Formal Verification of Complex Coherence Protocols Using Symbolic State Models

Fong Pong and Michel Dubois

CENG Technical Report 94-01

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4475

January 1994

Formal Verification of Complex Coherence Protocols Using Symbolic State Models

Fong Pong and Michel Dubois

Department of EE-Systems
University of Southern California
Los Angeles, CA90089-2562
pong@paris.usc.edu (213) 740-9130

January 1994

Abstract

Directory-based coherence protocols are so complex that verification techniques based on systematic procedures are required to establish their correctness. In this study, we provide a complete design of a write-invalidate, full-map directory-based coherence protocol for non-FIFO interconnection networks --i.e. networks in which the order of messages between two nodes is not preserved from source to destination. To promote efficiency, the protocol avoids the acknowledgment of every message exchanged between caches and main memory.

We also develop the concepts and notations to verify the protocol with a symbolic state model (SSM). We compare SSM with the Stanford Murφ system for the purpose of verifying the protocol, and show that SSM is more efficient in terms of verification time and memory consumption and therefore holds the promise of verifying much more complex protocols. A unique feature of SSM is that it verifies protocols for any system size and therefore provides reliable verification results in one run of the tool.

1 Introduction

Caching data close to the processor dynamically is an important technique for reducing the latency of memory accesses in a shared-memory multiprocessor system. However, because multiple copies of the same memory block may exist in a system with private caches, a cache coherence protocol is required to maintain coherence among all data copies [28]. A cache coherence protocol is vital to correct system operation: the protocol must provide processors with a globally consistent view of the memory and promote good system performance and scalability. As a result, the complexity of proposed protocols keeps increasing. Their complexity is hampered however by the ability to implement them correctly.

Currently *snooping* protocols [3] are widely understood and accepted. Because they rely on the broadcasting of update values or of invalidations to keep data copies consistent, their appli-

cability is inherently limited to systems with an interconnection supporting efficient broadcast, mostly shared-bus systems. Unfortunately a shared bus is non-scalable in the sense that a few processors can saturate it even in the presence of private caches.

By contrast, directory-based protocols [1, 4, 7, 17] are more appropriate for systems with very large numbers of processors. By associating every memory block with a directory entry keeping track of caches with a copy of the block, coherence messages are sent to individual processors rather than broadcast to all processors, removing the need for an efficient broadcast medium. However, efficient directory-based protocols are generally more complex. Usually the number of states and the number of types of messages exchanged is much greater in a directory than in a snooping protocol. When several coherence transactions bearing on the same block are initiated at the same time by different processors, messages may enter a race condition, from which the protocol behavior is often hard to predict [2] because the protocol designer can not visualize all possible temporal interleavings of coherence messages.

In general, a directory protocol can be greatly simplified if the interconnection network is FIFO (First In First Out) between any two nodes [17]. To illustrate why, consider the following simple case of a processor first sending the copy of a dirty block to memory on a replacement (write-back) and then, immediately afterwards, accessing the block again and sending a request for the block to memory (miss). If the network is not FIFO the memory controller may become totally confused if it receives the request for the block caused by the miss before it receives the copy of the dirty block sent at the replacement! In general this problem can be solved by making sure that a message is fully acknowledged before another message involving the same destination node and the same block is issued. This approach however may lead to a drastic increase of the memory traffic since acknowledgments must be ordered as well. This traffic caused by excessive acknowledgments unnecessarily penalizes every transaction in order to avoid problems caused by extremely rare events.

One first contribution of this paper is to provide a complete, correct design in which messages exchanged between caches and the memory are not necessarily acknowledged in systems with non-FIFO networks. Whenever there is a threat of entering erroneous states because of races among messages, the protocol synchronizes the interactions between caches and the memory by entering transient states and by initiating synchronization messages. The approach taken in this paper is to assume at first no additional acknowledgments (w.r.t. the protocol with a FIFO network) and to add them as needed after races leading to inconsistent states have been detected.

The protocol is based on the full-map scheme proposed by Censier and Feautrier [7]. It has been noted by Archibald in [2] that the specification of this protocol is incomplete in the case of general interconnection networks. One problem is that the protocol must deal with *ghost signals*. Ghost signals are caused by messages whose meaning is lost between their transmission and their reception because some other message has changed the state of the block. For example, a cache issues a request and, while the request travels to memory, the cache receives a message set-

ting the block into a new state in which the primary request is no longer valid; later on, unexpected (ghost) signals caused by that request may be received by the cache [2, 5]. The race conditions identified in [2] are rather simple and a complete design is not provided. Also, there is no evidence as to the extent to which the number of handshake messages between caches and memory can be reduced while still guaranteeing a correct design. It is not even clear that simply acknowledging every message will do. For example we could have race conditions between acknowledgment messages and other messages related to the same block.

To detect these races we use a new protocol verification technique called the *Symbolic State Model* or SSM and applied in [23] to the verification of snooping protocols at a high level of abstraction. More recently, the method was extended to directory-based protocols in [24]. The SSM method is based on reachability analysis using finite state machines to model the behavior of constituent processes in the protocol. We develop in this paper the concepts and notations needed to verify complex directory protocols with the symbolic state model (SSM). We also compare SSM with the Stanford Murφ system [11] for the purpose of verifying the protocol, and show that SSM is more efficient in terms of verification time and memory consumption and therefore holds the promise of verifying much more complex protocols. The paper is structured as follows. Section 2 describes some relevant techniques for protocol verification, including the Murφ system. Section 3 provides a detailed description of the protocol for non-FIFO networks. Correctness issues and mechanisms for detecting various types of flaws are discussed in Section 4. We then develop the methodology in Section 5 and present the results of our study in Section 6.

2 Approaches for Protocol Verification

One important class of verification techniques derives from state enumeration methods (*reachability* or *perturbation* analysis), which explore all possible system states [10, 13, 15, 29]. Generally, the expansion process starts with a given initial state, from which all possible transitions are exercised, leading to a number of new states. The same process is applied repeatedly for every new state until no new states are generated. (Some transitions may lead back to states which have already been generated.) At the end, a *global state transition diagram* or a *reachability graph* showing the transition relations among global states is reported.

