Investigating Simulations between I/O Automata of Asynchronous Memory Systems

Shien Jin Ong *
MIT Laboratory for Computer Science, Cambridge, MA 02139 USA

August 29, 2001

Abstract

We investigate three memory models which are Central Memory, Strong Cache and Replicated State Machine. A proof technique known as *simulation* is used to prove that Replicated State Machine implements Central Memory. In addition to that, we also show that Strong Cache and Central Memory are equivalent. The three automata were first written in IOA and then their simulation relation and step correspondence were verified with the IOA Simulator. A mathematical proof of Replicated State Machine implementing Central Memory is given and additionally we also show that to guarantee a correct step correspondence between the two automata, the perform transition in Central Memory must occur in between the perform_first and respond transition in Replicated State Machine.

1 Introduction

1.1 Input/Output Automaton Overview

The Input/Output(I/O) automaton model, developed by Nancy Lynch and Mark Tuttle [8], models components in asynchronous concurrent systems as labeled transition systems. It is a very robust and general model of describing asynchronous distributed systems.

To describe an I/O automaton formally, we use the language of IOA. A good manual describing the IOA language can be found in [10].

1.2 Proof Methods for I/O Automaton

After describing an I/O automaton in IOA, it is now possible to check certain properties of the automaton. We may to prove interesting *invariants* or perhaps show that an automaton A *implements* another automaton B. In order to do this, we need to either prove it mathematically by hand or using the Larch Prover. Although the Larch Prover(LP) is less prone[1] to give erroneous proofs, LP is extremely tedious and difficult to use[2]. Consequently, in this paper, we will prove all our assertions mathematically without the use of LP.

To aid our proof, we exploited the IOA Simulator which can check proposed invariants, simulation relations and step correspondences. Laura Dean's thesis [5] includes a user guide to the simulator.

*E-mail: shienjin@mit.edu. The author welcomes any comments, suggestions and criticisms.
[1]Assuming that the Larch Prover is bug-free.
[2]Constitutes only the author's opinion.

1.3 Paper Overview

In this paper, we will investigate three asynchronous distributed memory systems: Central Memory (the atomic object), Strong Cache and Replicated State Machine. Section 2 gives a description of the Central Memory and Strong Cache, and goes on to prove that the two automata are equivalent. The proof that Replicated State Machine implements Central Memory is given in Section 3. We go on further in Section 3.3 by showing that the `perform` transition's timing in Central Memory is essential to guarantee a correct step correspondence. Section 4 describes the capabilities of the IOA Simulator in checking proposed properties of an I/O Automaton. Finally, in Section 5 we conclude by proposing four enhancements to the IOA Simulator.

2 Central Memory and Strong Cache

In the study of asynchronous distributed memory algorithms, we introduce two basic automata which are Central Memory and Strong Cache. Basically, Central Memory models an atomic object. Strong Cache is essentially the same as Central Memory, expect that there are caches in between the nodes and the central memory. Because of their similarity, it is easy to prove the equivalence of the two automata (that is Strong Cache implements Central Memory and vice versa). We will first describe the two automata and then present the simulation relation and step correspondence between them. For a formal proof, the reader should refer to [3] or [1].

Since all the distributed algorithms described in this paper utilizes the same input/output transitions, we will describe them here before proceeding to describe Central Memory and Strong Cache.

The common input transition is `invoke(v, n)` where v is of type **Invocation** and n is of type **Nat** (natural numbers). v represents an invocation and can take values of natural numbers or messages like read or write. If $v = write(x)$, then v is represented as a natural number and `v = val(x)`. If $v = read$, then v is represented as `msg(read)`.

The node where is invocation is requested, on the other hand, is represented as n. Since nodes can be numbered, n can be easily represented as a natural number. Therefore, if the user issues a $write(5)$ invocation at node 2, `invoke(val(5), 2)` will occur and if the user issues a read invocation at node 5, `invoke(msg(read), 5)` occurs.

The common output transition is `respond(r, n)` where as before, r is of type **Invocation** and n is of type **Nat**. v represents an response and can take values of natural numbers or an OK message. If r is a response to a write, then v is represented as `msg(OK)`. If r is a response to a read and the value of the central memory is x, then r is represented as a natural number and `r = val(x)`. In particular, if the automaton responds to a read at node 5 when the central memory is 20, then a `respond(val(20), 5)` occurs. If, however, the automaton responds to a write(3) at node 1, then a `respond(msg(OK), 1)` will occur.

2.1 Central Memory

The Central Memory automaton consist of n nodes[3] with each node having its own private request(`req`) and respond(`resp`) state. Its IOA description is given in Figure 1. We represent the Central Memory automaton as `CentralMem` and throughout this paper, we will use both names interchangeably.

When a user issues a invocation to node n, the invocation is stored in `req[n]`. Responses are stored in `resp[n]`, and `perform(n)` performs a pending invocation in `req[n]` to the centralized memory and reports the responses to `resp[n]`.

Given below is the description of the states of `CentralMem`:

[3]We can think of each node as a client and the centralized memory as a server connected to all the clients.

State	Description
mem	Centralized memory of the automaton. All nodes can modify this memory.
resp[n]	Value of response at node n (only visible to its own node).
req[n]	Value of the pending request invocation at node n (only visible to its own node).

2.2 Strong Cache

Similar to Central Memory, the Strong Cache automaton(`StrongCache`), as illustrated in Figure 2, consists of n nodes and a centralized memory. However, between each node and the central memory, there is a cache. To update or remove values from the cache, we have two additional internal transitions: copy and drop. The transition `copy(n)` copies the current value of the memory to the node n's cache(`cache[n]`), and `drop(n)` just nullifies the value of the `cache[n]`.

Given below is the description of the *additional* state of `StrongCache`:

State	Description
cache[n]	Value of cache at node n (only visible to its own node).

2.3 Simulation relation between Central Memory and Strong Cache

To show the equivalence of Central Memory and Strong Cache, we need to give the relation of between the states and transitions of both the automata. The definition of *simulation relation* is given in [7] and the definition of *equivalent* is given below:

Definition 1 *Automata A and B are* ***equivalent*** *if and only if automaton A implements automaton B and automaton B implements automaton A.*

Lemma 1 *The step correspondence between* `CentralMem` *and* `StrongCache` *is defined as below:*

Central Memory	$\rightarrow$	**Strong Cache**
`invoke(v, n)`	$\rightarrow$	`invoke(v, n)`
`respond(r, n)`	$\rightarrow$	`respond(r, n)`
`perform(n)`	$\rightarrow$	`copy(n), perform(n), drop(n)`

The step correspondence between `StrongCache` *and* `CentralMem` *is defined as below:*

Strong Cache	$\rightarrow$	**Central Memory**
`invoke(v, n)`	$\rightarrow$	`invoke(v, n)`
`respond(r, n)`	$\rightarrow$	`respond(r, n)`
`perform(n)`	$\rightarrow$	`perform(n)`
`copy(n)`	$\rightarrow$	$\emptyset$
`drop(n)`	$\rightarrow$	$\emptyset$

Theorem 1 *The binary relation f over c: states*(`CentralMem`) *and s: states*(`StrongCache`) *defined by*

$$f(c, s) \Leftrightarrow c.mem = s.mem \wedge c.req = s.req \wedge c.resp = s.resp$$

is a simulation relation from `CentralMem` *to* `StrongCache` *and also from* `StrongCache` *to* `CentralMem`*.*

Remark We will omit the proof of Lemma 1 and Theorem 1 as they are simple to prove and available in [3] and [1].

