Correctness Proof for a Distributed Memory System

Vicente Cholvi-Juan[1] and José M. Bernabéu-Aubán[2]

[1] University Jaume I
Castelló (Spain)
[2] Polytechnic University of Valencia
Valencia (Spain)

Abstract. Distributed shared memory systems offer the possibility of implementing parallel algorithms using the familiar operations of shared memory access. In this paper we present the proof of correctness of an algorithm implementing such a shared memory system over a distributed environment. Such proof is based in the formal model for memory systems introduced by the authors in previous works.

1 Introduction

A typical (loosely coupled) distributed systems is composed of a collection of independent computers interconnected through some type of network. In order to cooperate, applications written to span several computers on such a system need to have some mechanism to allow each one of their parts to exchange information.

The *shared memory model* (SMM) provides a shared address space which can be used by processes in the same way as local memory, even if they are executed concurrently in different processors. Thus, every process can access any address by means of the basic operations $data = read(address)$ and $write(address, data)$, where *read* returns the *data* in *address*, and *write* associates *data* with *address*.

When a SMM is built on top of a distributed system, we get what is known as a DSMM. In this paper we present the formal proof of correctness of a DSMM algorithm which implements a particular shared memory model. This model is a mixture of the *atomic* memory model [3] and the *sequential* model [4], thus presenting the developer with a familiar memory access semantics, while allowing a more efficient implementation. The proof is carried out using the formal model introduced by the authors in [1]. We base our framework on the I/O automata formalism [5] for discrete event systems. The I/O automata formalism has been successfully used to prove the correctness of algorithms as well as to specify properties of memory systems and transactional systems.

2 Correctness Proof

It is well known the inherent complexity of distributed algorithms. Indeed, there are several widely known algorithms that were considered correctly designed and

that later have been found incorrect. Therefore, it is very important to be able to ensure that implementations of models work correctly.

Most of the work on shared models has not paid enough attention to the way such models are specified. The availability of an adequate formal framework makes it possible to answer questions about models and algorithms implementing them. In another work [2], we have introduced such a formal framework. Here, we show, basing on the specifications of two given models, how to we verify the correctness of a proposed DSMM implementing algorithm.

Our framework takes into account two type of operations: $\mathsf{write}(i, x, u)$ which associates the value u to the variable x, and $\mathsf{read}(i, x, u)$ which informs of the value u associated with the variable x (being i the process that executes the operation). The way these operations are executed are described as actions of that formalism. A $\mathsf{write}(i, x, u)$ operation can be identified as a pair of $\mathsf{bwrite}(i, x, u) - \mathsf{ewrite}(i, x)$ events, and a $\mathsf{read}(i, x, u)$ operation can be identified as a pair of $\mathsf{bread}(i, x) - \mathsf{eread}(i, x, u)$ events (the first to begin the operation and the second to finish it).

In our formalism to specify a given memory model it is enough to specify the set of executions it allows. Moreover, the sequences of the model that we are going to implement (a mixture of the atomic and sequential) are characterized by the existence of atomics and sequential "views" of those sequences. Formally, from [2], "a sequence α is *sequential* if $VIEWS(SEQ, \alpha)$ is not an empty set (ditto for the *atomic* sequences)".

The proposed DSMM implementing algorithm permits to take advantage of the larger degree of parallelism given by the sequential model, allowing at the same time the use of a stricter model (atomic) for a subset of the variables.

The way in which the DSMM implementing algorithm deals with atomic variables consists of invalidating all the replicas previously to each writing. Thus, each variable in isolation will be atomic [3]. Thereby, if we consider that the atomic model is compositional, we can assume (in the verification) that all the variables are sequential without losing generality.

Our solution contains three system's descriptions: OpSpec, the operational requirement specification; SysImpl, the hight level system implementation; and DiscSysImpl, the discrete system implementation.

The operational specification describes the system by using a single automaton. Next, we split it into two automata: one of them models the architectural platform and the other the DSMM implementing algorithm. Finally, we split the later automaton into the automata that model its distributed implementation.