The Murφ system, developed by Dill et al. [11], is based on a traditional state enumeration which explores all reachable system states and reduces the state space search by exploiting structural symmetry in the system [14]. In this respect, it is similar to the SSM method. There are two versions of Murφ: the non-symmetric Murφ system (Murφ-ns) and the symmetric Murφ system (Murφ-s) [11]. In Murφ-ns, two system states are equivalent if and only if they are identical whereas Murφ-s exploits the symmetry of the system by lumping together states whose representations are permutations of each other [14]. For example, two system states composed of three local cache states, (shared, shared, invalid) and (invalid, shared, shared), are deemed equivalent because the order of cache states in the global state representation is irrelevant to the correctness of the protocol. In general, because of the state enumeration complexity and because of the *state*

space explosion [13], verifying a system with a large number of caches becomes rapidly impractical. As protocols become more complex, it is not clear whether verifying a small-scale system model can provide a reliable error coverage for all system sizes.

To overcome the state space explosion problem, McMillan and Schwalbe [19] evaluate the truth value of protocol correctness conditions in temporal logic without constructing the global state diagram explicitly. The transition relations among global states must be constructed [20], which itself is not feasible for most complex and large-scale systems [9]. Moreover, the size of the underlying *BDD* (*binary decision diagram*) representation for the transition relations increases in proportion to the complexity and the scale of the system model. For a complex and large-scale system model, the method may lead to *BDD size explosion*. By using *higher-order* logic, Loewenstein and Dill [18] showed the *correspondence* (*simulation relation*) between state machines representing an implementation and the behavior specification at a high level of abstraction. Although the logic is powerful, finding the simulation relation is a difficult and inefficient procedure.

Methods for reasoning about protocol correctness independently of system sizes have been explored by several researchers. Browne et al. [6] verified systems with many identical finite state processes by finding a *correspondence* between two (global state) structures M and M' representing systems of manageable and overwhelming complexities respectively. A given relation E can determine whether there is a correspondence relation between two structures. If two structures correspond, formulas in temporal logic can be evaluated on the simpler *base* state machine. In practice, this approach may be difficult to apply to complex protocols. First, considerable human ingenuity is needed to discover the correspondence relation between two structures. Second, constructing the global state transition diagram of the *base* machine may be a difficult procedure itself, because the base machine capturing the essential properties of the system may still be very complex.

Recently, Kurshan and McMillan [16] suggested the use of partial-order relations among processes to form an *invariant* process. A process is invariant if the composition of the invariant process with a new process is less than or equal to the invariant. This method has great potential, but the construction of the invariant process in [16] is not automated and requires considerable ingenuity for complex protocols.

Similar to other approaches enumerating states, the SSM method used in this paper reduces the complexity of the state space search by exploiting equivalence among global states. Briefly, the equivalence relation exploited in [23, 24] is based on the observation that the behavior of all caches are characterized by the same finite state machine. Caches in the same state are combined into an equivalence class; a global state is then composed of equivalence classes. Moreover, the number of caches in a state class is abstractly represented by a set of repetition constructors indicating 0, 1, or multiple instances of caches in that class. As a result, a global state represents a family of equivalent states and can be efficiently expanded because expanding a global state is

equivalent to expanding a set of states. In addition, the SSM method provides reliable results because it verifies a protocol for any system size. In the following, the SSM approach is applied to a complex directory-based protocol, which is first specified.

3 A Directory-Based Protocol for Non-FIFO Networks

In Censier and Feautrier's write-invalidate protocol every memory block is associated with a *vector of presence bits*, each of which indicates whether a cache has a copy of the block [7]. The presence bit is set when the copy is first loaded in cache and is reset when the copy is invalidated or replaced. When multiple copies exist in different caches, they must be identical to the memory copy. We say that the copies are *clean* and *Shared*. An extra *dirty* bit per block indicates whether or not a dirty cached copy exists. In this case, there cannot be more than one cached copy and we say that the copy is *Exclusive*. The cache with the exclusive copy is also called the *Owner* of the block. To enforce ownership of blocks, invalidations must be sent to caches with their presence bits set. One major difference between our protocol and Censier and Feautrier's is that, on a replacement, we do not clear the presence bit if the replaced copy is *Shared* (clean), in order to cut the memory traffic. The trade-off is that more invalidations are sent each time a presence bit at the memory is set while the corresponding cache has no copy.

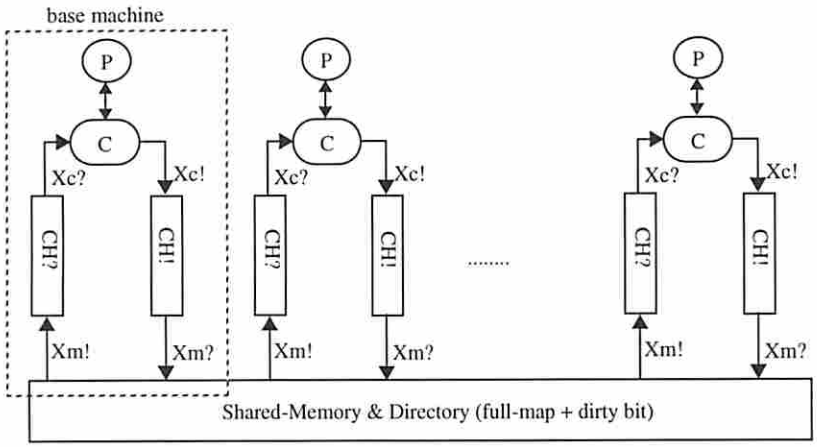


FIGURE 1. Architectural Model for Systems with a Directory-Based Coherence Protocol.

The architectural model of the system shown in Figure 1 consists of a central directory and of multiple processor-cache pairs. Each processor is associated with one message *sending channel* (*CH!*) and one message *receiving channel* (*CH?*) to model the message flow between caches and main memory. The message channels do not preserve the execution order of memory accesses (in order to model *non-FIFO* interconnections). Messages are never lost but they may be received in a different order than they are issued.

3.1 Cache States, Memory Commands and Memory States

In the verification method, we track the state of a single block. To simplify the discussion

we refer to the "state of the block in the cache" as the "state of the cache". The same convention applies to the state of the block in the directory and in other state machines throughout the paper.

