Distributed Memory Algorithms Coded in IOA: Challenge Problems for Software Analysis and Synthesis Methods

Andrej Bogdanov, Laura Dean, Christine Karlovich
MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139

January 25, 2001

Abstract

This paper describes a collection of input-output automaton models of distributed memory systems specified in the IOA language. The models are analyzed with relation to one another using formal techniques such as invariants and simulation proofs. Part of the verification was performed using the interactive Larch Prover. The models are further intended to be used as test examples for formal analysis tools such as simulators and model checkers and for synthesis tools such as automatic code generators.

1 Introduction

IOA [4] is a programming language for modeling and implementing distributed systems based on the I/O automaton model for reactive systems [6]. It extends and formalizes the descriptive notions used in *Distributed Algorithms* [5], uses Larch specifications to define the semantics of abstract data types and interfaces with analytic tools such as simulators, model checkers and theorem provers.

As a practical tool for distributed system analysis, IOA provides a mechanism for computer-based formal verification of I/O automaton properties. In particular, it allows the use of mathematical techniques for proving assertions about abstract I/O automata such as invariants and simulations to ensure correctness of real world systems implemented in the IOA language. Several theorem proving environments, including PVS, Isabelle and Larch [2] provide formal constructs which facilitate the application of these proof techniques in the setting of I/O automata [1].

We present a collection of examples written in the IOA language, designed to be used as test studies for tools which interface with IOA. The examples describe three commonly used distributed memory models: A single memory model, a strong caching model and a majority voting model. We prove that these three models are equivalent using invariants and simulation proofs. The equivalence of the strong cache and the single memory model, as well as the key invariant of the majority voting model from section 5 were fully verified using the Larch Prover.

The exposition of the proofs in the following sections attempts to capture the essential difficulties faced when using an automated theorem prover. For this reason, the proofs may seem technical. However, this level of technical detail is often necessary when reasoning about complicated high-level systems formally.

The following result, whose proof can be found in [1], will be our main tool for verifying automaton equivalence:

Theorem 1 (Abstraction Theorem). *Let A and B be I/O automata. If there is a simulation relation from A to B, then $traces(A) \subseteq traces(B)$.*

Notation. In statements and proofs about automaton A, we use the following notation:

s_0 is a start state of A,

π is a transition of A,

s is the state of A before transition π, and

s' is the state of A after transition π.

In simulation proofs from automaton A to automaton B, we use s to denote states of A, u to denote states of B and $\alpha(\pi)$ to denote the execution fragment of B which simulates π.

2 Shared specifications of the memory models

In this section we describe the data types used in the specification of the memory models and the external transitions shared between the models. Data type specifications are provided at an abstract level, through a set of *axioms* that each data type must satisfy. The formal description of the data type specifications is written in the Larch Shared Language (LSL).

The following traits are used to describe the data components of the memory models:

`Address` **and** `Value`. An *address* are used to access a particular location in memory. A *value* is the type of the entity stored at each address. These types do not satisfy any axioms. In particular, neither the set of addresses nor the set of values needs to be finite. The value `v0` is the default value to which all memory locations are initialized.

`Node`. Each *node* represents a distributed entity which interacts with the environment through *requests* and *responses*. Each memory model imposes particular responsibilities on the nodes of the system. There is a finite number of nodes, and there is at least one node.[1]

```
Node: trait
includes Set(Node), Integer
introduces
  all_nodes: -> Set[Node],
  num_nodes: -> Int
asserts with n: Node
  \A n (n \in all_nodes);
  size(all_nodes) = num_nodes;
  num_nodes > 0;
```

`CacheLoc`. The `CacheLoc` data structure is used to represent a storage location in memory models where data is cached at the nodes.

[1]The implementation proofs in section 5 rely on these assumptions on trait `Node`.

```
CacheLoc: trait
  CacheLoc tuple of n: Node, addr: Address
```

`Response`. A *response* is reported to the environment after an invocation request is issued at a given node. The `Response` type is not constrained by any axioms.

`Inv`. The `Inv` data type represents *invocations*, abstractions through which the memory model interacts with its environment. Invocations are divided into *read* and *write* invocations, which denote a request to read from and write to memory, correspondingly. The operator `F` relates an invocation to the response that it produces. In particular, it ensures that the value read from a read invocation equals the value reported.

```
Inv: trait
  F_range tuple of r: Response, v: Value
  introduces
    __.addr: Inv -> Address
    isRead: Inv -> Bool
    isWrite: Inv -> Bool
    F: Inv, Value -> F_range
  asserts with inv: Inv, v, v': Value, r: Response
    isRead(inv) \/ isWrite(inv);
    F(inv, v) = [r, v'] /\ isRead(inv) => v = v'
```

The memory models interact with the environment through the following two families of external actions:

`input invoke(i: Inv, n: Node)`. The `invoke` actions are used to specify a read or write invocation at a given node.