Corollary 1 `StrongCache` *implements* `CentralMem` *and* `CentralMem` *implements* `StrongCache`*.*

Proof Follows immediately from Theorem 1.

```
automaton CentralMem

signature
 input invoke(v: Invocation, n: Nat)
 output respond(r: Invocation, n: Nat)
 internal perform(n: Nat)

states
  mem:  Nat := 0,
  req:  Array[Nat, Invocation] := constant(msg(nil)),
  resp: Array[Nat, Invocation] := constant(msg(nil))

transitions
 input invoke(v, n)
  eff req[n] := v

 output respond(r, n)
  pre resp[n] = r ∧ resp[n] ≠ msg(nil)
  eff req[n] := msg(nil);
     resp[n] := msg(nil)

 internal perform(n)
  pre req[n] ≠ msg(nil) ∧ resp[n] = msg(nil)
  eff if req[n] = msg(read) then resp[n] := val(mem)
          else mem := req[n].val;
              resp[n] := msg(OK)
     fi
```

Figure 1: IOA description of Central Memory

```
automaton StrongCache

signature
 input invoke(v: Invocation, n: Nat)
 output respond(r: Invocation, n: Nat)
 internal perform(n: Nat), copy(n: Nat), drop(n: Nat)

states
  mem:   Nat := 0,
  cache: Array[Nat, Invocation] := constant(msg(nil)),
  req:   Array[Nat, Invocation] := constant(msg(nil)),
  resp:  Array[Nat, Invocation] := constant(msg(nil))

transitions
 input invoke(v, n)
  eff req[n] := v

 output respond(r, n)
  pre resp[n] = r ∧ resp[n] ≠ msg(nil)
  eff req[n] := msg(nil);
      resp[n] := msg(nil)

 internal copy(n)
  eff cache[n] := val(mem)

 internal drop(n)
  eff cache[n] := msg(nil)

 internal perform(n)
  pre req[n] ≠ msg(nil) ∧ resp[n] = msg(nil)
      ∧ (req[n] ≠ msg(read) ∨ cache[n] ≠ msg(nil))
  eff if req[n] = msg(read) then resp[n] := cache[n]
         else mem := req[n].val;
              resp[n] := msg(OK);
              cache := constant(msg(nil))
     fi
```

Figure 2: IOA description of Strong Cache

3 Simulating a Single State Machine: Replicated State Machine

In this section, we will describe Lamport's Replicated State Machine [6], an asynchronous memory algorithms, and try to prove that it implements an atomic object. For simplicity, we will represent the Replicated State Machine automaton as `RSM` and these two terms will be used interchangeably. We will model the atomic object as the Central Memory automaton. Consequently, this is equivalent to showing that every trace produced by the Replicated State Machine algorithm is a valid trace of Central Memory. In order to prove this, we will need to use `simulation relation`.

After describing the `RSM` automaton, we will formally prove, using mathematical techniques, that Replicated State Machine implements Central Memory. Furthermore, we extend our result in showing that the timing of the perform transition in the Central Memory automaton can happen no later than `respond` transition and no earlier than the `perform_first` [4] transition in the Replicated State Machine automaton.

The reader should note that Bogdanov,Tauber and Vyzovitis [2] proved that an intermediate model, which they called `synch`, implements Central Memory using the Larch Prover. We will take a slightly different approach as we do not use the Larch Prover. Therefore, the IOA specification of Replicated State Machine and Central Memory will differ slightly from the ones used by them in [2].

3.1 Replicated State Machine

The Replicated State Machine description is shown in Figure 5. Like the previous two automata (`CentralMem` and `StrongCache`), the `RSM` automaton consists of n nodes. Each node has its own local copy of the memory and stores all the invocation it has heard but yet to perform in its (invocation) buffer. Nodes communicate with each other via FIFO[5] channels.

When a node receives a local invocation, its first transforms it into a **TimedInvocation** using the rules of Lamport's *LogicalTime* [6]. Then, the node will broadcast the invocation to every other node and stores it in its own buffer. A node places a foreign invocation into its buffer when it receives that invocation from a desginated channel. The LSL specification of **TimedInvocation** is given in Figure 3. Michael Tsai implemented this new type in the simulator.

To extract the minimum-time invocation from the buffer, we use **PriorityMinQueue**. This data type whose description is shown in Figure 4, is implemented in the simulator by Atish Dev Nigam and Michael Tsai.

Given below is the description of the states of `RSM`:

State	Description
mem[n]	Node n's local copy of memory (only visible to its own node)
resp[n]	Value of response at Node n. (only visible to its own node)
buffer[n]	Invocation buffer for Node n
channel[[i,j]]	FIFO channel from Node i to Node j.
clock[[i, j]]	If $i = j$, then it represents the current time of Node i Otherwise, it represents Node j's view of the clock of Node i.

Note that the state `v: TimedInvocation` is not a *real* state and was used to shorten our code in the *effect* clause of `perform(n)`.

For more information about the Replicated State Machine algorithm, the reader should refer to Section 18 of Lynch's *Distributed Algorithms* [7].

[4]The definition of `perform_first` will be given in Section 3.2.

[5]FIFO means first in, first out.

```
TimedInvocation(E): trait
introduces
  make: E, Nat, Nat → TimedInvocation[E]
  __.inv:  TimedInvocation[E]                      → E
  __.node: TimedInvocation[E]                      → Nat
  __.time: TimedInvocation[E]                      → Nat
  set_inv: TimedInvocation[E], Invocation          → TimedInvocation[E]
  set_node: TimedInvocation[E], Nat                → TimedInvocation[E]
  set_time: TimedInvocation[E], Nat                → TimedInvocation[E]
  __>__: TimedInvocation[E], TimedInvocation[E]    → Bool
asserts with v, v1: E, n, n1: Nat, t, t1: Nat, t, t1: TimedInvocation[E]
  sort TimedInvocation[E] generated by make;
  sort TimedInvocation[E] partitioned by .inv, .node, .time;
  make(v, n, t).inv  = v;
  make(v, n, t).node = n;
  make(v, n, t).time = t;
  set_inv(make(v, n, t), v1)  = make(v1, n, t);
  set_node(make(v, n, t), n1) = make(v, n1, t);
  set_time(make(v, n, t), t1) = make(v, n, t1);
  t > t1 ⇔ (t.time > t1.time) ∨ ( t.time = t1.time ∧ t.node > t1.node)
```