3.1.1 Cache States

In the simplest form of the protocol, caches can be in three stable states: `Invalid` (`I`), `Shared` (`S`; clean copy potentially shared with other caches), and `Owner` (`O`; modified and only cached copy--also called `Exclusive`). Three transient states are added to each cache block frame to keep track of requests issued by the cache but not yet completed.

1. *Read-Miss-Pending* (`RMP`) state: the block frame is empty pending the reception of the block after a read miss.
2. *Write-Miss-Pending* (`WMP`) state: the block frame is empty pending the reception of the block with ownership after a write miss.
3. *Write-Hit-Pending* (`WHP`) state: the block frame contains a shared copy pending the reception of ownership rights to complete a write access.

These are traditional states in any protocol designed for a FIFO network. To deal with race conditions in non-FIFO networks we define three additional transient states which synchronize the interactions between caches and memory. These states are: *Transient Owner-to-Invalid* (`TxOI`), *Transient Shared-to-Invalid* (`TxSI`), and *Transient Owner-to-Shared* (`TxOS`). The possible races leading to the introduction of these transient states are illustrated in Figure 2 and coherence messages are defined in Table 1. In Figure 2.a, p_1 is granted an exclusive copy at the same time as p_2 wants an exclusive copy; the invalidation (`InvO`) requesting p_1 to yield ownership reaches p_1 before p_1 receives the exclusive copy first granted to it. To resolve this race, a cache in state `WHP` moves to state `TxOI` when it receives an invalidation so that, when it receives the data block, it executes its pending write, writes the block back to memory and invalidates its copy to end up in state `I`. A similar scenario can occur if p_1 first executes a read miss and request a shared copy, while p_2 wants an exclusive copy. When it receives the invalidation the cache of p_1 moves from state `RMP` into state `TxSI` as show in Figure 2.b so that, at the reception of the data block, it supplies the processor with the data and invalidates its block copy. Finally, state `TxOS` solves a similar race occurring when p_1 first requests an exclusive copy, while p_2 wants a shared copy (see Figure 2.c).

Traditionally, while a coherence transaction is in progress, the directory entry must be locked to maintain a critical (semi-critical) section on each memory block [22, 26, 28]. The scenarios of Figure 2 violate this requirement: the directory entry is unlocked after sending the response (such as the data block) to the requester. This optimization does not compromise correctness in systems with FIFO networks. In systems with non-FIFO networks the additional transient states are required. Without them, the processor which receives a block on a miss would have to acknowledge memory to release the locked directory entry explicitly; with them, concurrency of accesses to the same block is enhanced (which makes it less likely that a request must be aborted

and retried) and less acknowledgment messages are exchanged (which frees network bandwidth). These savings are significant, considering that the race conditions shown in Figure 2 are rare.

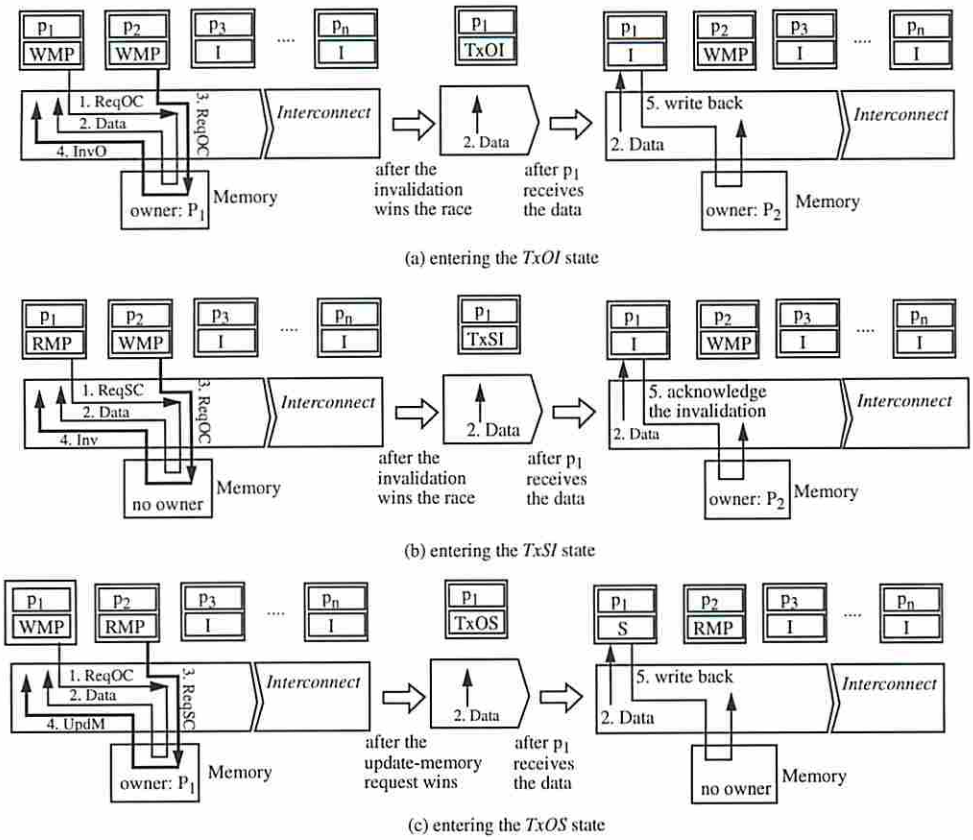


FIGURE 2. Scenarios Justifying the need for the Three Additional Cache States.

3.1.2 Memory States

The stable memory states are represented by the presence bits and the dirty bit in the directory entry. When the directory is in a stable state, it is free or unlocked, meaning that the memory controller may accept new requests for the block. Additionally, the directory may be in any one of five transient states: `XData`, `XOwn`, `XOwnC`, `Synch1` and `Synch2`. While in a transient state, the directory entry is locked. When a request reaches a locked (busy) directory entry it must be aborted and retried[1]. States `XData`, `XOwn` and `XOwnC` record that a request for a shared copy, for ownership rights or for an exclusive copy (respectively) is in progress and are classical states in systems with FIFO networks.