`output respond(r: Response, n: Node)`. The `respond` actions produce responses as a result to an invocation at a given node.

3 The central memory model

The `Mem` automaton is an atomic memory algorithm which models a centralized memory system. Its specification in IOA language is shown below.

The internal `compute` transitions perform the read/write computation which is requested by an invocation at a node. The `report` and `reported` data structures are used for bookkeeping of the current invocation locally at each node.

```
automaton Mem(Address, Response : Type)

uses Inv, Node, Null(Inv), Value

signature
  input invoke(i: Inv, n: Node)
  output respond(r: Response, n: Node)
```

```
  internal compute(r: Response, v: Value, n: Node)

states
  mem : Array[Address, Value] := constant(v0),
  invocation : Array[Node, Null[Inv]] := constant(nil),
  report : Array[Node, Response],
  reported : Array[Node, Bool] := constant(true)

transitions
input invoke(i, n)
  eff
    invocation[n] := embed(i)

output respond(r, n)
  pre
    report[n] = r /\
    reported[n] = false
  eff
    reported[n] := true

internal compute(r, v, n)
  pre
    invocation[n] ~= nil /\
    F(invocation[n].val, mem[invocation[n].val.addr]) = [r, v] /\
    reported[n] = true
  eff
    mem[invocation[n].val.addr] := v;
    report[n] := r;
    reported[n] := false
```

4 The strong caching model

The `StrongCache` automaton models a strong caching distributed memory system. This system consists of a *memory*, which acts as a shared storage for the system data and a set of *caches* which contain replicated copies of the memory data. Each node is associated to exactly one cache.

When new data is written to the system, the data is stored in the memory and the caches are invalidated. Data is read from the caches only. Invalidated caches may be updated from the memory. In addition, cache locations may be invalidated at any point.

In this section, we show that the `StrongCache` automaton is equivalent to the `Mem` automaton, in the sense that both automata have the same set of traces. The proof relies on the existence of two simulation relations: one from `StrongCache` to `Mem` and another one from `Mem` to `StrongCache`. We construct and prove these two simulation relations. The proofs presented here were verified formally using the Larch Prover.

4.1 IOA specification of the Strong Cache model

The `StrongCache` automaton performs internal read/write computations using the following transitions:

`compute_write`. These transitions write the value of the current write invocation at a given node to memory and invalidates all cache locations corresponding to the modified memory location.

`compute_read`. These transitions read the value of the current read invocation at a given node from the cache corresponding to that node, if the cache location is valid.

`copy`. These transitions copies the value of a memory location to the corresponding location of a given cache.

`drop`. These transitions invalidates a given cache location.

The `report` and `reported` data structures are used for bookkeeping of requests, as in the `Mem` automaton. The value `nil` is used to describe an invalidated cache location.

```
automaton StrongCache(Address, Response : Type)

uses Inv, CacheLoc, Node, Value, Null(Value), Null(Inv)

signature
  input invoke(inv: Inv, n: Node)
  output respond(r: Response, n: Node)
  internal
    compute_read(r: Response, v: Value, n: Node),
    compute_write(r: Response, v: Value, n: Node),
    copy(a: Address, n: Node),
    drop(a: Address, n: Node)

states
  mem : Array[Address,Value] := constant(v0),
  invocation : Array[Node, Null[Inv]] := constant(nil),
  report: Array[Node, Response],
  reported: Array[Node, Bool] := constant(true),
  cache: Array[CacheLoc, Null[Value]] := constant(nil)

transitions
input invoke(i, n) where isRead(i) \/ isWrite(i)
  eff
    invocation[n] := embed(i)

output respond(r, n)
  pre
    report[n] = r /\
```

```
    reported[n] = false
  eff
    reported[n] := true

internal compute_read(r, v, n)
  pre
    invocation[n] ~= nil /\
    isRead(invocation[n].val) /\
    cache[[n, invocation[n].val.addr]] ~= nil /\
    F(invocation[n].val,
      cache[[n, invocation[n].val.addr]].val) = [r, v] /\
    reported[n] = true
  eff
    report[n] := r;
    reported[n] := false

internal compute_write(r, v, n)
  pre
    invocation[n] ~= nil /\
    isWrite(invocation[n].val) /\
    F(invocation[n].val,
      mem[invocation[n].val.addr]) = [r, v] /\
    reported[n] = true
  eff
    mem[invocation[n].val.addr] := v;
    report[n] := r;
    reported[n] := false;
    for n': Node in Node do
      cache[[n', invocation[n].val.addr]] := nil
    od

internal copy(a, n)
  eff
    cache[[n, a]] := embed(mem[a])

internal drop(a, n)
  eff
    cache[[n, a]] := nil
```