Figure 3: LSL description of TimedInvocation

```
PriorityMinQueue(E, <:E,E→Bool): trait
  assumes TotallyOrdered(E)
  includes Count(E, PMQ, empty, add, count),
           Size(E, PMQ, empty, add, len)
  introduces
    empty:              → PMQ[E]
    add:     E, PMQ[E]  → PMQ[E]
    __in__:  E, PMQ[E]  → Bool
    head:    PMQ[E]     → E
    tail:    PMQ[E]     → PMQ[E]
    isEmpty: PMQ[E]     → Bool
  asserts with e, e1: E, q: PMQ[E]
    sort PMQ[E] generated by empty, add;
    sort PMQ[E] partitioned by head, tail, isEmpty;
    e in q ⇔ count(e, q) > 0;
    head(add(e, q)) =
      (if q = empty ∨ e < head(q) then e else head(q));
    tail(add(e, q)) =
      (if q = empty ∨ e < head(q) then q else add(e, tail(q)));
    isEmpty(q) ⇔ q = empty
  implies with e, e1, e2: E, q: PMQ[E]
    traits Container(E, PMQ, empty, add);
    add(e1, add(e2, q)) = add(e2, add(e1, q));
    add(e, q) ≠ empty;
    converts count, in, head, tail, len, isEmpty
      exempting head(empty), tail(empty)
```

Figure 4: LSL description of PriorityMinQueue

```
type Message = enumeration of read, write, OK, dummy, nil
type Invocation = union of val: Nat, msg: Message
uses TimedInvocation(Invocation)
uses PriorityMinQueue(TimedInvocation[Invocation])
type Index = tuple of first: Nat, second: Nat

automaton RSM
signature
 input invoke(v: Invocation, n: Nat)
 output respond(r: Invocation, n: Nat)
 internal receive(i: Nat, j: Nat), perform(n: Nat)

states
  mem: Array[Nat, Nat] := constant(0),
  resp: Array[Nat, Invocation] := constant(msg(nil)),
  clock: Array[Index, Nat] := constant(0),
  channel: Array[Index, Seq[TimedInvocation[Invocation]]] := constant({}),
  buffer:  Array[Nat, PMQ[TimedInvocation[Invocation]]] := constant(empty),
  v: TimedInvocation[Invocation]

transitions
 input invoke(v, n)
  eff clock[[n,n]] := clock[[n,n]] + 1;
     buffer[n] := add(make(v, n, clock[[n,n]]), buffer[n]);
     for i: Nat so that i > 0
       do channel[[n,1]] := channel[[n,1]] ⊢ make(v, n, clock[[n,n]]) od

 output respond(r, n)
  pre resp[n] = r ∧ resp[n] ≠ msg(nil)
  eff resp[n] := msg(nil)

 internal receive(i, j)
  pre len(channel[[i,j]]) ≠ 0 ∧ i ≠ j
  eff clock[[j,j]] := max(clock[[j,j]], head(channel[[i,j]]).time);
     clock[[i,j]] := head(channel[[i,j]]).time;
     buffer[j] := add(head(channel[[i,j]]), buffer[j]);
     channel[[i,j]] := tail(channel[[i,j]])

 internal perform(n)
  pre ¬ isEmpty(buffer[n]) ∧ ∀ i: Nat (head(buffer[n]).time < clock[[i, n]])
  eff v := head(buffer[n]);
     if v.node = n then
       if v.inv = msg(read)
         then resp[n] := val(mem[n])
         else mem[n] := (v.inv).val; resp[n] := msg(OK)
       fi
     elseif v.inv ≠ msg(read) then mem[n] := (v.inv).val
     fi;
     buffer[n] := tail(buffer[n])
```

Figure 5: IOA description of Replicated State Machine

3.2 Replicated State Machine implements Central Memory

Before attempting to prove that `RSM` implements `CentralMem`, we need a few definitions and lemmas.

Definition 2 (Perform_first transition) *The transition* `perform_first(n)` *represents the* `perform(n)` *transition when applied to an invocation for the first time in* `RSM`*. When the same invocation is performed other times at different nodes, we represent that* `perform(n)` *transition as* `perform_other(n)`*.*

Definition 3 (Types of invocations) *An invocation π is a* **local** *invocation of node n if $\pi.node = n$. Otherwise, it is a* **foreign** *invocation.*

Definition 4 (Ordering of TimedInvocation) *TimedInvocation π is greater than TimedInvocation π' if and only if $(\pi.time > \pi'.time) \vee (\pi.time = \pi'.time \wedge \pi.node > \pi'.node)$.*

Remark We need not worry about the case when two TimedInvocations have the same logical time and node value since according the rules of Lamport's *LogicalTime* [6], this cannot happen.

Definition 5 (History variables) *We define* **Recent** *to take the value of the node in which the transition* `perform_first` *last occurred in* `RSM`*.*

LatestResp(n) *is defined to be the value of* `resp[n]` *in automaton* `RSM` *if* `resp[n]` *is not null and if it is null,* **LatestResp(n)** *is the future value of* `resp[n]` *when all the invocation up and until node n's local invocation is processed.*

We define **PerformNode** *take the value of $\pi.node$ where π is the invocation last performed in* `RSM`*.*

We also make an assumption on the environment, that is `invoke(n)` can only occur after the `respond(n)` has occurred for the previous invocation. This means that before firing `invoke(n)`, no invocations can be pending at node n.

Having all these definitions in mind, we begin our proof with a key lemma:

Lemma 2 (Total order) *Each node performs its invocations in the order of their logical times. Stated formally, this means that for all $\pi, \pi' \in$* `buffer[n]`*, if $\pi > \pi'$ then π is performed after π'.*

Proof When a node n performs invocation π, it explicitly checks that invocation π is indeed the minimum TimedInvocation pending. Furthermore, it ensures that the known-times of all the other nodes is more than the logical time of π. The FIFO property of the channels between each node implies that node n will never receive an invocation that has logical time less than π. The proof of this lemma is also given in Lynch's *Distributed Algorithms* [7].

Theorem 2 *The binary relation f over r: states*`(RSM)` *and c: states*`(CentralMem)` *defined by*

$$
\begin{aligned}
f(r,c) \quad \Leftrightarrow \quad & r.mem[Recent] = c.mem \\
\wedge \quad & \forall n \in \mathbb{N}(r.resp[n] = c.resp[n] \vee (r.resp[n] = msg(nil) \wedge c.resp[n] = LatestResp(n)))
\end{aligned}
$$

is a simulation relation from `RSM` *to* `CentralMem`*.*

Proof To prove that the above *simulation relation* is correct, we need a correct step correspondence.

The step correspondence between `RSM` and `CentralMem` is defined as below:

RSM	$\to$	**CentralMem**
`invoke(v, n)`	$\to$	`invoke(v, n)`
`respond(r, n)`	$\to$	`respond(r, n)`
`perform_first(n)`	$\to$	`perform(PerformNode)`
`perform_other(n)`	$\to$	$\emptyset$
`receive(n)`	$\to$	$\emptyset$

The start state of `RSM` is when all the buffer and channels are empty, the response and memory state are nil, and the clocks of the nodes are at an arbitrary predetermined time.

Clearly the start state of `CentralMem`, which is when the memory, respond and request state are nil, has a relation to the start state of `RSM`.