Transient directory states `Synch1` and `Synch2` are introduced to coordinate the interac-

1. Alternatively, the request could be queued at the memory module. Both alternatives can be easily modeled in the state expansion process.

tions between an owner and memory in non-FIFO networks. At the time an owner victimizes its modified copy for replacement, the memory state remains `Exclusive` until the write-back message reaches the memory controller. Between the transmission and the reception of the write-back message the memory controller may receive a request for a shared copy issued by another cache and forward it to the owner. When the memory controller receives the block copy sent at the time of replacement, it will "believe" that the block copy was sent in response to its forwarded request. This problem was also identified in [2]. The solution suggested in [2] does not distinguish between the two ambiguous messages and counts on the presumed owner to ignore the forwarded request. However, in a large-scale system with unpredictable network delays, intractable problems can be caused by the forwarded request if it is further outpaced by other messages.

TABLE 1. Coherence Messages

Type	Message	Action
Memory To Cache(Σ_r)	`Inv`	Request to invalidate the local copy.
	`InvO`	Request to invalidate the local copy and write it back to memory.
	`UpdM`	Request to update the main memory copy and change the local copy to the Shared state.
	`O-ship`	Ownership grant.
	`Data`	Block copy supplied by the memory controller.
	`NAck`	Negative acknowledgment indicating that a request was rejected because of a locked directory entry.
Cache To Memory(Σ_s)	`ReqSC`	Request a Shared copy.
	`ReqO`	Request Ownership.
	`ReqOC`	Request Ownership and block copy.
	`DxM`	Block copy supplied by an owner in response to an `UpdM` message from memory.
	`DOxMR`	Block copy supplied by an owner after replacement.
	`DOxMU`	Block copy supplied by an owner in response to an `InvO` message from memory.
	`IAck`	Acknowledgment indicating invalidation complete.
	`SAck`	Synchronization message.

To solve this problem in general, we need different message IDs for a write-back message caused by a replacement (`DOxMR`) or by an invalidation (`DOxMU`). Moreover, when the (presumed) owner receives the forwarded request after the replacement, it replies with a synchronization message (`SAck`). The memory controller unlocks the directory entry only when it has received both the synchronization message and the replaced data block, as shown in Figure 3. For example, when the memory controller receives a request for a shared copy (`ReqSC`), the request is forwarded to the owner and the memory state is changed to `XData`. If the memory controller then receives a block message of type `DxM` from the owner, it forwards a data copy to the requesting cache and completes the transaction. But, if the memory controller receives a block message of type `DOxMR` or a synchronization message (`SAck`) from the owner, the directory enters the

transient state Synch2 and waits for the synchronization message (SAck) or for the write-back message (DOxMR) from the owner respectively.

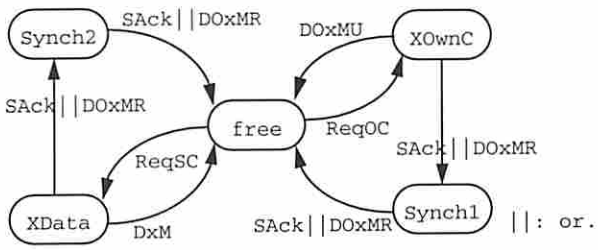


FIGURE 3. Synchronization Steps Between an Owner and Memory.

3.2 Cache Coherence Algorithm

We use Hoare's CSP language [12] to specify the detailed description of the cache coherence algorithm in Appendix A. The algorithm consists of five indexed processes **Proc**, **Cache**, **Replacement, SCH**, **RCH**, and **Memory**, which cooperate in the protocol. A repetitive execution, is indicated by a leading asterisk and represents as many iterations as possible of its constituent statements. The selection of execution is specified by an alternative command $[G_1 \rightarrow S_1 [] G_2 \rightarrow S_2 [] ... [] G_n \rightarrow S_n]$ where G_i is a *guard* boolean expression and S_i is a set of *guarded* statements. The process chooses and executes an arbitrary S_i for which G_i holds. Communication between two processes is specified by two input and output commands: `P!x` outputs a message `x` to process `P`, and `Q?y` inputs a message from process `Q` and saves the message in variable `y`. Two processes can communicate whenever their input and output commands match.

Coherence is enforced on each cache block. Generally, a data block is denoted by `blk` with address `blk.no` and with data values `blk.data`; its cache (memory directory) state is `Cache.st[blk.no]` (`Memory.st[blk.no]`). The value at location `addr` in the cache (or in the main) memory array is `Cache.m[addr]` (or `Memory.m[addr]`). We also use a unspecified function `BlockNo(addr)` which returns the address of the block containing the location `addr`, and use the notation `Cache.m[blk.no]` to represent all the data values in the block.

3.2.1 Processor and Cache Processes

The simple sequential program **Proc** describes the operation of a processor executing loads and stores repeatedly and stalled while waiting for the completion of its memory accesses. The **Cache** process specifying the activity of the cache controller is a non-terminating loop, each step of which accepts the request from the local processor or consumes a message from the receiving channel.

Given the current cache state, the message received from the network or the request sub-

mitted by the processor, Table 2 indicates the next state and the reply messages. Since the replacement of the block is caused by an access miss to a different block we model it by a control step that stimulates the expansion process with a memory access to a different (irrelevant) address causing the eviction of the block being tracked in the verification. To simplify our algorithmic descriptions in Appendix A, the cache block frame containing the victim is implicitly reserved for the new data block.

TABLE 2. The Cache State Transition Table

	Message from Network						Request from Processor		
C-State	Inv	InvO	UpdM	O-ship	Data	NAck	Read	Write	Repl
I	I/IAck	I/SAck	I/SAck	Error	Error	Error	RMP/ReqSC	WMP/ReqOC	I
O	Error	I/DOxMU	S/DxM	Error	Error	Error	O	O	I/DOxMR
S	I/IAck	Error	Error	Error	Error	Error	S	WHP/ReqO	I
RMP	TxSI	RMP/SAck	RMP/SAck	Error	S	RMP/ReqSC	---	---	---
WMP	WMP/IAck	TxOI	TxOS	Error	O	WMP/ReqOC	---	---	---
WHP	WMP/IAck	TxOI	TxOS	O	Error	WHP/ReqO	---	---	---
TxOI	Error	Error	Error	I/DOxMU	I/DOxMU	WMP/SAck, ReqOC	---	---	---
TxSI	Error	Error	Error	Error	I/IAck	RMP/IAck, ReqSC	---	---	---
TxOS	Error	Error	Error	S/DxM	S/DxM	WMP/SAck, ReqOC	---	---	---

notations: 1. next state / response message(s), 2. ---: processor is blocked.