4.2 The equivalence of StrongCache and Mem

We prove the equivalence of StrongCache and Mem by showing that each of the two automata implements the other. The result then follows from the abstraction theorem.

Intuitively, automaton StrongCache implements automaton Mem because the central memory contents of the two automata match. The correct behavior of StrongCache upon read invocations depends on the consistency between its memory and its caches. In formal language, this can be stated as an invariant of StrongCache:

Lemma 1. *Let I be a predicate on the states of* StrongCache *given by*

$$I(s) \iff \forall n : \texttt{Node}, a : \texttt{Address} \quad s.\texttt{cache}[[n,a]] = \texttt{nil} \lor s.\texttt{cache}[[n,a]] = s.\texttt{mem}[a]$$

Then I is an invariant of StrongCache.

Proof. We prove the lemma by structural induction on the reachable states of StrongCache. In the start state s_0, all cache locations are initialized to nil, so $I(s_0)$ is trivially deducible from the start state definition.

The families of actions invoke, respond and compute_read do not modify the contents of mem and cache. For these transitions, the conclusion $I(s')$ follows directly from the assumption $I(s)$.

For the family of actions compute_write, a proof by cases is necessary. For the assignment $a = \texttt{invocation}[n].val.addr$, it follows that $\forall n' : \texttt{Node}\ s'.\texttt{cache}[[n',a]] = \texttt{nil}$ so $I(s')$ holds. Otherwise, both mem and cache remain unchanged and $I(s')$ follows directly from $I(s)$.

A proof by cases shows that $I(s)$ implies $I(s')$ in the case of copy and drop as well. At cache location $[n',a]$, the transition effects result in one of the two conditions for $I(s')$ being satisfied. Otherwise, mem and cache remain unchanged. □

We are now ready to show that StrongCache implements Mem:

Theorem 2. *Let F be the following relation from* States[StrongCache] *to* States[Mem]*:*

$$\begin{aligned} F(s,u) \quad \iff \quad & u.\texttt{mem} = s.\texttt{mem} \land \\ & u.\texttt{invocation} = s.\texttt{invocation} \land \\ & u.\texttt{report} = s.\texttt{report} \land \\ & u.\texttt{reported} = s.\texttt{reported} \end{aligned}$$

then F is a simulation relation.

Proof. We prove the theorem by structural induction on the reachable states of StrongCache. The start states s_0 of StrongCache and u_0 of Mem satisfy $F(s_0, u_0)$ by definition.

For the invoke, respond, compute_write, copy and drop transitions, the simulation proofs are straightforward. We define:

$$\begin{aligned} \alpha(\texttt{invoke(i,n)}) &= \texttt{invoke(i,n)} \\ \alpha(\texttt{respond(r,n)}) &= \texttt{respond(r,n)} \\ \alpha(\texttt{compute\_write(r,v,n)}) &= \texttt{compute(r,v,n)} \\ \alpha(\texttt{copy(a,n)}) &= \epsilon \\ \alpha(\texttt{drop(a,n)}) &= \epsilon \end{aligned}$$

In all cases, it is trivial to check that assuming $F(s,u)$, $\alpha(\pi)$ is always enabled and $F(s',u')$ holds.

For the compute_read transitions, we let $\alpha(\texttt{compute\_read(r,v,n)}) = \texttt{compute(r,v,n)}$. From the Inv axioms and the precondition assumption $\texttt{isRead}(s.\texttt{invocation}[\texttt{n}].\texttt{val})$, it follows that

$$u.\texttt{v} = s.\texttt{cache}[[\texttt{n}, \texttt{invocation}[\texttt{n}].\texttt{val.addr}]]$$

now, using the invariant I, we conclude that

$$u.\texttt{v} = s.\texttt{mem}[\texttt{invocation}[\texttt{n}].\texttt{val}.\texttt{addr}]$$