It is easy to see that the `invoke`, `respond` and `receive` transitions in `RSM` along with their corresponding transitions in `CentralMem` preserves the state relation.

Next, we consider the transition `perform_other(n)`. The corresponding transition in `CentralMem` is $\emptyset$. Since `perform_other(n)` does not effect the value of *Recent* and $Recent \neq n$, clearly $r.mem[Recent] = c.mem$ still holds. If `perform_other(n)` happens to a **local** invocation, then $c.resp[n]$ will change from $msg(nil)$ to *LatestResp* Since we know that $c.resp[n] = msg(nil)$ before the transition `perform_other(n)` occurred, $c.resp[n] = LatestResp = r.resp[n]$. However, if `perform_other(n)` happens to a **foreign** invocation, *r.resp* and *c.resp* do not change and therefore the state relation still holds.

The most difficult transition to analyze is the `perform_first(n)` transition. We know that the precondition of the transition `perform(PerformNode)` in Central Memory must be true since there is a pending invocation in $c.req[PerformNode]$. Next, we consider two cases:

1. **Case 1 (perform_first(n) on local invocation):** In this case, $Recent = PerformNode = n$. Then, $r.mem[Recent] = r.mem[n] = c.mem$ after the transition. Also, *r.resp* is modified only at node n, and therefore $r.resp[n] = c.resp[n]$. Hence, the state relation holds.
2. **Case 2 (perform_first(n) on foreign invocation):** If it is a foreign invocation, then $PerformNode \neq n$ but $Recent = n$. Hence, $r.mem[Recent] = r.mem[n] = c.mem$ after the transition. Since we perform a foreign invocation in Replicated State Machine, *r.resp* remains the same and the only change to *c.resp* is in node *PerformNode*. We see that $c.resp[PerformNode] = LatestResp[PerformNode]$ because there is an local invocation is node *PerformNode* that is still pending.

3.3 Timing of Perform transition in Central Memory

In the last subsection, we proposed a step correspondence along with a state relation between Replicated State Machine and Central Memory and proved that it is correct. Since Replicated State Machine implements Central Memory, there could be infinitely many step correspondence that are correct. However, we will prove that the `perform` transition for a particular invocation π in Central Memory must happen between the `perform_first` and `respond` transition for the same invocation π in Replicated State Machine.

To prove this fact, we need to formalize what we mean by a step correspondence.

Definition 6 *Assume that automaton A implements automaton B. A step correspondence from A to B is an algorithm* [6] *that takes a valid finite execution of A (σ_A) as an input and outputs a corresponding valid finite execution of B (σ_B) such that $trace(\sigma_A) = trace(\sigma_B)$.*

To aid our proof, the parameter for `perform`, `perform_first` and `respond` will **not** be node n, as in the previous subsection, but instead will be invocation π. This means that when we say `perform(`π`)` occurred, we mean that a invocation π is performed in Central Memory.

By `perform_first(`π`)`, we mean that we perform invocation π for the first time in Replicated State Machine and `respond(`π`)` means that we respond to an invocation that has already been performed in Replicated State Machine (**not** Central Memory).

[6] An algorithm is essentially a Turing Machine.

Theorem 3 *In any valid step correspondence from Replicated State Machine to an atomic object (in our case, Central Memory), the* `perform(`π`)` *transition in the atomic object cannot happen before the* `perform_first(`π`)` *transition and cannot happen after the* `respond(`π`)` *transition in Replicated State Machine.*

Remark Note that this theorem is general, that is our atomic object need not be Central Memory and invocations π can represent anything. A generalized description of Replicated State Machine is given in [7]. Therefore, the reader should keep in mind that while we use `CentralMem` in our proof below, the proof is valid for any atomic object.

Proof First, we will prove that `perform(`π`)` cannot happen before `perform_first(`π`)`. Assume by contradiction that we have a valid step correspondence that takes in a certain execution of `RSM` (σ_{RSM}) and outputs σ_{CM} (a valid execution of `CentralMem`) such that σ_{CM} ends with `perform(`π`)` but there is no `perform_first(`π`)` in σ_{RSM}. Then, we as the adversary can foil this step correspondence from being correct. We just need to append a series of `receive` transitions after σ_{RSM} until all the channels in `RSM` are empty, and then perform invocations with logical time less than π. After that, we will respond at any node n if `resp[n]` $\neq$ `msg(nil)`. Since we responded at node n in `RSM`, we must also respond at node n in `CentralMem`. However, in general, the value of the response cannot be equal since we have performed invocation π in `CentralMem` but have yet to perform any n copies of the invocation π in `RSM`. Therefore, the step correspondence is not valid.

Next, we will prove that `perform(`π`)` cannot happen after `respond(`π`)`. This part of the proof is easy since when we fire a `respond(`π`)` in `RSM` we need to fire the similar respond transition in `CentralMem`. However, it is clear that we cannot respond to invocation π in `CentralMem` unless we have performed it.

Our main purpose of proposing and proving this theorem is because there were attempts to prove that `RSM` implements `CentralMem` in the Larch Prover but failed. Although there were no published results on the failed attempts, my conversations with Joshua Tauber revealed that initially, he and some of his colleagues tried a step correspondence that fired the `perform(`π`)` transition in `CentralMem` once all the clocks in `RSM` were beyond the logical time of π. This step correspondence is false because there may be executions whereby the `perform(`π`)` transition in `CentralMem` occurs before the `perform_first(`π`)` transition in `RSM`. The IOA Simulator, as we see in Section 4, is capable of detecting errors in step correspondence and state relation between I/O Automata.

4 Checking Properties of I/O Automaton

Often when we design a new distributed algorithm, we want to ensure that it is correct (i.e. it has certain properties that we are interested in). A good way to do this is to describe the algorithm as an I/O Automaton and then test it in the IOA Simulator. While the Simulator is not capable of proving that an automaton is correct or a certain property is always true of that automaton, it can detect errors if there are any. In this section, we will see a few examples illustrating the capabilities of the Simulator. For those who are unfamiliar with the Simulator, a user guide can be found in [5]. For further reading, Chefter's [4] and Ramirez-Robredo's [9] theses contain the implementation and design of the Simulator.

4.1 Verifying Invariants of IOA automata

The simulator can check for specified invariants in an IOA automaton. For example an interesting invariant of StrongCache is ∀ `n: Nat (cache[n] = msg(nil)` ∨ `cache[n].val = mem)`.

Currently, the simulator cannot handle universal or existential quantifiers. Further discussion on this matter can found in Section 5.1. Therefore, we can only test for the invariant for finitely many values of n. For simplicity, we will test it for $1 \leq n \leq 5$. To test this invariant in the simulator, we need to write the following additional statement:

```
invariant A of StrongCache:
   (cache[1] = msg(nil) ∨ cache[1].val = mem)
 ∧ (cache[2] = msg(nil) ∨ cache[2].val = mem)
 ∧ (cache[3] = msg(nil) ∨ cache[3].val = mem)
 ∧ (cache[4] = msg(nil) ∨ cache[4].val = mem)
 ∧ (cache[5] = msg(nil) ∨ cache[5].val = mem)
```

The reader should note that the invariant statement goes **after** the IOA code and the scheduling block of `StrongCache`. The scheduling block is shown in Figure 6.