Here, we present only those parts we consider necessary to understand the verification process. The details of the proofs are available in [2]. In what follows, $\mathcal{P}$ will stand for the set of processes, $\mathcal{M}$ for the set of variables (being $\mathcal{M}_0$ the initial values) and $msig(\mathcal{P}, \mathcal{M})$ for the set of memory operations.

2.1 Operational Specification

The operational specification, OpSpec, consists of a single automaton, $BLACK(\mathcal{P}, \mathcal{M}, \mathcal{M}_0)$ (Fig. 1), which models all required properties in order to

solve the sequential model. Moreover, it incorporates the fairness condition that guarantees liveness of the implementation. The state transitions are described by specifying the "preconditions" under which each action can occur and the "effect" of each action. This constitutes an abstract representation harder to understand than the axiomatic specification of the sequential model. However, it is easier to use in the proofs.

Actions:
- Inputs: $\mathsf{bread}(i, x)$, $\mathsf{bwrite}(i, x, u)$
- Outputs: $\mathsf{eread}(i, x, u)$, $\mathsf{ewrite}(i, x)$

where $i \in \mathcal{P}$, $x \in \mathcal{M}$ and $u \in range(x)$

State: a set of elements, S, in $msig(\mathcal{P}, \mathcal{M})$, initially empty

Transitions:

- $\mathsf{bwrite}(i, x, u)$

  Effect:
  $S = S \cdot \mathsf{bwrite}(i, x, u)$
- $\mathsf{bread}(i, x)$

  Effect:
  $S = S \cdot \mathsf{bread}(i, x)$
- $\mathsf{ewrite}(i, x)$

  Precondition:
  $last(S \mid i) = \mathsf{bwrite}(i, x, \cdot)$

  Effect:
  $S = S \cdot \mathsf{ewrite}(i, x)$
- $\mathsf{eread}(i, x, u)$

  Precondition:
  $last(S \mid i) = \mathsf{bread}(i, x)$
  $VIEW(SEQ, T) \neq \emptyset$
  (where $T = S \cdot \mathsf{eread}(i, x, u)$)

  Effect:
  $S = S \cdot \mathsf{eread}(i, x, u)$

Partition: $\{\{\mathsf{ewrite}(i, \cdot)\}, \{\mathsf{eread}(i, \cdot, \cdot)\}\}$

Fig. 1. Specification of $BLACK(\mathcal{P}, \mathcal{M}, \mathcal{M}_0)$.

2.2 System Implementation

To implement a model, memory operations have to be translated into operations of the underlying system. The nature of such operations depends on the architecture of the system on which the model is to be implemented. The architecture gives us only part of the full implementation of the model. The other part is the algorithm which, based on the architecture, implements the model (MMS).

In this section we provide the specification of the components that compose the system by using two automata. The first implements the architecture and the other one the MMS.

We consider a distributed architecture consisting of a set of locations $\mathcal{D}$ connected by a fault-free, ordered communication channel. We identify the set of variables at each location as $\mathcal{W}$, being $\mathcal{W}_0$ their initial values. Then, the architecture is globally modeled by using a single automaton $DST(\mathcal{D}, \mathcal{W}, \mathcal{W}_0)$

(Fig. 2). Roughly speaking, each memory operation is modeled by using three actions (the first to start it, the second to "physically perform" it and the later to finish it). Moreover, data at each location is modeled by an array of storing values. Accesses to those data are made through three FIFO operations: $HEAD$ returns the queue's bottom value, DEQ dequeues that value and ENQ puts the value on the top of the queue.

Actions:

- Inputs: $\mathsf{br}(j, y)$, $\mathsf{bw}(j, y, v)$
- Outputs: $\mathsf{er}(j, y, v)$, $\mathsf{ew}(j, y)$
- Internals: $\mathsf{xr}(j, y, v)$, $\mathsf{xw}(j, y, v)$

where $j \in \mathcal{D}$, $y \in \mathcal{W}$ and $v \in range(y)$

State:

- $F_{in}[j]$, $F_{out}[j]$: array of FIFO queues of elements in $in(sig(DST(\mathcal{D}, \mathcal{W}, \mathcal{W}_0)))$ and $out(sig(DST(\mathcal{D}, \mathcal{W}, \mathcal{W}_0)))$ respectively, initially empties
- Store[j,y]: array of variables in the range of y, initially y_0

Transitions:

- $\mathsf{bw}(j, y, v)$
  Effect:
  $ENQ(F_{in}[j], \mathsf{bw}(j, y, v))$
- $\mathsf{ew}(j, y)$
  Precondition:
  $HEAD(F_{out}[j]) = \mathsf{ew}(j, y)$
  Effect:
  $DEQ(F_{out}[y])$
- $\mathsf{xw}(j, y, v)$
  Precondition:
  $HEAD(F_{in}[j]) = \mathsf{bw}(j, y, v)$
  Effect:
  $Store[j, y] = v$
  $DEQ(F_{in}[y])$
  $ENQ(F_{out}[y], \mathsf{ew}(j, y))$
- $\mathsf{br}(j, y)$
  Effect:
  $ENQ(F_{in}[j], \mathsf{br}(j, y))$
- $\mathsf{er}(j, y, v)$
  Precondition:
  $HEAD(F_{out}[j]) = \mathsf{er}(j, y, v)$
  Effect:
  $DEQ(F_{out}[j])$
- $\mathsf{xr}(j, y, v)$
  Precondition:
  $HEAD(F_{in}[y]) = \mathsf{br}(j, y)$
  $v = Store[j, y]$
  Effect:
  $DEQ(F_{in}[j])$
  $ENQ(F_{out}[j], \mathsf{er}(j, y, v))$

Partition: $\{\{\mathsf{ew}(j, \cdot)\}, \{\mathsf{er}(j, \cdot, \cdot)\}, \{\mathsf{xw}(j, \cdot, \cdot)\}, \{\mathsf{xr}(j, \cdot, \cdot)\}\}$

Fig. 2. Specification of $DST(\mathcal{D}, \mathcal{W}, \mathcal{W}_0)$.

The MMS is modeled by using a single automaton $RED(\mathcal{P}, \mathcal{M})$ (Fig. 3). That continues being a rather abstract representation of the system which is still far away from a realistic representation of the proposed distributed memory system. However, it is intended to be used in the discrete system specification correctness proof.

At this stage it has to be proved that the interaction of the MMS with the architecture provides sequential executions.

Lemma 1. *Let* SysImpl $= hide_{\Sigma}(RED(\mathcal{P}, \mathcal{M}) \; \Pi \; DST(\mathcal{D}, \mathcal{W}, \mathcal{W}_0))$ *(where* $\Sigma = extsig(DST(\mathcal{D}, \mathcal{W}, \mathcal{W}_0))$*). Then* SysImpl *solves* OpSpec.

Actions:
- Inputs: $\mathsf{bread}(i,x)$, $\mathsf{bwrite}(i,x,u)$, $\mathsf{er}(j,y,v)$, $\mathsf{ew}(j,y)$
- Outputs: $\mathsf{eread}(i,x,u)$, $\mathsf{ewrite}(i,x)$, $\mathsf{br}(j,y)$, $\mathsf{bw}(j,y,v)$

where $i \in \mathcal{P}$, $x \in \mathcal{M}$, $u \in range(x)$, $j \in \mathcal{D}$, $y \in \mathcal{W}$ and $v \in range(y)$

State: a set of elements, S, in $sig(RED(\mathcal{P},\mathcal{M}))$, initially empty

Transitions:

- $\mathsf{bwrite}(i,x,u)$
  Effect:
  $S = S \cdot \mathsf{bwrite}(i,x,u)$
- $\mathsf{ewrite}(i,x)$
  Precondition:
  $last(S \mid i) = \mathsf{ew}(i,x)$
  Effect:
  $S = S \cdot \mathsf{ewrite}(i,x)$
- $\mathsf{bread}(i,x)$
  Effect:
  $S = S \cdot \mathsf{bread}(i,x)$
- $\mathsf{eread}(i,x,u)$
  Precondition:
  $last(S \mid i) = \mathsf{er}(i,x,u)$
  $VIEW(SEQ,T) \neq \emptyset$
  (where $T = (S \cdot \mathsf{eread}(i,x,u)) \mid msig(\mathcal{P},\mathcal{M})$)
  Effect:
  $S = S \cdot \mathsf{eread}(i,x,u)$
- $\mathsf{bw}(j,y,v)$
  Precondition:
  $last(S \mid j) = \mathsf{bwrite}(j,y,v)$
  Effect:
  $S = S \cdot \mathsf{bw}(j,y,v)$
- $\mathsf{ew}(j,y)$
  Effect:
  $S = S \cdot \mathsf{ew}(j,y)$
- $\mathsf{br}(j,y)$
  Precondition:
  $last(S \mid j) = \mathsf{bread}(j,y)$
  Effect:
  $S = S \cdot \mathsf{br}(j,y)$
- $\mathsf{er}(j,y,v)$
  Effect:
  $S = S \cdot \mathsf{er}(j,y,v)$

Partition: $\{\{\mathsf{eread}(i,\cdot,\cdot)\},\{\mathsf{ewrite}(i,\cdot)\},\{\mathsf{br}(j,\cdot)\},\{\mathsf{bw}(j,\cdot,\cdot)\}\}$

Fig. 3. Specification of $RED(\mathcal{P},\mathcal{M})$.

2.3 Discrete System Implementation

In the previous sections we have presented two levels of description of our solution, which, by their very nature, are quite abstract and do not correspond directly with an implementation over a distributed system.

We now proceed to describe the discrete system model of the algorithm. In this model, the distributed nature of the underlying system is made explicit by using FIFO communication channels and explicit message passing between nodes in the distributed system.

The communications system is modeled by an automaton $FIFO_M$, which, for our purposes, is in charge of sending and making receive the messages used by our protocol.

To obtain DiscSysImpl we compose the automata that will model the modules responsible of requesting variables as well as providing them ($\{WRKR_i(\mathcal{P},\mathcal{M})\}_{i\in\mathcal{P}}$) with the module taking care of arbitrating conflicting requests for variables, ordering them and keeping track of circulating variables ($SEQ(\mathcal{P},\mathcal{M})$) and that which models the *communication channel* module ($FIFO_M$).

At this stage it has to be proved that the interaction of the automata that model the discrete system implementation can be safely used for any task for which $RED(\mathcal{P},\mathcal{M})$ is satisfactory.

Lemma 2.
Let DiscSysImpl $= hide_{\Sigma}(FIFO_M \ \Pi \ SEQ(\mathcal{P},\mathcal{M}) \ \Pi \ \{WRKR_i(\mathcal{P},\mathcal{M})\}_{i\in\mathcal{P}})$ *(where* $\Sigma = extsig(FIFO_M)$*). Then* DiscSysImpl *solves* $RED(\mathcal{P},\mathcal{M})$.

3 Conclusion

In this paper we have shown how to model, using our formalization of memory coherency models, a particular algorithm implementing a distributed shared memory system. First we modeled the problem using an I/O automata. The possible execution sequences of this automata define the "problem" we claim our algorithm "solves". Then we propose a more detailed I/O automaton which is shown to solve the original problem. This automaton, while still abstract, and not making use of the distributed nature of the underlying system, contains some details which facilitate its further break down into a fully distributed specification.

The method used in this paper would be appropriate in general to prove the correctness of any other MMS implementation. In our approach we use the full power of the I/O automata formalism. A similar approach can also be used to prove the correctness of algorithms built upon any particular memory model.

References

1. Bernabéu-Aubán, J.M., Cholvi-Juan, V.: Formalizing memory coherency models. Journal of Computing and Information **1** (1994) 653–672
2. Cholvi-Juan, V.: Formalizing memory models. PhD thesis, Department of Computer Science, Polythechnic University of Valencia (1994)
3. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects **12** 3 (1990) 463–492
4. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Transactions on Computers **28** 9 (1979) 690–691
5. Lynch, N.: I/O Automata: A model for discrete event system. Massachusetts Institute of Technology, Technical Report MIT/LCS/TM-351 (1988)