Since $s.\texttt{mem} = u.\texttt{mem}$, it follows that the transition $\texttt{compute}(\texttt{r},\texttt{v},\texttt{n})$ is enabled. Finally, from the specification of Mem, we have:

$$u.\texttt{v} = u'.\texttt{mem}[\texttt{invocation}[\texttt{n}].\texttt{val}.\texttt{addr}]$$

which, together with the fact $s'.\texttt{mem} = s.\texttt{mem}$ yields the conclusion

$$u'.\texttt{mem}[\texttt{invocation}[\texttt{n}].\texttt{val}.\texttt{addr}] = s'.\texttt{mem}[\texttt{invocation}[\texttt{n}].\texttt{val}.\texttt{addr}].$$

Because the other components of the state remain unmodified for both automata, $F(s', u')$ follows. $\square$

To prove that Mem implements StrongCache, we can show that the inverse of F is a simulation relation from Mem to StrongCache. However, the proof becomes considerably simpler if we choose a more natural relation on the states.

Theorem 3. *Let G be the following relation from* States[Mem] *to* States[StrongCache]*:*

$$\begin{array}{rcl} G(s,u) & \Longleftrightarrow & u.\texttt{mem} = s.\texttt{mem} \wedge \\ & & u.\texttt{invocation} = s.\texttt{invocation} \wedge \\ & & u.\texttt{report} = s.\texttt{report} \wedge \\ & & u.\texttt{reported} = s.\texttt{reported} \wedge \\ & & u.\texttt{cache} = \texttt{constant}(\texttt{nil}) \end{array}$$

then G is a simulation relation.

Proof. We prove the theorem by structural induction on the reachable states of Mem. The simulation relation holds trivially for the start states s_0 and u_0 of Mem and StrongCache.

The $\texttt{invoke}(\texttt{i},\texttt{n})$ and $\texttt{respond}(\texttt{r},\texttt{n})$ transitions carry over naturally from Mem to StrongCache. Therefore, if we define

$$\begin{array}{rcl} \alpha(\texttt{invoke}(\texttt{i},\texttt{n})) & = & \texttt{invoke}(\texttt{i},\texttt{n}) \\ \alpha(\texttt{respond}(\texttt{r},\texttt{n})) & = & \texttt{respond}(\texttt{r},\texttt{n}), \end{array}$$

it is easy to check that $G(s,u)$ implies $G(s',u')$.

For the $\texttt{compute}(\texttt{r},\texttt{v},\texttt{n})$ transition, we make the following definition:

$$\alpha(s.\texttt{compute}(\texttt{r},\texttt{v},\texttt{n})) = \begin{cases} \texttt{compute\_write}(\texttt{r},\texttt{v},\texttt{n}), \text{ if } \texttt{isWrite}(inv), \\ \texttt{copy}(a,\texttt{n}) \circ \texttt{compute\_read}(\texttt{r},\texttt{v},\texttt{n}) \circ \texttt{drop}(a,\texttt{n}) \text{ , otherwise.} \end{cases}$$

where $inv = u.\texttt{invocation}[\texttt{n}].val$ and $a = inv.\texttt{addr}$. We now proceed by cases. If $\texttt{isWrite}(inv)$, then the preconditions of $\texttt{compute}(\texttt{r},\texttt{v},\texttt{n})$ and $\texttt{compute\_write}(\texttt{r},\texttt{v},\texttt{n})$ become equivalent with respect to G. For the effects clauses, the only nontrivial part is to check that $u.\texttt{cache}$ remains $\texttt{constant}(\texttt{nil})$ after the transition. Otherwise, $\texttt{isRead}(inv)$ must hold. In this case, there are three interesting facts to show:

s.mem remains invariant. This follows from the axioms of Inv.

u.cache satisfies the precondition in compute_read(r, v, n). This follows from the effect of the copy(a, n) transition.

u'.cache = constant(nil). Since the same cache entry that was copied from memory was dropped by the drop(a, n) transition, the caches remain unchanged after the simulated execution fragment.

□

5 The majority voting model

The Majority automaton models the majority voting algorithm for a distributed memory system, as described on page 573 in [5]. This system's storage consists of a set of *storages*, one per node, with no centralized memory. Each storage location is also associated to a nonnegative integer *tag*. Initially, all storages are instantiated to the default value v0 and all tags are set to the value 0.

In order to perform a read request at a given node, the model reads at a majority of the copies stored at the invocation address. Among those, it chooses the one with the largest tag and returns the associated value.