Furthermore, to illustrate the capability of the simulator in detecting errors or false invariant assertions, we made up a false invariant of `StrongCache`, which we called *Invariant B* and is shown below:

```
invariant B of StrongCache:
   req[1] = resp[1]
 ∧ req[2] = resp[2]
 ∧ req[3] = resp[3]
 ∧ req[4] = resp[4]
 ∧ req[5] = resp[5]
```

We tested the two invariants assertion of `StrongCache` with the Simulator running 10 steps of the automaton and the result is shown in Figure 7. The simulator, as seen, took only 4 steps to verify that *Invariant B* is false and throughout the 10 steps, *Invariant A* was verified to be true. Since *Invariant A* is actually a correct invariant, no matter how many steps we run the Simulator on, the invariant will not fail.

4.2 Checking Implementation of Automata

Now that we seen an example of using the simulator to verify an invariant, the next and perhaps the most important use of the simulator is to check whether or not an Automaton A implements an Automaton B. As seen in Section 3.2, we proved that Replicated State Machine implements the atomic object (Central Memory). The proof was rather long and tedious, consequently to save time, it would ideal if we could somehow check whether what we are proving is actually true. The Simulator allows us to test-run both the automata simultaneouly, with the Simulator

```
schedule
states
die: Nat,
Node: Nat,
read_write: Bool,
value: Nat
do
  while true do
     die := randomNat(1,4);
     Node := randomNat(1,5);
     read_write := randomBool;
     value := randomNat(1,100);
     if die = 1 ∧ req[Node] = msg(nil) then
        if read_write then fire input invoke(msg(read), Node)
   else fire input invoke(val(value), Node)
fi
     elseif die = 2 ∧ resp[Node] ≠ msg(nil) then
        fire output respond(resp[Node], Node)
     elseif req[Node] ≠ msg(nil) ∧ resp[Node] = msg(nil)
           ∧ (req[Node] ≠ msg(read) ∨ cache[Node] ≠ msg(nil))
           then fire internal perform(Node)
     elseif die = 3 then fire internal copy(Node)
     elseif die = 4 then fire internal drop(Node)
     fi
 od
od
```

Figure 6: Scheduling block of Strong Cache

```
Automaton initialized
1: internal copy(2) in automaton StrongCache
2: internal drop(3) in automaton StrongCache
3: internal drop(2) in automaton StrongCache
4: input invoke(val(69), 3) in automaton StrongCache
>>>> Invariant B failed
5: internal drop(4) in automaton StrongCache
>>>> Invariant B failed
6: internal perform(3) in automaton StrongCache
>>>> Invariant B failed
7: internal copy(3) in automaton StrongCache
>>>> Invariant B failed
8: internal drop(5) in automaton StrongCache
>>>> Invariant B failed
9: output respond(msg(OK), 3) in automaton StrongCache
10: internal copy(2) in automaton StrongCache
No errors
```

Figure 7: Simulator Output of Strong Cache

following the scheduling block of the low-level automaton[7] while at the same time simulate the high-level automaton using the step correspondence specified by the user. After each step, the Simulator also check whether or not the simulation relation[8] between the two automata holds. If it does not, the Simulator halts and returns an error.

Shown in Figure 8 is the scheduling block for Replicated State Machine and in Figure 9, the scheduling clock for Central Memory. The step correspondence and the simulation relation between them is shown in Figure 13. Note that the `RSM` automaton that we are using is different from that in Figure 5. We will be using the `RSM` automaton [9] in Figures 10, 11 and 12 since that automaton specification is suited for the simulator and incorporates history variables. Nevertheless, both automata describe essentially the same asynchronous distributed memory system.

The main difference, however, between this *simulator-ready* `RSM` model and the one we had in Section 3.2 is the addition of a couple of internal transitions and history variables listed below:

- Internal transitions `perform_first` and `perform_other`: See explanation in Section 3.2.
- Internal transitions `perform_dummy` (performs a dummy invocation) and `broadcast_dummy` (broadcast a dummy invocation to all other nodes).
- Internal transition `InitializeClock` (randomized the starting time of the clock for all nodes).
- History variables `PerformNode`, `Recent` and `LatestResp`: See explanation in Section 3.2
- History variables `History` (keeps track of the invocations that have been performed at each node), `Counter` (counts the number of non-dummy invocations performed at each node), and `MaxCounter` (maximum over all counters).

We ran the Paired Simulator (`psim`) on both the Replicated State Machine and Central Memory automata to check our proposed simulation relation and step correspondence. After running the Paired Simulator on 500 steps, there were no errors reported and therefore, we can be rather certain that indeed our simulation relation and step correspondence is true.

Our main difficulty with running simulations on automata lies in ensuring the liveness of both automata. If the scheduling block for `RSM` is not programmed well, the simulator will perhaps never fire a `perform_first` or `perform_other` transition. Several modifications to our scheduling block had to be made before finally achieving a good result: 26 `perform_first` and 102 `perform_other` occured during the 500 step simulation of `RSM` and `CentralMem`. That constitutes to more than 25 percent of perform transitions happening in `RSM`. We managed to have such a high number of perform transitions occurring since we added the capability of the `RSM` automaton to broadcast dummy messages. A summary of the Paired Simulator output is shown in Figure 14.

[7] If A implements B, then A is the low-level automaton and B is the high-level automaton.

[8] We can think of the user-specified simulation relation as an invariant.

[9] The long IOA specification of `RSM` is broken up into 3 parts.

```
schedule
states
Node1: Nat,
Node2: Nat,
value: Nat,
invocation: TimedInvocation[Invocation]
do
 fire internal InitializeClock;
 while true do
   Node1 := 1;
   while Node1 ≤ 5 do
    value := randomNat(1,100);
    if randomNat(1, 20) ≠ 1 ∧ req[Node1] = msg(nil) then
       if randomBool then fire input invoke(msg(read), Node1)
          else fire input invoke(val(value), Node1)
       fi
       elseif randomNat(1, 20) = 1 then  fire internal broadcast_dummy(Node1)
    fi;

    if randomBool ∧ ¬isEmpty(buffer[Node1]) ∧ head(buffer[Node1]).inv = msg(dummy)
       then fire internal perform_dummy(Node1)
    fi;

    if ¬isEmpty(buffer[Node1]) ∧ head(buffer[Node1]).inv ≠ msg(dummy)
       ∧ head(buffer[Node1]).time < clock[[1,Node1]]
       ∧ head(buffer[Node1]).time < clock[[2,Node1]]
       ∧ head(buffer[Node1]).time < clock[[3,Node1]]
       ∧ head(buffer[Node1]).time < clock[[4,Node1]]
       ∧ head(buffer[Node1]).time < clock[[5,Node1]] then
       if Counter[Node1] = MaxCounter
          then fire internal perform_first(Node1)
          else fire internal perform_other(Node1)
       fi
    fi;

    if randomBool ∧ resp[Node1] ≠ msg(nil) then fire output respond(resp[Node1], Node1)
    fi;

    Node1 := Node1 + 1
   od;

   Node1 := 1;
   Node2 := 1;
   while Node1 ≤ 5 do
    while Node2 ≤ 5 do
     if randomNat(1, 5) ≠ 1
        ∧ len(channel[[Node1, Node2]]) ≠ 0 ∧ Node1 ≠ Node2
        then fire internal receive(Node1, Node2)
     fi;
     Node2 := Node2 + 1
    od;
    Node2 := 1;
    Node1 := Node1 + 1
   od
 od
od
```