3.2.2 Channel and Memory Processes

The channels for sending and receiving messages transmit the messages between caches and main memory. Several auxiliary functions such as *InsertSCH*, and *GetSCH* are clear and do not need further explanation. The state transitions of the memory process are specified in Table 3. Basically, while the directory entry is locked due to some pending request, requests to the block are aborted. Also, if no owner exists, the memory supplies its copy to requesters of shared copies; otherwise, the owner is asked to write its dirty block to main memory first, and then the block is forwarded to the requester. To satisfy a request for an exclusive copy, the memory controller must

first invalidate all cached copies.

TABLE 3. State Transition Table of the Memory Controller

M-State	Message	Condition	Goto	Action
Free	ReqSC	No owner exists.	Free	Memory provides a copy to the requester.
		Owner exists.	XData	Ask current owner to write back.
	ReqO	Presence bit of the requester is not set.	Free	Reject the request.
		The requester has the only cached copy.	Free	Memory transfers ownership to the requester.
		The block is shared.	XOwn	Send an invalidation to every cache with a copy.
	ReqOC	No cached copy exists.	Free	Memory sends a copy to the requester (new owner).
		The block is shared.	XOwnC	Send an invalidation to every cache with a copy.
		Owner exists.	XOwnC	Ask owner to relinquish and write back its copy.
	DOxMR	Write-back message from current owner.	Free	Update the memory.
	Others	DxM, DOxM, IAck, SAck	Error	Unspecified message reception.
XData	DxM	Update message from owner.	Free	Update memory; provide data to the pending requester.
	DOxMR	Write-back message from owner.	Synch2	Update main memory.
	SAck	Synchronization message from owner.	Synch2	Wait for write-back message from current owner.
	Request	ReqSC, ReqO, ReqOC	XData	Reject the request.
	Others	DOxMU, IAck	Error	Unspecified message reception.
XOwnC	DOxMR	Write-back message from owner.	Synch1	Update main memory.
	DOxMU	Update message from owner.	Free	Memory forwards data to the pending requester (new owner).
	IAck	Some cached copy still exists.	XOwnC	Waiting for invalidation acknowledgments.
		All cached copies are invalidated.	Free	Memory sends its copy to the pending requester (new owner).
	SAck	Synchronization message from owner.	Synch1	Wait for write-back message from current owner.
	Request	ReqSC, ReqO, ReqOC	XOwnC	Reject the request.
	Others	DxM	Error	Unspecified message reception.
XOwn	IAck	All cached copies are invalidated.	Free	Memory transfers ownership to the pending requester (new owner).
		Some cached copy still exists.	XOwn	Waiting for invalidation acknowledgments.
	Request	ReqSC, ReqO, ReqOC	XOwn	Reject the request.
	Others	DxM, DOxM, DOxMU, SAck	Error	Unspecified message reception.
Synch1	DOxMR	Write-back message from owner.	Free	Memory forwards the write-back data to the pending requester (new owner).
	SAck	Synchronization message from owner.	Free	Memory sends its copy to the pending requester (new owner).
	Request	ReqSC, ReqO, ReqOC	Synch1	Reject the request.
	Others	DxM, DOxMU, IAck	Error	Unspecified message reception.
Synch2	DOxMR	Write-back message from owner.	Free	Update memory; provide data to the pending requester.
	SAck	Synchronization message from owner.	Free	Memory sends its copy to the pending requester.
	Request	ReqSC, ReqO, ReqOC	Synch2	Reject the request.
	Others	DxM, DOxMU, IAck	Error	Unspecified message reception.

Another race problem, independent of the fact that the network is FIFO, was first identified in [2] and is illustrated in Figure 4. Initially, both p_1 and p_2 share a copy, and request ownership of the block (ReqO). When p_1's copy is invalidated by p_2's request, the request for ownership issued by p_1 becomes inconsistent with the state of p_1's cache, which no longer has a copy and whose presence bit is reset at the memory. Instead, a ReqOC message (asking for ownership and for the block) is expected by the memory controller. To solve this problem, the memory controller must check the presence bits when a ReqO message is received. If the presence bit

is reset, the memory controller can either interpret the ReqO message as a ReqOC message [2] or simply reject it. An alternative solution that always reloads the block on a write access was also proposed to solve this problem [5, 17]. However, when the above case is rare, superfluous transfers of the data block may be costly.

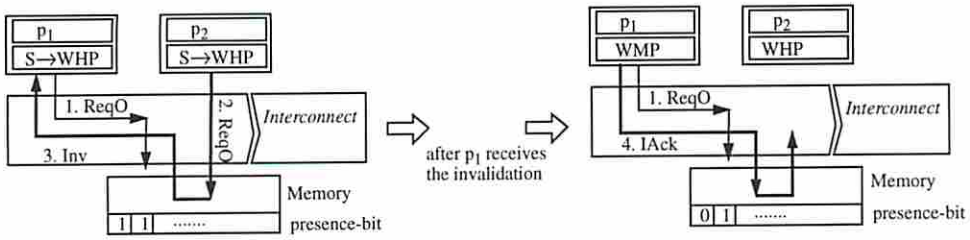


FIGURE 4. A Ghost Message.

4 Correctness Issues of the Protocol

4.1 Data Consistency

In systems where writes are not atomic, data consistency between multiple data copies is typically enforced by allowing one and only one update in progress at any time for each block [26]. As a result, concurrent accesses can be executed on different data copies but will appear to have executed atomically in some sequential order. The cache protocol must always return the latest value on each load. We formulate this condition within the framework of the reachability expansion as follows.