To perform a write request at a given node, the model selects a majority M of the nodes in the system. It queries all nodes in M to find the largest tag t. Then it writes the new value to all nodes in M and replaces the tags of M with $t + 1$.[2]

In this section, we show that the Majority automaton is equivalent to the Mem automaton, in the sense that both automata have the same set of traces. The proof relies on the existence of two simulation relations: one from Majority to Mem and another one from Mem to Majority. We construct and prove these two simulation relations. Certain parts of this proof, including the key invariant stated in Lemma 3, were formally verified using the Larch Prover.

5.1 IOA specification of the majority voting model

The Majority automaton performs internal node selection, read, write computations using the following transitions:

choose_majority. This transition selects a majority from the set of nodes. The majority is used when read and write operations are performed.

compute_write. These transitions write the value of the current write invocation to a majority of the nodes and update the tags of those nodes.

compute_read. These transitions read the value of the current read invocation at a given node by querying a majority of the nodes and selecting the value corresponding to the highest tag.

[2]The model in [5] allows using a different majority of nodes for finding the maximum tag and writing the updated (*value*, *tag*) pairs. Our small restriction to this model was adopted in favor of clarity of presentation.

The `report` and `reported` data structures are used for bookkeeping of requests, as in the `Mem` automaton. The set `majority_nodes` is used by the internal transitions to perform operations on a majority of nodes. The `mem_tag` array keeps the tags of each storage location.

```
automaton Majority(Address, Response : Type)

uses Inv, Node, Value, CacheLoc, Null(Inv)

signature
  input invoke(i: Inv, n: Node)
  output respond(b: Response, n: Node)
  internal
    choose_majority
    compute_read(r: Response, v: Value, n: Node)
    compute_write(r: Response, v: Value, n: Node)

states
  invocation: Array[Node, Null[Inv]] := constant(nil),
  report: Array[Node, Response],
  reported: Array[Node, Bool] := constant(true),
  mem: Array[CacheLoc, Value] := constant(v0),
  tag: Array[CacheLoc, Int] := constant(0),
  majority_nodes: Set[Node] := all_nodes

transitions
input invoke(i, n)
  eff
    invocation[n] := embed(i)

output respond(b, n)
  pre
    report[n] = b /\ reported[n] = false
  eff
    reported[n] := true

internal choose_majority
  choose nodes: Set[Node]
  pre
    (2 * size(nodes)) > num_nodes
  eff
    majority_nodes := nodes

internal compute_read(r, v, n)
  choose addr: Address, max_node : Node
  pre
    addr = invocation[n].val.addr /\
    invocation[n] ~= nil /\
```

```
    isRead(invocation[n].val) /\
    max_node \in majority_nodes /\
    \A n': Node (n' \in majority_nodes =>
      (tag[[max_node, addr]] >= tag[[n', addr]])) /\
    F(invocation[n].val, mem[[max_node, addr]]) = [r, v] /\
    reported[n] = true
  eff
    report[n] := r;
    reported[n] := false

internal compute_write(r, v, n)
  choose addr: Address, max_node: Node
  pre
    addr = invocation[n].val.addr /\
    invocation[n] ~= nil /\
    isWrite(invocation[n].val) /\
    max_node \in majority_nodes /\
    \A n': Node (n' \in majority_nodes =>
       (tag[[max_node, addr]] >= tag[[n', addr]])) /\
    F(invocation[n].val, mem[[max_node, addr]]) = [r, v] /\
    reported[n] = true
  eff
    for n' : Node in majority_nodes do
      tag[[n', addr]] := tag[[max_node, addr]] + 1;
      mem[[n', addr]] := v
    od;
    report[n] := r;
    reported[n] := false
```

5.2 The equivalence of `Majority` and `Mem`

We prove the equivalence of `Majority` and `Mem` by showing that each of the two automata implements the other. The result then follows from the abstraction theorem.

The proofs make use of the following key idea about the `Majority` automaton: If a node has the highest tag among a majority of nodes, then it has the highest tag in the whole set of nodes. We formalize and prove a somewhat stronger version of this invariant. We begin with a technical lemma on sets:

Lemma 2. *Let U, S and T be sets such that $|U| = n$, $2 \cdot |S| > n$, $2 \cdot |T| > n$, $S, T \subseteq U$ and $n > 0$. Then $|S \cap T|$ is nonempty.*

Proof. Note that $|S \cup T| \leq n$; Now apply inclusion-exclusion to conclude that $|S \cap T| > 0$.[3] ☐

Let $\texttt{max\_nodes}(a : \texttt{Address})$ denote the set of nodes with maximum tag at address a, or

$$\texttt{max\_nodes}(a) = \{n : \texttt{Node} \mid \forall n' : \texttt{Node}\ \texttt{tag}[[n, a]] \geq \texttt{tag}[[n', a]]\}.$$

[3] The formal verification of this argument from basic set axioms is considerably lenghtier. A proof by structural induction on sets is necessary to derive the inclusion-exclusion principle.