Figure 8: Scheduling block of Replicated State Machine

```
schedule
states
die: Nat,
Node: Nat,
read_write: Bool,
value: Nat
do
  while true do
     die := randomNat(1,2);
     Node := randomNat(1,5);
     read_write := randomBool;
     value := randomNat(1,100);
     if die = 1 ∧ req[Node] = msg(nil) then
        if read_write then fire input invoke(msg(read), Node)
   else fire input invoke(val(value), Node)
fi
     elseif die = 2 ∧ resp[Node] ≠ msg(nil) then
        fire output respond(resp[Node], Node)
     elseif req[Node] ≠ msg(nil) ∧ resp[Node] = msg(nil)
            then fire internal perform(Node)
     fi
  od
od
```

Figure 9: Scheduling block of Central Memory

```
type Message = enumeration of read, write, OK, dummy, nil
type Invocation = union of val: Nat, msg: Message
uses TimedInvocation(Invocation)
uses PriorityMinQueue(TimedInvocation[Invocation])
type Index = tuple of first: Nat, second: Nat

automaton RSM

signature
 input invoke(v: Invocation, n: Nat)
 output respond(r: Invocation, n: Nat)
 internal receive(i: Nat, j: Nat), perform_first(n: Nat),
          perform_other(n: Nat), perform_dummy(n: Nat),
  broadcast_dummy(n: Nat), InitializeClock

states
  mem: Array[Nat, Nat] := constant(0),
  resp: Array[Nat, Invocation] := constant(msg(nil)),
  clock: Array[Index, Nat] := constant(0),
  channel: Array[Index, Seq[TimedInvocation[Invocation]]] := constant({}),
  buffer:  Array[Nat, PMQ[TimedInvocation[Invocation]]] := constant(empty),
  % history variables%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  Recent: Nat := 0,
  PerformNode: Nat := 0,
  History: Array[Nat, Seq[TimedInvocation[Invocation]]] := constant({}),
  LatestResp: Array[Nat, Invocation] := constant(msg(nil)),
  Counter: Array[Nat, Nat] := constant(0),
  MaxCounter: Nat := 0,
  v: TimedInvocation[Invocation]
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

transitions
 input invoke(v, n)
  eff clock[[n,n]] := clock[[n,n]] + 1;
     buffer[n] := add(make(v, n, clock[[n,n]]), buffer[n]);
     channel[[n,1]] := channel[[n,1]] ⊢ make(v, n, clock[[n,n]]);
     channel[[n,2]] := channel[[n,2]] ⊢ make(v, n, clock[[n,n]]);
     channel[[n,3]] := channel[[n,3]] ⊢ make(v, n, clock[[n,n]]);
     channel[[n,4]] := channel[[n,4]] ⊢ make(v, n, clock[[n,n]]);
     channel[[n,5]] := channel[[n,5]] ⊢ make(v, n, clock[[n,n]])

 output respond(r, n)
  pre resp[n] = r ∧ resp[n] ≠ msg(nil)
  eff req[n] := msg(nil);
     resp[n] := msg(nil)

 internal receive(i, j)
  pre len(channel[[i,j]]) ≠ 0 ∧ i ≠ j
  eff clock[[j,j]] := max(clock[[j,j]], head(channel[[i,j]]).time);
     clock[[i,j]] := head(channel[[i,j]]).time;
     buffer[j] := add(head(channel[[i,j]]), buffer[j]);
     channel[[i,j]] := tail(channel[[i,j]])
```

Figure 10: IOA Description of Replicated State Machine for the Simulator (Part 1)

```
internal perform_first(n)
 pre ¬ isEmpty(buffer[n])
    ∧ Counter[n] = MaxCounter
    ∧ head(buffer[n]).inv ≠ msg(dummy)
    ∧ head(buffer[n]).time < clock[[1,n]]
    ∧ head(buffer[n]).time < clock[[2,n]]
    ∧ head(buffer[n]).time < clock[[3,n]]
    ∧ head(buffer[n]).time < clock[[4,n]]
    ∧ head(buffer[n]).time < clock[[5,n]]
 eff % history variables %%%%%%%%%%%%%%%%%%%%%%
   v := head(buffer[n]);
   Counter[n] := Counter[n] + 1;
   MaxCounter := max(MaxCounter, Counter[n]);
   Recent := n;
   PerformNode := v.node;
   History[n] := History[n] ⊢ v;
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   if v.node = n then
      if v.inv = msg(read)
         then resp[n] := val(mem[n])
         else mem[n] := (v.inv).val; resp[n] := msg(OK)
      fi
   else
      if v.inv = msg(read)
         then LatestResp[v.node] := val(mem[n])
         else mem[n] := (v.inv).val; LatestResp[v.node] := msg(OK)
      fi
   fi;
   buffer[n] := tail(buffer[n])

internal broadcast_dummy(n)
 eff clock[[n,n]] := clock[[n,n]] + 1;
    buffer[n] := add(make(msg(dummy), n, clock[[n,n]]), buffer[n]);
    channel[[n,1]] := channel[[n,1]] ⊢ make(msg(dummy), n, clock[[n,n]]);
    channel[[n,2]] := channel[[n,2]] ⊢ make(msg(dummy), n, clock[[n,n]]);
    channel[[n,3]] := channel[[n,3]] ⊢ make(msg(dummy), n, clock[[n,n]]);
    channel[[n,4]] := channel[[n,4]] ⊢ make(msg(dummy), n, clock[[n,n]]);
    channel[[n,5]] := channel[[n,5]] ⊢ make(msg(dummy), n, clock[[n,n]])

internal InitializeClock
 eff clock[[1,1]] := choose det do yield randomNat(1,50) od;
    clock[[2,2]] := choose det do yield randomNat(1,50) od;
    clock[[3,3]] := choose det do yield randomNat(1,50) od;
    clock[[4,4]] := choose det do yield randomNat(1,50) od;
    clock[[5,5]] := choose det do yield randomNat(1,50) od
```