Definition 1. (Data Consistency) *With respect to a particular memory location, the protocol preserves data consistency if and only if the following condition is always true during the reachability analysis: the family of global states originated from G', including G' itself, consistently observe the value written by a STORE transition* τ *which brings a global state G to G' or the value written by STORE transitions after* τ*. That is, states reached by expanding G' are not allowed to access the old value defined before* τ*.*

Specifically, a cache may have a data block in one of three status: nodata (the cache has no copy), fresh (the cache has an up-to-date copy), and obsolete (the cache has an out-of-date copy); the memory copy is also either fresh or obsolete [23]. During the course of verification, the expansion process keeps track of the status of all block copies in conformance with the protocol semantics. For example, when a processor successfully writes to its local copy, it owns a fresh copy and all other data copies (including the memory copy) become obsolete.

4.2 Unspecified Message Reception

A commonly accepted definition for unspecified message reception is that some entity in

the protocol receives a message which is unexpected given its current state [29]. This type of flaw due to an incomplete specification is likely to appear at the early design phase of the protocol because of unforeseen interleavings of memory accesses. Therefore, verification techniques that assist in incrementally synthesizing and incrementally verifying the protocol are preferable. To this end, a logic proof may not be suitable since a proof of an incorrect protocol provides an existence evidence rather than a constructive expostulation. On the other hand, a state machine model is able to show the path leading to the erroneous state. Moreover, the detection procedure for unspecified message reception is simple and is directly tied to the structure of the reachability graph: an unspecified message reception is detected when the system is in a state and a message is received for which no transition out of the state is specified in the protocol description.

4.3 Deadlock and Livelock

Within the framework of reachability analysis, a deadlock means that the protocol enters a state without possible exit (looping indefinitely in the same state) and a livelock is a situation where processes interacting in the protocol could theoretically make progress but, because of a fortuitous timing of events, they are trapped in a loop of states without making progress.

Definition 2. (Livelock) *In the context of coherence protocols, a livelock is a condition in which a given block is locked by one processor so that some processor is permanently prevented from accessing the block* [20].

Given the above definition, we check the following two conditions in order to detect livelocks and correct the protocol:

Conditions for Livelock-freeness: *A cache coherence protocol is livelock-free if and only if it satisfies two conditions:*

(a) The protocol can visit every state in the global state transition diagram infinitely many times, that is, the global state transition diagram must be strongly-connected. Given a global state, every other state in the global state transition diagram is reachable [2].

(b) If a processor issues a memory access to a block, this memory access must eventually be satisfied even if it may give rise to coherence actions such as requesting a missing block. Specifically, given an initial global state in which a cache is in an "invalid" state, there must exist reachable global states in which the cache state becomes "shared" or "dirty" after a read miss or a write access [19].

(a) is obviously a necessary condition for a livelock-free protocol because, without that condition, a protocol can enter into a set of global states in which the system is locked forever, with no escape route. (a) and (b) form a sufficient condition for definition 2 because, with the aid of condition (a), (b) means that a processor can access a block an arbitrary number of times.

5 Symbolic State Model

5.1 Formal Protocol Model

Given the architectural model of Figure 1, we now formally define the constituent finite state machines interacting in the protocol. Following the CSP notation [12], message transmission is represented by the postfix '!', and message reception by the postfix '?'.

Definition 3. (Receiving Channel) *The receiving channel machine recording the messages received from the memory and in transit to the cache has structure* $\mathcal{RCHM}=(Q_r, \Sigma_r, \mathcal{X}m!, \delta 1_r, \mathcal{X}c?, \delta 2_r)$, *where*

Q_r: *state symbols,*

Σ_r: *set of memory-to-cache messages* (Table 1),

$\delta 1_r$: $\mathcal{X}m! \times Q_r \rightarrow Q_r$, $\mathcal{X}m! \in \Sigma_r$, (messages from memory),

$\delta 2_r$: $Q_r \times \Sigma_r \rightarrow Q_r \times \mathcal{X}c?$, $\mathcal{X}c? \in \Sigma_r$, (messages to cache),

$\mathcal{X}m!$ *and* $\mathcal{X}c?$ *are the messages issued by the memory controller and consumed by the cache, respectively.*

Definition 4. (Sending Channel) *The sending channel machine recording the messages issued by the cache and in transit to the memory controller has structure* $\mathcal{SCHM}=(Q_s, \Sigma_s, \mathcal{X}c!, \delta 1_s, \mathcal{X}m?, \delta 2_s)$, *where*

Q_s: *state symbols,*

Σ_s: *set of cache-to-memory messages* (Table 1),

$\delta 1_s$: $\mathcal{X}c! \times Q_s \rightarrow Q_s$, $\mathcal{X}c! \in \Sigma_s$, (messages from cache),

$\delta 2_s$: $Q_s \times \Sigma_s \rightarrow Q_s \times \mathcal{X}m?$, $\mathcal{X}m? \in \Sigma_s$, (messages to memory),

$\mathcal{X}c!$ *and* $\mathcal{X}m?$ *are the messages issued by the cache and consumed by the memory, respectively.*

The state of a channel machine is made of all the messages in transit. At each state expansion step, a receiving (sending) channel may record the command sent by the memory (its cache), or may propagate a command to its cache (the memory).

Definition 5. (Cache Machine) *The state machine characterizing the cache behavior has structure* $\mathcal{CM}=(Q_c, \Sigma_r, \Sigma_s, \mathcal{X}c?, \delta 1_c, \mathcal{X}c!, \delta 2_c)$, *where*

Q_c: *cache state symbols,*

Σ_r, Σ_s: *coherence messages as defined in definitions* 3 *and* 4,

$\delta 1_c$: $\mathcal{X}c? \times Q_c \rightarrow Q_c \times (\varnothing \cup \mathcal{X}c!)$, $\mathcal{X}c? \in \Sigma_r, \mathcal{X}c! \in \Sigma_s$,

$$\delta 2_c: \quad Q_c \times \Sigma_s \rightarrow Q_c \times Xc!, \quad Xc! \in \Sigma_s ,$$

Xc? and Xc! are the messages consumed and produced by the cache, respectively.

The behavior of the cache controller is given in definition 5. Upon receiving a message, a cache controller may or may not respond by generating response messages according to $\delta 1_c$. Additionally, the cache may issue requests as described by $\delta 2_c$. Finally, we have the definitions for main memory and for the global states as follows.