Lemma 3. *For all reachable states s of* `Majority`*,*

$$\forall a\ 2 \cdot |s.\mathtt{max\_nodes}(a)| > \mathtt{num\_nodes} \quad \wedge$$
$$2 \cdot |s.\mathtt{majority\_nodes}| > \mathtt{num\_nodes}.$$

Proof. We prove the lemma by structural induction on the reachable states of `Majority`. In the start state, both the set $s.\mathtt{max\_nodes}(a)$ and $s.\mathtt{majority\_nodes}$ are equal to the set `all_nodes` of size `num_nodes`, so the inequalities are satisfied.

The `invoke`, `respond` and `compute_read` transitions do not modify either of the sets. The `choose_majority` transition modifies set `majority_nodes`, but its effects clause ensures that the invariant is preserved.

We now look at the $\mathtt{compute\_write}(\mathtt{r}, \mathtt{v}, \mathtt{n})$ transitions. The key result to prove here is that $\mathtt{max\_node} \in s.\mathtt{max\_nodes}(a)$, with $a = \mathtt{invocation}[\mathrm{n}].\mathtt{val}.\mathtt{addr}$. Assuming this for a moment, from the effects clause of the transition, we have

$$\forall n_1 \in \mathtt{majority\_nodes}\ s'.\mathtt{tag}[[n_1, a]] = s.\mathtt{tag}[[\mathtt{max\_node}, a]] + 1.$$

By the definition of `max_nodes`, this yields

$$\forall n_1 \in \mathtt{majority\_nodes}, n_2 : \mathtt{Node}\ s'.\mathtt{tag}[[n_1, a]] \geq s.\mathtt{tag}[[n_2, a]] + 1.$$

Proceeding by case $n_2 \in \mathtt{majority\_nodes}$, it is easy to see that

$$\forall n_2 : \mathtt{Node}\ s.\mathtt{tag}[[n_2, a]] + 1 \geq s'.\mathtt{tag}[[n_2, a]],$$

and combining the last two results, we obtain that

$$s.\mathtt{majority\_nodes} \subseteq s'.\mathtt{max\_nodes}(a). \tag{1}$$

The invariant now follows from this equation, the inductive assumption, and the observation that for all addresses other than a the tags remain unmodified.

We prove that $\mathtt{max\_node} \in s.\mathtt{max\_nodes}(a)$ by contradiction. Suppose that $\mathtt{max\_node} \notin s.\mathtt{max\_nodes}(a)$. Then

$$\forall n_1 \in s.\mathtt{max\_nodes}\ s.\mathtt{tag}[[n_1, a]] > s.\mathtt{tag}[[\mathtt{max\_node}, a]].$$

From the precondition of $\mathtt{compute\_write}(\mathtt{r}, \mathtt{v}, \mathtt{n})$, it also follows that

$$\forall n_2 \in s.\mathtt{majority\_nodes}\ s.\mathtt{tag}[[\mathtt{max\_node}, a]] \geq s.\mathtt{tag}[[n_2, a]].$$

Combining these two results, we obtain

$$\forall n_1 \in s.\mathtt{max\_nodes}, n_2 \in s.\mathtt{majority\_nodes}\ s.\mathtt{tag}[[n_1, a]] > s.\mathtt{tag}[[n_2, a]]. \tag{2}$$

From Lemma 2, it follows that the set $s.\mathtt{max\_nodes}(a) \cap s.\mathtt{majority\_nodes}$ is nonempty. Therefore, we can pick an element n_c in this set. Instantiating both n_1 and n_2 by n_c in (2), we obtain

$$s.\mathtt{tag}[[n_c, a]] > s.\mathtt{tag}[[n_c, a]].$$

Contradiction. □

Using this Lemma, we show that Majority implements Mem:

Theorem 4. *Let F be the following relation from* States[Majority] *to* States[Mem]*:*

$$
\begin{aligned}
F(s,u) \iff\ & u.\mathtt{invocation} = s.\mathtt{invocation} \land \\
& u.\mathtt{report} = s.\mathtt{report} \land \\
& u.\mathtt{reported} = s.\mathtt{reported} \land \\
& \forall a : \mathtt{Address}, n \in \mathtt{max\_nodes}(a)\ u.\mathtt{mem}[a] = s.\mathtt{mem}[[n,a]]
\end{aligned}
$$

then F is a simulation relation.

