity.*

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

4 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 $Users$, whose code could be found in [6]. 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 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).

Lemma 3. *For $x, y \in requested$, $x = y \Leftrightarrow x.id = y.id$.*

The PO-Machine signature and state variables appear in Figure 1, and its transitions appear in Figure 2. This automaton 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 ordered write operations to unordered ones (see action add-edge).

An unordered operation π may become ordered at any time after it has acquired incoming edges from all write operations that completed before π began (i.e., all writes in $prec(\pi)$). This ensures that constraints 1 and 3 of Property 1 hold among all writes, and between writes and reads. Constraint 1 is also trivially preserved among reads as edges originating at read requests are disallowed by the PO signature (see Figure 1). 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 (see action order) so that constraint 2 of Property 1 is satisfied.

An ordered operation may become completed at any time; when a read operation ϕ completes, it also forces each write operation π immediately preceding ϕ in the partial order to complete. This ensures that every read operation invoked after ϕ completes will find π in its *prec* set, and will therefore, become ordered only after it has an incoming edge from π. This guarantees that constraint 4 of Property 1 is satisfied, and also 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 (see action response). Thus, constraint 5 of Property 1 is satisfied.

In [6], we prove that the limit of the transitive closure of $(vertices, edges)$, maintained in the derived variable *dag*, satisfies Property 1. Since every trace of PO-Machine is obviously well-formed, by Lemma 2, PO-Machine implements an atomic register:

Theorem 1. *Each trace of the PO-Machine satisfies atomicity.*

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:

vertices $\subseteq \mathcal{O}$, initially empty
ordered $\subseteq \mathcal{O}$, initially empty
completed $\subseteq \mathcal{O}$, initially empty
responded $\subseteq \mathcal{O}$, initially empty
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 ($vertices$, $edges$)

For $x \in \mathcal{O}_r$, $last\text{-}writes(x) = last\text{-}prec\text{-}writes(x, dag)$

Fig. 1. PO-Machine signature and states

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:
  $responded := responded \cup \{x\}$

Fig. 2. PO-Machine transitions

5 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 algorithm of Attiya, Bar-Noy, and Dolev [3] (which we call ABD). We prove correctness of ABD by showing that ABD implements PO-Machine, which by Theorem 1, implies that ABD implements an atomic register.

The original ABD protocol implements a wait-free atomic read/write register using a collection of n processes communicating among themselves through reliable point-to-point channels. The implementation is resilient to up to $n/2$ process crashes. Each process in ABD is responsible for both: handling the client operation requests, and storing and updating the local copy of the register value.

Here, we present a generalized version of ABD where we let the two roles in the ABD protocol be performed by two classes of agents: *clients* and *replicas*. This design allows for flexibility in assigning roles to actual network locations thus simplifying the algorithm deployment in real systems. We also use a separate client to handle each user request so that the actual clients can handle any number of requests and in whatever order (for example, requests can be partitioned among several threads, or executed sequentially). Our implementation also supports multiple writers using the technique of [23].

We now describe the ABD implementation (the ABD automaton) in more detail. 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 automaton is the composition of the Users automaton of Section 4, the client automata C_x, $x \in \mathcal{O}$, the replica automata R_p, $p \in P$, and the reliable point-to-point channel automata connecting each client C_x with replica R_p and vice versa. The client's interface and state variables appear in Figure 3. The code of the reader client, the writer client and the replica appear in Figures 4, 5, and 6 respectively. 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 two-field records consisting of a sequence number sn, which is a non-negative integer, and a request identifier id. Tags are ordered lexicographically with the precendence to the sequence number field.

Clients access read (resp. write) quorums by first sending a message to all the replicas, and then awaiting responses from a write (resp. a read) quorum. 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*. The operation's status is initially *idle*. It is changed to *pending (*p*)* at the beginning of the read phase. It becomes *sending (*s*)* at the beginning of the write phase. It is changed to *committed (*c*)* upon completion of the write phase, and finally to *responded (*r*)* after a response is returned.