Figure 11: IOA Description of Replicated State Machine for the Simulator (Part 2)

```
internal perform_other(n)
 pre ¬ isEmpty(buffer[n])
    ∧ Counter[n] < MaxCounter
    ∧ head(buffer[n]).inv ≠ msg(dummy)
    ∧ head(buffer[n]).time < clock[[1,n]]
    ∧ head(buffer[n]).time < clock[[2,n]]
    ∧ head(buffer[n]).time < clock[[3,n]]
    ∧ head(buffer[n]).time < clock[[4,n]]
    ∧ head(buffer[n]).time < clock[[5,n]]
 eff % history variables %%%%%%%%%%%%%%%%%%%%%%%
   v := head(buffer[n]);
   Counter[n] := Counter[n] + 1;
   MaxCounter := max(MaxCounter, Counter[n]);
   History[n] := History[n] ⊢ v;
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   if v.node = n then
      if v.inv = msg(read)
         then resp[n] := val(mem[n])
         else mem[n] := (v.inv).val; resp[n] := msg(OK)
      fi
   else
      if v.inv ≠ msg(read)
         then mem[n] := (v.inv).val
      fi
   fi;
   buffer[n] := tail(buffer[n])

internal perform_dummy(n)
 pre ¬ isEmpty(buffer[n]) ∧ head(buffer[n]).inv = msg(dummy)
 eff buffer[n] := tail(buffer[n])
```

Figure 12: IOA Description of Replicated State Machine for the Simulator (Part 3)

```
forward simulation from RSM to CentralMem :
   RSM.req = CentralMem.req
∧ RSM.mem[RSM.Recent] = CentralMem.mem
% ∀ n: Nat (RSM.resp[n] = CentralMem.resp[n]
%        ∨ (RSM.resp[n] = msg(nil) ∧ CentralMem.resp[n] = RSM.LatestResp[n]))
∧ (RSM.resp[1] = CentralMem.resp[1]
    ∨ (RSM.resp[1] = msg(nil) ∧ (CentralMem.resp[1] = RSM.LatestResp[1])))
∧ (RSM.resp[2] = CentralMem.resp[2]
    ∨ (RSM.resp[2] = msg(nil) ∧ (CentralMem.resp[2] = RSM.LatestResp[2])))
∧ (RSM.resp[3] = CentralMem.resp[3]
    ∨ (RSM.resp[3] = msg(nil) ∧ (CentralMem.resp[3] = RSM.LatestResp[3])))
∧ (RSM.resp[4] = CentralMem.resp[4]
    ∨ (RSM.resp[4] = msg(nil) ∧ (CentralMem.resp[4] = RSM.LatestResp[4])))
∧ (RSM.resp[5] = CentralMem.resp[5]
    ∨ (RSM.resp[5] = msg(nil) ∧ (CentralMem.resp[5] = RSM.LatestResp[5])))
proof
  initially
     CentralMem.mem  := 0;
     CentralMem.req  := constant(msg(nil));
     CentralMem.resp := constant(msg(nil))
  for input invoke(v: Invocation, n: Nat)
      do fire input invoke(v, n) od
  for output respond(r: Invocation, n: Nat)
      do fire output respond(r, n) od
  for internal perform_first(n: Nat)
      do fire internal perform(RSM.PerformNode) od
  for internal perform_other(n: Nat)
      ignore
  for internal Initialize
      ignore
  for internal receive(i: Nat, j: Nat)
      ignore
  for internal broadcast_dummy(n: Nat)
      ignore
  for internal perform_dummy(n: Nat)
      ignore
```

Figure 13: Simulation Relation and Step Correspondence between RSM and CentralMem

```
Automaton initialized
1: internal InitializeClock in automaton RSM
2: input invoke(val(8), 1) in automaton RSM
2: input invoke(val(8), 1) in automaton CentralMem
3: input invoke(msg(read), 2) in automaton RSM
3: input invoke(msg(read), 2) in automaton CentralMem
4: input invoke(msg(read), 3) in automaton RSM
5: input invoke(val(100), 4) in automaton RSM
5: input invoke(val(100), 4) in automaton CentralMem
6: input invoke(msg(read), 5) in automaton RSM
6: input invoke(msg(read), 5) in automaton CentralMem
7: internal receive(1, 2) in automaton RSM
8: internal receive(1, 3) in automaton RSM
9: internal receive(1, 4) in automaton RSM
10: internal receive(1, 5) in automaton RSM
11: internal receive(2, 1) in automaton RSM
12: internal receive(2, 3) in automaton RSM
13: internal receive(2, 4) in automaton RSM
14: internal receive(2, 5) in automaton RSM
15: internal receive(3, 1) in automaton RSM
16: internal receive(3, 2) in automaton RSM
17: internal receive(3, 5) in automaton RSM
18: internal receive(4, 1) in automaton RSM
19: internal receive(4, 2) in automaton RSM
20: internal receive(4, 3) in automaton RSM
...
41: internal perform_first(4) in automaton RSM
41: internal perform(4) in automaton CentralMem
42: internal receive(2, 4) in automaton RSM
43: internal receive(2, 5) in automaton RSM
44: internal broadcast_dummy(2) in automaton RSM
45: internal perform_first(4) in automaton RSM
45: internal perform(2) in automaton CentralMem
...
495: internal receive(5, 3) in automaton RSM
496: internal perform_dummy(1) in automaton RSM
497: input invoke(msg(read), 3) in automaton RSM
497: input invoke(msg(read), 3) in automaton CentralMem
498: internal perform_other(5) in automaton RSM
499: internal receive(3, 1) in automaton RSM
500: internal receive(3, 2) in automaton RSM
>>>> No errors
```

Figure 14: Paired Simulator Output of RSM implementing CentralMem

5 Future Improvements in the IOA Simulator

While the IOA Simulator is capable of checking correctness and quickly detecting errors in IOA codes, it still suffers from not being able to recognize a lot of IOA syntax. Furthermore, the scheduling block is generally difficult to code especially if the precondition of a transition is very long.

In this section, we suggest four improvements that can be made to the IOA Simulator. In particular, we also suggest ways of implementing it.

5.1 Support of Existential and Universal Quantifiers

Although IOA supports syntax like ∀ `x: Nat P(x)`, the simulator cannot support this for two reasons:

- The test set[10] is **infinite**. In the example above, we could not possibly compute in finite time $P(x)$ for all the values of $x \in N$.
- Even if the test case is not finite, it is generally very difficult to actually determine the test set. For example, consider an IOA statement like this:

  ∀ `x: L P(x)` **where** `x ∈ SAT ∧ length(x) ≤ 10`

  To actually compute the values of x satisfying the **where** takes exponential(a very long) time and is therefore impractical for most purposes.

To solve this problem, we suggest adding a `det` block after the **where** clause. The `det` block, first suggested in [9], is a way to tell the Simulator to handle non-deterministic actions. While evaluating the value of a universal or existential quantifier is deterministic, `det` proved useful as they can actually tell the Simulator what values of x should it evaluate.

Adding a `det` would cause universal and existential quantifiers to be written in this general scheme:

```
∀ x: X P(x) where Q(x) det
do
  #tell the simulator the values of x where P(x) should be evaluated#
od
```

In the scheme above, X is the data type of x and if $Q(x)$ is true, $P(x)$ is evaluated. Therefore, the logical equivalence of the above statement is:

$$\bigwedge_{x \in X} (P(x) \vee \overline{Q(x)})$$

Of course, if ∀ is replaced with ∃, then the logical equivalence would be:

$$\bigvee_{x \in X} (P(x) \vee \overline{Q(x)})$$

With this improvement, we can write statements like:

```
∀ n: Nat buffer[n] := buffer[n] ⊢ v where 1≤n ∧ n ≤ 10 det
do
  while i ≤ 10
  do
   yield i;
   i := i + 1;
  od
od
```

For more detailed information about `det` blocks, the reader is referred to [9].