Proof. We prove the theorem by structural induction on the reachable states of Majority. We focus on the relation between $s.\mathtt{mem}$ and $u.\mathtt{mem}$ in the simulation, as the correspondences between the other components of the state are trivial to check.

In the start states, the mem components of both automata are initialized to v0 everywhere, and for all a, $\mathtt{max\_nodes}(a) = \mathtt{all\_nodes}$, so the relation F holds.

The invoke, respond, choose_majority and compute_read transitions preserve the state of $s.\mathtt{mem}$ and the tags of all storage locations, so the proof for these transitions follows directly from the hypothesis $F(s,u)$ and the definitions

$$
\begin{aligned}
\alpha(\mathtt{invoke(i,n)}) &= \mathtt{invoke(i,n)} \\
\alpha(\mathtt{respond(r,n)}) &= \mathtt{respond(r,n)} \\
\alpha(\mathtt{choose\_majority}) &= \epsilon \\
\alpha(\mathtt{compute\_read(r,v,n)}) &= \mathtt{compute(r,v,n)}
\end{aligned}
$$

For the compute_write transitions, we define

$$\alpha(\mathtt{compute\_write(r,v,n)}) = \mathtt{compute(r,v,n)}.$$

For addresses other than $a = \mathtt{invocation[n].val.addr}$, the mem contents remain unchanged for both automata. To finish the proof, it is enough to conclude that $F(s,u)$ implies

$$\forall n \in s'.\mathtt{max\_nodes}(a)\ s'.\mathtt{mem}[[n,a]] = u'.\mathtt{mem}[a].$$

We begin by showing that $s'.\mathtt{max\_nodes}(a) \subseteq s.\mathtt{majority\_nodes}$, the other direction of equation (1). Suppose $n \notin s.\mathtt{majority\_nodes}$. From Lemmas 2 and 3, it follows that the set $s.\mathtt{max\_nodes}(a) \cap s.\mathtt{majority\_nodes}$ is nonempty. Therefore, we can pick a node n_c in this set.

By definition of $s.\mathtt{max\_nodes}(a)$, it follows that

$$s.\mathtt{tag}[[n_c,a]] = s.\mathtt{tag}[[\mathtt{max\_node},a]],$$

from where

$$s.\mathtt{tag}[[n_c,a]] \geq s.\mathtt{tag}[[n,a]].$$

Since $n_c \in s.\mathtt{majority\_nodes}$, but $n \notin s.\mathtt{majority\_nodes}$, it further follows that

$$
\begin{aligned}
s'.\mathtt{tag}[[n,a]] &= s.\mathtt{tag}[[n,a]] \\
s'.\mathtt{tag}[[n_c,a]] &> s.\mathtt{tag}[[n_c,a]].
\end{aligned}
$$

Combining the last three equations, we obtain

$$s'.\mathtt{tag}[[n_c, a]] > s'.\mathtt{tag}[[n, a]].$$

From where we conclude that $n \notin s'.\mathtt{max\_nodes}(a)$. This proves the containment claim.

To finish the proof, we note that the effects clause of the $s.\mathtt{compute\_write}(\mathtt{r}, \mathtt{v}, \mathtt{n})$ transition implies

$$\forall n \in s.\mathtt{majority\_nodes}\ s'.\mathtt{mem}[[n, a]] = \mathtt{v}.$$

This, together with the containment claim, yields

$$\forall n \in s'.\mathtt{max\_nodes}(a)\ s'.\mathtt{mem}[[n, a]] = \mathtt{v}.$$

From the effects clause of $u.\mathtt{compute}(\mathtt{r}, \mathtt{v}, \mathtt{n})$, it follows that

$$u'.\mathtt{mem}[a] = \mathtt{v}.$$

This concludes the proof. □

Theorem 5. *Let G be the following relation from* `States[Mem]` *to* `States[Majority]`*:*

$$\begin{aligned}
G(s, u) \quad \Longleftrightarrow \quad & u.\mathtt{invocation} = s.\mathtt{invocation} \wedge \\
& u.\mathtt{report} = s.\mathtt{report} \wedge \\
& u.\mathtt{reported} = s.\mathtt{reported} \wedge \\
& \forall a : \mathtt{Address}, n : \mathtt{Node}\ u.\mathtt{mem}[[n, a]] = s.\mathtt{mem}[a] \wedge \\
& \forall a : \mathtt{Address}, n, n' : \mathtt{Node}\ u.\mathtt{tag}[[n, a]] = u.\mathtt{tag}[[n', a]] \wedge \\
& u.\mathtt{majority_nodes} = \mathtt{all_nodes}
\end{aligned}$$

then G is a simulation relation.