Definition 6. (Memory-Directory) *The main memory machine keeping the directory has structure* $\mathcal{MM}=(Q_m, \Sigma_r, \Sigma_s, Xm?, \delta_m, Xm!)$, *where*

Q_m: *memory state symbols,*

Σ_r, Σ_s: *messages as defined in definitions* 3 *and* 4,

$$\delta_m: \quad Xm? \times Q_m \rightarrow Q_m \times (\varnothing \cup Xm!), \quad Xm? \in \Sigma_s, Xm! \in \Sigma_r \quad ,$$

Xm? and Xm! are the caches-to-memory and memory-to-caches commands respectively.

Definition 7. (Base Machine) *The base machine is the composition of the cache machine and of its two corresponding channel machines, that is,* $\mathcal{BM}_i=(\mathcal{CM}_i \# \mathcal{RChM}_i \# \mathcal{SChM}_i)$.

Definition 8. (Protocol Machine) *The protocol machine is defined as the composition of all base machines and of the memory machine, that is,* $\mathcal{PM}=(\mathcal{BM}_1 \# \mathcal{BM}_2 \# \ldots\ldots \# \mathcal{BM}_n \# \mathcal{MM})$ *for a system with n caches.*

The memory controller consumes messages from caches and responds according to the block state and the message type. Finally, the state of the protocol machine is also referred to as the *global state* in this paper.

5.2 State Equivalences

The exploitation of equivalences among states in order to reduce the complexity of the state space search can benefit any state enumeration method. In Murφ-ns, two states are equivalent only if they are identical. In Murφ-s, equivalence is extended to take advantage of the symmetry in the machine model. For instance, the model of Figure 1 exhibits obvious structural symmetry. The order of the states of base machines in a global state representation is irrelevant to protocol correctness. In the SSM method, we further broaden the equivalence relations based on the following two observations:

Observation 1. The behavior of every cache (base machine) can be modeled by the same finite state machine, and

Observation 2. In all existing protocols, data coherence is enforced by either broadcasting writes to all copies so that they remain coherent, or by invalidating the copies in all other caches so that

modifications are done in one and only one cache at a time.

Clearly, the exact number of data copies in a clean, shared state is irrelevant to protocol correctness. What is really critical is whether there exists 0, 1, or multiple copies in a particular state (such as more than one Owner). Therefore, we can map global states to more abstract states, which do not keep track of the exact number of machines in particular states. We use the following notation to represent these abstract states.

Definition 9. (Repetition Constructors)

1. *The* **Singleton** (1) *indicates one and only one instance. This constructor can be omitted.*
2. *The* **Plus** (+) *indicates one or multiple instances.*
3. *The* **Star** (*) *indicates zero, one or multiple instances.*
4. *The* **Universe** (v) *indicates zero, one or multiple instances (same as *).*

These repetition constructors are useful for building concise representations of complex states. For example, we can represent the set of global states such that "one or multiple caches are in the Invalid state, and zero, one or multiple caches are in the Shared state" as (I^+, S^*). A set of base machines in the same state is represented by

$$\underset{R}{\overset{p}{C}}{}_{S}^{r}$$

in which C is the cache state, p is the value of the presence bit in the directory, r is the number of base machines in the set (specified by one of the repetition constructors above), R is the state of the receiving channel, and S is the state of the sending channel. R and S are specified by the messages in transit in the channels. Since the channels model non-FIFO networks the order of the messages in each channels is irrelevant. Often, when there is no confusion, part of the notation may be omitted. For example, we use the notation ${}^{p}q^{r}$, where q combines the cache state with the states of its two message channels.

Although the singleton, the plus and the star are useful in representing an unspecified number of instances of a given construct (such as base machines in a given global state), they are not precise enough to model intermediate states in protocol transactions relying on event counting. Consider an abstract system state (S^*,...), in which the memory controller receives a request for an exclusive copy. Two cases should be considered, according to the definition of *: the case where no copy is cached and the case where at least one copy is cached. The same problem also arises while counting the number of invalidation acknowledgments received by the memory controller when a request for an exclusive copy is pending at the memory[2]. In principle, we could overcome this problem by introducing *non-determinism* in the state expansion process. For example, when the memory controller receives the request for the exclusive copy in the system state

2. This problem does not exists when invalidations are propagated atomically, which explains why v was not defined in [23].

(S^*,...), two states, one corresponding to no cached copy (S^0,..) and one corresponding to at least one cached copy (S^+,..) could be generated; however, as soon as one invalidation has been received, S^+ must turn into S^* again, and the process never stops. As we will show in Section 5.4, using the * constructor by itself to represent any number of copies may prevent the expansion of some possible states; the problem is solved by introducing the universe constructor v, which has the same meaning as * but is used at a different phase of the expansion to refine *.

In a system with an unspecified number of caches, we group base machines in the same state into *state classes* and specify their number in each class by one of the repetition constructors.

Definition 10. (Composite State) *A composite state represents the state of the protocol machine for a system with an arbitrary number of cache entities. It is constructed over state classes of the form* $(q_1^{r_1}, q_2^{r_2}, ..., q_n^{r_n}, q_{\mathcal{MM}})$, *where* $n=|Q_{\mathcal{BM}}|$[3] *is the number of states of a base machine,* $q_i \in Q_{\mathcal{BM}}$, $r_i \in [0, 1, +, *, v]$[4] *and* $q_{\mathcal{MM}} \in Q_m$ *is the memory machine state.*

Repetition constructors can be ordered by the possible states they specify. The resulting order is $1 < + < * < v$; the null instance can be ordered with respect to * and v, i.e., $0 < * < v$. This order leads to the definition of *state containment*.

Definition 11. (Containment) *We say that composite state* S_2 *contains composite state* S_1, *or* $S_1 \subseteq S_2$, *if*

$$\forall q^{r_1} \in S_1 \quad \exists q^{r_2} \in S_2 \quad such \quad that \quad q^{r_1} \leq q^{r_2} \text{ i.e. } r_1 \leq r_2 and\ q_{\mathcal{MM}1} = q_{\mathcal{MM}2}$$

where $q \in Q_{\mathcal{BM}}$ *and* $r_1, r_2 \in [0, 1, +, *, v]$.