Specifically, to handle a write request x, the client C_x (see Figure 5) 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 $(t.sn, x.id)$ at a write quorum. It then responds with *ack*. To handle a read request y, client C_y (see Figure 4) 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.

The replica's algorithm (see Figure 6) is simple: In response to a read phase message, a replica p either responds with its current tag (for write requests), or the current tag and the value (for read requests). In response to a write phase message carrying a tag which is bigger than p's current tag, p overwrites its current tag and the value with those in the message. Otherwise, the p's state is left unchanged. In both cases, p responds with ack.

Types:

$Tag = \mathcal{N}^{\geq 0} \times \mathcal{O}.id$, with selectors sn and id, ordered lexicographically

$Phase = \{\mathsf{idle}, \mathsf{p}, \mathsf{s}, \mathsf{c}, \mathsf{r}\}$, ordered so that $\mathsf{idle} < \mathsf{p} < \mathsf{s} < \mathsf{c} < \mathsf{r}$

Signature:

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

Output:
- $\mathsf{response}(x, v)$, $v \in V \cup \{ack\}$
- $\mathsf{send}(m)_{x,p}$, $p \in P$, $m \in \{r, w\} \cup (Tag \times V)$

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
$val \in V$, initially undefined
$tag \in Tag$, initially $(0, i_0)$
$read\text{-}resp \in P$, initially empty
$write\text{-}resp \in P$, initilly empty
for each $p \in P$: $req\text{-}buffer_p \in seqof(\{r, w\} \cup (Tag \times V))$, initially λ

Fig. 3. The state and signature of client automata C_x, $x \in \mathcal{O}$ for ABD.

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

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

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

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

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

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

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

Fig. 4. Transitions of reader C_x, $x \in \mathcal{O}_r$ for ABD.

Correctness of ABD: We now prove that ABD implements an atomic register. Our strategy will be to show that ABD implements PO-Machine by exhibiting a forward simulation from ABD to PO-Machine. In the following, for each $x \in \mathcal{O}$, we will use subscript x to refer to the state variables of C_x. It is convenient for the ABD correctness proof to define several derived variables for the ABD automaton. These are summarized in Figure 7.

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

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

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

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

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

Output $\mathsf{response}(x, ack)$
Precondition:
  $status = \mathsf{c}$

Effect:
  $status := \mathsf{r}$

Output $\mathsf{send}(m)_{x,p}$
Precondition:
  $m = head(\mathit{req\text{-}buffer}_p)$
Effect:
  delete head of $\mathit{req\text{-}buffer}_p$

Fig. 5. Transitions of writer C_x, $x \in \mathcal{O}_w$ for ABD.

Signature:

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

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

State:

$val \in V$, initially v_0

$tag \in Tag$, initially $(0, i_0)$

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

Transitions:

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

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

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

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

Fig. 6. Replica automaton R_p, $p \in P$ for ABD

Among these variables, the most interesting one is *min-tag* which 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. At the beginning and before any replica has responded, *min-tag* is the smallest tag among the maximum tags carried by replicas in every read quorum. As the client is progressing through the read phase it might get a response from a replica whose tag is bigger than the current value of *min-tag*. In this case, the definition of *min-tag* ensures that *min-tag* is assigned to that higher value. Finally, upon completion of the read phase, the value of *min-tag* is fixed to be the maximum tag received during the phase. The simulation proof relies on the following key property of *min-tag*:

Lemma 4. *For each $x \in \mathcal{O}$, min-tag(x) is non-decreasing.*

The simulation mapping from the states of ABD to the states of the PO-Machine appears in Figure 8. The first four components of the mapping are

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

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

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

$$\textit{min-tag}(x) = \begin{cases} \max[tag_x, \min_{Q \in \mathcal{Q}_r} \max\{\textit{new-tag}(x, p) : p \in Q \setminus \textit{read-resp}_x\}], \\ \qquad\qquad \text{if } \forall Q \in \mathcal{Q}_r, \textit{read-resp} \not\supseteq Q \\ tag_x, \text{otherwise} \end{cases}$$

Fig. 7. Derived variables for the ABD automaton

straightforward: All the operations that have ever been requested (indicated by status $>$ idle) are mapped to *vertices*; the operations that have completed the read phase and acquired final tags (indicated by status $>$ p) are mapped to *ordered*; and the operations that have responded (indicated by status $>$ c) are mapped to *responded*.