Proof. We prove the theorem by structural induction on the reachable states of `Mem`. We define

$$\begin{aligned}
\alpha(\mathtt{invoke}(\mathtt{i}, \mathtt{n})) &= \mathtt{invoke}(\mathtt{i}, \mathtt{n}) \\
\alpha(\mathtt{respond}(\mathtt{r}, \mathtt{n})) &= \mathtt{respond}(\mathtt{r}, \mathtt{n})
\end{aligned}$$

and

$$\alpha(\mathtt{compute}(\mathtt{r}, \mathtt{v}, \mathtt{n})) = \begin{cases} \mathtt{compute\_write}(\mathtt{r}, \mathtt{v}, \mathtt{n}), \text{ if } \mathtt{isWrite}(inv), \\ \mathtt{copy}(a, \mathtt{n}) \circ \mathtt{compute\_read}(\mathtt{r}, \mathtt{v}, \mathtt{n}) \circ \mathtt{drop}(a, \mathtt{n}), \text{ otherwise,} \end{cases}$$

with $inv = s.\mathtt{invocation}[\mathtt{n}].\mathtt{val}$.

The states carry over naturally by the $\mathtt{invoke}(\mathtt{i}, \mathtt{n})$ and $\mathtt{respond}(\mathtt{r}, \mathtt{n})$ transitions. For these two transition families, it is trivial to check that the preconditions are satisfied and that $G(s, u)$ implies $G(s', u')$ in all cases. The $\mathtt{compute}(\mathtt{r}, \mathtt{v}, \mathtt{n})$ transition requires a proof by cases; as in the previous proof, we focus on the correspondence between the `mem` and `tag` components of the state of `Majority` and the `mem` component of the state of `Mem`.

If $s.\mathtt{isRead}(inv)$ is true, it is enough to show that the $\mathtt{compute\_read}(r, v, n)$ is enabled for some values of the `choose` parameters. We make the selection

$$\begin{aligned}
\mathtt{addr} &= inv.\mathtt{addr} \\
\mathtt{max_node} &= n \text{ s.t. } \mathtt{tag}[[n, \mathtt{addr}]] = max\{\mathtt{tag}[[n', \mathtt{addr}]] \mid n' : \mathtt{Node}\}
\end{aligned}$$

Note that the existence of such n depends on the following property of sets: A finite set of totally ordered elements has a maximum element. This property can be formally proved by structural induction on sets. It is now easy to check that these values for the `choose` parameters satisfy the precondition of `compute_read`(r, v, n).

Finally, if s.`isRead`(inv) is true, we make the same choice for the `choose` parameters of `compute_write`(r, v, n). We can show that the precondition is satisfied by the same reasoning as for `compute_read`(r, v, n). It remains to see that $G(s, u)$ implies $G(s', u')$.

Since u.`majority_nodes` $=$ u.`all_nodes`, u.`mem`$[[n', \texttt{addr}]]$ gets the value v for all nodes n', so the relation between s.`mem` and u.`mem` holds. Also, for all nodes n', s.`tag`$[[n', \texttt{addr}]]$ is assigned to the same value, so the condition for u.`tag` holds as well. The predicate $G(s', u')$ follows. $\square$

References

[1] Jorgen F. Sogaard-Anderson, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants, "Computed-assisted simulation proofs," *Computer-Aided Verification, Fifth International Conference*, 1993.

[2] Stephen J. Garland and John V. Guttag. "A guide to LP, the Larch Prover." TR-82, DEC Systems Research Center, 1991. Updated version available as `http://nms.lcs.mit.edu/Larch/LP/overview.html`.

[3] Stephen J. Garland and Nancy A. Lynch. "Using I/O automata for developing distributed systems." *Foundations of Component-Based Systems*, Gary T. Leavens and Murali Sitaraman, editors, Cambridge University Press, 2000, pages 285-312.

[4] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: a language for specifying, programming, and validating distributed systems.* MIT Laboratory for Computer Science, 1997 (revised January, 2001).

[5] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[6] Nancy A. Lynch and Mark Tuttle. "Hierarchical correctness proofs for distributed algorithms." Technical Report MIT/LCS/TR-387, MIT Laboratory for Computer Science, 1987.