Different approaches to solving the existential and universal quantifiers support problem can be found in [5].

[10]By test set, we mean that the set of values that we want to test over.

5.2 Eliminate the Preconditions in Scheduling Block

Currently, to ensure that the scheduling block fires[11] a legal transition, we would have to write preconditions of that transition into the scheduling block. However, since the precondition of a transition is already specified in the IOA code, the precondition in the scheduling block is superfluous. A small modification in the simulator will allow statements like

```
if performDummy ∧ buffer[[Node1, Node2]] ≠ {}
   ∧ head(buffer[[Node1, Node2]]).inv ≠ msg(dummy)
  then fire internal perform_dummy(Node1, Node2)
fi;
```

to be rewritten as

```
fire internal perform_dummy(Node1, Node2) subject to precondition
```

This means that the simulator will test whether the precondition of the internal `perform_dummy` transition holds, and will fire the transition if and only if the precondition holds. Obviously, this improvement would make the scheduling block easier to write and less prone to mistakes.

5.3 Automatically Detecting Valid Traces

The main difficulty in proving simulation relations between two given automata is to provide a good and correct step correspondence. This proposal is an ambitious one. We want to the Simulator to automatically detect whether a given trace[12] is valid for an automaton or not. If the Simulator is capable of doing so, we can test whether it is likely to be true that an automaton A implements another automaton B. We first generate random traces of automaton A and test for their validity as a trace of automaton B. If a trace of automaton A is not a valid trace of automaton B, then clearly automaton A does not implement automaton B. However, if all the traces of A that was tested by the simulator is valid trace of B, then this indicates that automaton A may be an implementation of automaton B (note that a simulator can never prove implementation as the set of all traces is infinite).

Nevertheless, a simulator cannot, in general, prove that a trace of A is not a valid trace of B. Proving a trace σ of A does not belong to B involves showing all the executions[13] of B can never produce σ. Since the set of all executions is infinite (in general), determining whether validity of a trace σ of A with respect to B is impossible.

So, how can we prove that a trace σ is a valid trace of automaton B? We just need to produce an execution of B which has the same trace as σ. To do so, our proposed algorithm starts with the inputs and outputs transitions of σ and try to insert additional internal transitions so that the preconditions of each transition holds with respect to B.

For example, let us consider the automata Central Memory and Strong Cache. Since Central Memory implements Strong Cache [3], we expect that a trace of Central Memory would be a valid trace of Strong Cache too. An example of a trace of Central Memory is as follows:

invoke(val(10), 2), **invoke(msg(read), 5)**, perform(2), perform(5), **respond(msg(OK), 2)**, **respond(val(10), 5)**

The corresponding execution of StrongCache that would lead to the same trace is:

invoke(val(10), 2), **invoke(msg(read), 5)**, drop(2), perform(2), copy(2), drop(5), perform(5), copy(5), **respond(msg(OK), 2)**, **respond(val(10), 5)**

Notice that to obtain a valid execution of Strong Cache, we first need to take the given trace of Central Memory(external transitions/trace are shown in bold) and add internal transitions to ensure that the precondition of every transition in the execution holds in Strong Cache.

[11] Firing a transition also means executing a transition.

[12] A trace is a sequence of external transitions.

[13] Unlike a trace, an execution is a sequence of all transitions (internal and external).

An effective method to add internal transitions in not known. We could, in theory, do a exhaustive search in including internal transitions; first adding one internal transitions, if that fails, then add two internal transitions and so forth. However, this method of guessing takes exponential time and it not practical for most purposes.

5.4 Making Fair Transitions

This is another difficult improvement. We want to simulator to fire transitions randomly but *randomly* must first be defined. A good definition of making a *fair*(random) transition is picking at random one of the n possible transitions at a given state of the automaton. Of course, we may also want to give different weightage to each transition.

First, we define an *essential* parameter to be a parameter of a transition whose changes may affect the enabled state of the transition. A *non-essential* parameter is a parameter that is not *essential*. For example, in the input transition `invoke(v, n)`, n is an *essential* parameter whereas v is a *non-essential* parameter since if `invoke(v1, n)` is enabled if and only if `invoke(v2, n)` is enabled for all valid invocations $v1$, $v2$. However, `invoke(v, n1)` may be enabled without having `invoke(v, n2)` to be enabled.

Then with these definitions in place, the Simulator can count the number of possible transitions by looping over all the *essential* parameters of each transitions. With the weightage associated to each transition, the Simulator can choose using a pseudo-random number generator, a transition to be fired. The *non-essential* parameters will be chosen at random from a user-specified set. For example, assume that the Simulator chooses to fire `invoke(v, 5)` and that the user-specified set for the values of parameter v is {read, write(1), write(2), ..., write(10)}. The Simulator will pick at random one of the values of the parameter v and fire the transition.

Nevertheless, in general it is difficult for the user to specify the set of possible values for all the *essential* and *non-essential* parameters. Currently the best approach to have randomized transitions is to program a good scheduling block. Hopefully, in the future, a *smart* Simulator would be able to handle the firing of transitions with minimal user input.

6 Acknowledgements

I would like to extend my gratitude to Professor Nancy Lynch and Dr. Steve Garland for giving me the opportunity to conduct this research under the Undergraduate Research Opportunities Program 2001.

This work would not have been possible without the help and insight of Andrej Bogdanov who introduced to the Replicated State Machine implementing Central Memory problem. Laura Dean, Michael Tsai and Toh Ne Win helped a great deal in improving the IOA Simulator by reacting quickly to my suggestions and offering advice on the Simulator.

Last but not least, I would have to thank all the people whom I have worked with during the summer of 2001 for giving me the support and encouragement necessary in pursuing this research.

References

[1] Christine Karlovich Andrej Bogdanov, Laura Dean. Distributed memory algorithms coded in IOA: Challenge problems for software analysis and synthesis methods. January 2001.

[2] Dimitris Vyzovitis Andrej Bogdanov, Joshua Tauber. Formal verification of a replicated data management algorithm. May 2001.

[3] Andrej Bogdanov. Formal verification of simulation between I/O automaton. Master's thesis, Massachusetts Institute of Technology, 2001.

[4] Anna E. Chefter. A simulator for the IOA language. Master's thesis, Massachusetts Institute of Technology, 1998.

[5] Laura Dean. Improved simulation of Input/Output automata. Master's thesis, Massachusetts Institute of Technology, 2001.

[6] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1998.

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

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

[9] J. Antonio Ramirez-Robredo. Paired simulation of I/O automata. Master's thesis, Massachusetts Institute of Technology, 2000.

[10] Mandana Vaziri Stephen J. Garland, Nancy A. Lynch. *IOA: A Language for Specifying, Programming, and Validating Distributed Systems*, December 2000.