The set of edges consists only of edges among operations that have completed their read phases (8.7). The edges among these operations are determined by their tag order and type. Specifically, any two writes x and y, such that $tag_x < tag_y$, are connected by edge (x, y) (8.8); and each read x and write y such that $tag_x = tag_y$, are connected through edge (y, x) (8.9). To maintain the mapping for *edges*, each $\mathsf{rq\text{-}collected}(x)$ for $x \in \mathcal{O}_w$ is simulated by a sequence of $\mathsf{add\text{-}edge}(y, x)$ for each ordered write operation y such that $tag_y \leq tag_x$, followed by $\mathsf{order}(x)$; and each $\mathsf{rq\text{-}collected}(x)$ for $x \in \mathcal{O}_w$ is simulated by a sequence of $\mathsf{add\text{-}edge}(y, x)$ for each ordered operation y such that $tag_y = tag_x$. No actions involving unordered operations (i.e., the operations with status $<$ s) result in adding new edges.

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

1. $u.requested = s.requested$
2. $u.vertices = s.pending$
3. $u.ordered = s.ordered$
4. $u.completed = s.completed \cup \bigcup_{r \in \mathcal{O}_r \cap s.completed} s.\textit{last-writes}(r)$
5. $u.responded = s.responded$
6. For all $x \in u.vertices$, if $y \in u.prec(x)$, then $s.tag_y \leq s.\textit{min-tag}(x)$
7. $u.dag \subseteq s.ordered \times s.ordered$
8. For all $x, y \in \mathcal{O}_w \cap u.ordered$, if $(x, y) \in u.dag$, then $s.tag_x < s.tag_y$
9. 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 $s.tag_x = s.tag_y$

Fig. 8. Forward simulation from ABD to PO-Machine

The most interesting part of the proof is to show that $\mathsf{order}(x)$ becomes enabled once all the (y, x) edges have been added. For that we need to show that the tag acquired by x at the end of the read phase is at least as big as the tag of every operation that had completed before x began. Since at the end of the read phase, $tag_x = \mathit{min\text{-}tag}(x)$, the necessary enabling condition is provided by part 8.6 of the mapping that requires that for each $y \in \mathit{prec}(x)$, $tag_y \leq \mathit{min\text{-}tag}(x)$.

To show that 8.6 is maintained throughout the read phase of x, $\mathsf{request}(x)$ is simulated by the $\mathsf{request}(x)$ action of the PO-Machine; and each $\mathsf{receive}$ is simulated by the empty sequence. Since at the time x is invoked, the tag of every $y \in \mathit{prec}(x)$ has been stored at a write quorum of replicas, and because every pair of write and read quorums intersects, $\min_{Q \in Q_r} \max_{p \in Q}\{tag_p\} \geq tag_y$. Hence, 8.6 is preserved by $\mathsf{request}(x)$. Finally, since $\mathit{min\text{-}tag}(x)$ is non-decreasing (Lemma 4) and $\mathit{prec}(x)$ is not affected by any action except $\mathsf{request}$, 8.6 is preserved by $\mathsf{receive}$. Hence, by the end of the read phase of x, for each $y \in \mathit{prec}(x)$, $tag_y \leq \mathit{min\text{-}tag}(x)$ as required.

We argued informally that the mapping in Figure 8 is a forward simulation from ABD to the PO-Machine. A detailed proof appears in [6].

Lemma 5. *The mapping in Figure 8 is a forward simulation from ABD to the PO-Machine.*

Since by Theorem 1, each trace of the PO-Machine satisfies atomicity, the same is true for every trace of ABD:

Theorem 2. *Each trace of ABD satisfies atomicity.*

Automated Tools Support: We have used the TIOA to PVS translator and TAME library [2] to generate descriptions of the PO-Machine and the ABD algorithm in the language of the Prototype Verification System (PVS) [26]. We used PVS to substantially increase the level of detail and assurance of some of our previous hand proofs. In fact, we discovered several gaps and bugs in our hand proofs. Automatic translation enabled us to easily tweak the simulation relations and rerun the proof scripts. We also used the IOA code generator tool [31, 30] to compile the verified ABD automaton into an executable Java code. This way, a single formal representation of the ABD algorithm was used for specification, verification, and execution.

6 Timed ABD

In this section, we present an optimized version of the ABD protocol, called *Timed-ABD*, that takes advantage of perfectly synchronized clocks at the writers to eliminate the read phase of the write implementation (see [8]).

The Timed-ABD is the composition of the following timed automata: the replica and reader client automata in Figures 6 and 4 respectively augmented with arbitrary trajectories that keep their state unchanged; and the writer client automata whose code appears in Figure 9. To model synchronized clocks, each writer maintains a local variable *clock* whose trajectory is $d(\mathit{clock}) = 1$ (i.e., the clock value grows continuously, at the same rate as the real time).

The writer algorithm is as follows: To write a value, the writer first takes its current clock reading, and then delays its execution until its clock exceeds the

Signature:

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

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

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

State:
$clock \in \mathcal{R}$, initially 0
Discrete $\textit{req-time} \in \mathcal{R}$, initially 0
$status \in Phase$, initially $idle$
$tag \in Tag$, initially $(0, x.id)$
$\textit{write-resp} \subseteq P$, initilly empty
for each $p \in P$: $\textit{req-buffer}_p \in seqof(\{w\} \cup (Tag \times V))$, initially λ

Transitions:

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

Internal order$_x$
Precondition:
$clock > \textit{req-time}$
$status =$ p
Effect:
$tag.sn := clock$
$status :=$ s
for each $p \in P$:
append $\langle tag, x.val \rangle$ to $\textit{req-buffer}_p$

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

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

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

Output send$(m)_{x,p}$
Precondition:
$m = head(\textit{req-buffer}_p)$
Effect:
delete head of $\textit{req-buffer}_p$

Trajectories:

evolve

$d(clock) = 1$

All the other state variables are kept unchanged

Fig. 9. Writer client C_x, $x \in \mathcal{O}_w$ for Timed-ABD

initial reading. The second clock reading is used as the tag with which the client performs the write phase.

The simulation mapping from the states of Timed-ABD to the states of Timed-PO (i.e., the PO-Machine augmented with arbitrary trajectories that do not change its state) appears in Figure 10. To see that the mapping is preserved, we observe that a write operation becomes ordered once it is verified that a non-zero amount of time has elapsed since it was requested. We therefore, simulate each Timed-ABD trajectory corresponding to a non-zero time interval by a trajectory of Timed-PO of the same length, followed by a sequence of add-edge actions, followed by order. The rest of the simulation proof is straightforward (see [6] for details).

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

1-5: Identical to 1-5 in Figure 8
6: For all $x \in u.vertices \cap \mathcal{O}_r$, if $y \in u.prec(x)$, then $s.tag_y \leq s.\textit{min-tag}(x)$
7-8: Identical to 7-8 in Figure 8
9: For all $x \in u.vertices \cap \mathcal{O}_w$, if $y \in u.prec(x)$, then $s.tag_y.sn \leq s.\textit{req-time}_x$
10: 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 $(y, x) \in u.edges$.

Fig. 10. Forward simulation from Timed-ABD to Timed-PO

7 Other Algorithms

We discuss briefly how to prove atomicity of the unbounded multi-writer/multi-reader register construction of Vitanyi and Awerbuch [36] (referred to henceforth as VA), and of a general atomic object implementation based on the replicated state machine algorithm of Lamport [15] (referred to henceforth as RSM).

First, we observe that VA can be recast as a special case of ABD with the write quorums being the rows and the read quorums being the collumns of the matrix. Therefore, the simulation proof of VA is almost identical to that of ABD. In particular, it is easy to see that the simulation from ABD to PO-Machine in Figure 8 is also a forward simulation from VA to PO-Machine.

To prove atomicity of RSM, we use a simplified version of the PO-Machine, called *TO-Machine*. The TO-Machine constructs a single total order of all the requested operations. In particular, every operation becomes ordered only after it is ordered relative to all the other ordered operations. The TO-Machine is parameterized by the emulated object sequential specification and initial state which are used to compute responses. The simulation proof is based on the observation that in RSM, an operation x becomes ordered once the *local* timestamp at each replica becomes greater than that of x. The full proof appears in [6].

8 Conclusions and 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, such as bounded-timestamp-based constructions (see e.g., [34]), whose proofs have been notoriously difficult and bug-prone. An interesting challenge will be to extend our framework to capture implementations that are not explicitly based on timestamps, for example, the construction that creates atomic bits from safe bits [32]. 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 [29]), 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 [24, 25, 1] based on Byzantine quorums [24].

Yet another interesting application domain for our techniques is the verification of multi-threaded programs based on lock-free synchronization primitives (such as CAS, LL/SC, etc.). This area has recently been receiving an increased attention due to the growing popularity of multi-processor computing platforms, and the introduction of lock-free synchronization primitives into the Java concurrency package.

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 (e.g., a unified version of the PO- and TO-Machines).

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. Myla Archer. TAME: PVS Strategies for special purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1/4), February 2001.
3. 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.
4. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill Publishing Company, UK, 1998.
5. Andrej Bogdanov. Formal verification of simulations between i/o automata. Master's thesis, Massachusetts Institute of Technology, July 2001.
6. G. Chockler, N. Lynch, S. Mitra, and J. Tauber. Proving atomicity: An assertional approach. Technical Report MIT/LCS/TR-XXX, MIT Laboratory for Computer Science, 2005.
7. Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, pages 97–114, 2004.
8. Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Alex A. Shvartsman, and Jennifer L. Welch. GeoQuorums: Implementing atomic memory in ad hoc networks.
9. 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.
10. 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.
11. 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`.
12. 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.
13. 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.
14. 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.
15. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
16. Leslie Lamport. On interprocess communication: Part I and II. *Dist. Comput.*, 1:77–101, 1986.
17. Leslie Lamport. On interprocess communication, Part II: Algorithms. *Distributed Computing*, 1(2):86–101, April 1986.
18. Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):18–25, December 2001.
19. Butler Lampson. The ABCD's of paxos. In *Proceedings of the Twentieth Annual ACM symposium on Principles of Distributed Computing*, Newport, RI, August 2001.
20. N. A. Lynch and M.R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.
21. Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.
22. 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.
23. 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.
24. Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Journal of Distributed Computing*, 11(4):203–213, 1998.
25. 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.
26. 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.
27. 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.
28. A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Comput. Surv.*, 25(3):225–262, 1993.
29. 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.
30. Joshua A. Tauber. *Verifiable Compilation of I/O Automata without Global Synchronization*. PhD thesis, Massachusetts Institute of Technology, Cambridge,MA, September 2004.
31. 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.
32. John Tromp. How to construct an atomic variable. In *LNCS 392, Proc. 3rd International Workshop On Distributed Algorithms*, pages 292–302. Springer-Verlag, 1989.
33. K. Vidyasankar. Concurrent reading while writing revisited. *Distributed Computing*, 4:81–85, 1990.
34. 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.
35. 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.
36. 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.
37. Liqiang Wang and Scott D. Stoller. Static analysis for programs with non-blocking synchronization. In *Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, June 2005.