The consequence of containment is that if $S_1 \subseteq S_2$, then the family of states represented by S_2 is a superset of the family of states represented by S_1. Therefore, S_1 can be discarded during the verification process provided we keep S_2. The expansion process based on the expansion rules defined in Section 5.3 is a *monotonic* operator on the set of composite states S, that is, if $S_1 \subseteq S_2$, then $\tau(S_1) \subseteq \tau(S_2)$ where operator τ is a memory event.

Theory 1. (Monotonicity) If $S_1 \subseteq S_2$, then for every $\overline{S}_1$ reachable from S_1 there exists $\overline{S}_2$ reachable from S_2 such that $\overline{S}_1 \subseteq \overline{S}_2$.

Proof: See Appendix B.

As the expansion process progresses, new composite states are created. A new state is discarded if it is contained in a visited state and all visited states contained in a newly expanded state are discarded. At the end of the expansion process all visited states are *essential* states.

---

3. $Q_{\mathcal{BM}}$ denotes the set of possible states of a base machine.

4. The constructor "0" means "null instance" and is added for completeness.

Definition 12. (Essential State) *Composite state S is essential if and only if there does not exist a composite state* $\overline{S}$ *such that* $S \subseteq \overline{S}$.

The state space at the end of the expansion process is partitioned into several families of states (which may be overlapping) represented by essential composite states.

5.3 Rules and Algorithm for the Expansion Process

The set of operators applicable to composite states during the state generation process is defined as follows, where '/' signifies "or" selection and '$\rightarrow$' means a transition. Also, the state of the machine corresponding to a requester (owner) recorded at the memory is specially annotated with a subscript as q_{req} (q_{own})

1. **Aggregation**: $(q^0, q^r) \equiv q^r$, $(q^*, q^*) \equiv q^*$, $(q^v, q^{*/v}) \equiv q^v$, $(q, q^{1/+/*/v}) \equiv q^+$, and $(q^+, q^{1/+/*/v}) \equiv q^+$, where $q \in Q_{\mathcal{BM}}$. Aggregation rules are equivalence rules between composite states obtained by merging base machine states.

2. **Coincident Transition**: ${q_1}^r \rightarrow^{\tau} {q_2}^r$, where $r \in [1, +, *, v]$ and τ is an observed transition. For instance, the memory controller sends an invalidation signal to every cache with a valid copy.

3. **One-step Transition**: $q_1 \rightarrow^t q_2$, ${q_1}^{+/*/v} \rightarrow^t (q_2, {q_3}^*)$ where t is a transition applied to the base machine in state q_1 such that $q_1 \rightarrow^t q_2$, and t causes all other base machines in q_1 to move to state q_3. For instance, if the memory receives a request for an exclusive copy from a base machine in class q_1, this particular base machine changes its state to q_2 (the request message is removed from the sending channel); in this case all other machines in q_1 stay in the same state because the memory can process only one request for an exclusive copy at a time.

4. **N-step Transitions**: This rule specifies the repetitive application of the same transition N times, where N is an arbitrary positive integer.
   (a) $(Q, {q_1}^+, q_{\mathcal{MM}}) \rightarrow^t (Q, {q_2}^1, {q_1}^*, q_{\mathcal{MM}}) \rightarrow^t (Q, {q_2}^2, {q_1}^*, q_{\mathcal{MM}}) \rightarrow^t \ldots \rightarrow^t (Q, {q_2}^+, {q_1}^*, q_{\mathcal{MM}})$,
   (b) $(Q, {q_1}^{*/v}, q_{\mathcal{MM}}) \rightarrow^t \ldots \rightarrow^t (Q, {q_2}^*, {q_1}^v, q_{\mathcal{MM}})$
   The same transition t: $q_1 \rightarrow^t q_2$ can be applied infinitely many times as long as there are base machines in state q_1. Every application of the transition brings down the number of base machines in state q_1 by one and increases the number of base machines in state q_2. The transition t has no effect on other machines (denoted by Q in the tuple). Typical examples are: (1) processors replacing their copy in a shared state, (2) processors receiving the same type of messages, and (3) processors issuing the same memory access independently.

5. **Progress Transitions**: Two additional rules with similar interpretation as N-step transitions are required for the progress of the expansion process. Provided $q_1 \rightarrow^t q_2$, we have
   (a) $({}^0Q, {}^1\textit{Inv-Set}, {}^1{q_1}^v, q_{req}, q_{\mathcal{MM}}) \rightarrow^t \ldots \rightarrow^t ({}^0Q, {}^1\textit{Inv-Set}, {}^0{q_2}^*, {}^1{q_1}^v, q_{req}, q_{\mathcal{MM}})$, and

(b) $({}^{0}Q,\ {}^{1}Inv\text{-}Set,\ {}^{1}q_1{}^{v},\ q_{req},\ q_{\mathcal{MM}}) \rightarrow^{t} \ldots \rightarrow^{t} ({}^{0}Q,\ {}^{0}q_2{}^{*},\ q_{req} \rightarrow q_{own},\ q_{\mathcal{MM}}')$.

These two rules model the processing of a request for an exclusive copy at the memory. Transition *t* is an acknowledgment for an invalidation request (such as the `IAck` message in Table 1). *Inv-Set* (for *Invalidation-Set*) represents the set of caches with their presence bits set at the memory and which must be invalidated before the memory grants an exclusive copy. Rule (a) applies during the invalidation process whereas rule (b) applies after the successful invalidation of all copies. The justification for these rules is presented in the following section.

Algorithm: Essential States Generation.

W: list of working composite states.
H: list of visited composite states.(output:essential states)

```
while (W is not empty) do
begin
    get current state A from W.
    for all cache state class v ∈ A
        for all applicable operations τ on v
            A →^τ A'.
            for any state P ∈ W and Q ∈ H
              if (A' ⊆ P or A' ⊆ Q or A' ⊆ A)
                 then discard A'.
              else begin
                      remove P from W if P ⊆ A'.
                      remove Q from H if  Q ⊆ A'.
                      add A' to W.
                      if (A ⊆ A') then discard A  and terminate
                        all FOR loops starting a new run.
                    end
        end
    end
    insert A to H if A is fully expanded and is not contained.
end.
```

FIGURE 5. Algorithm for Generating Essential States.

During the state expansion process, the next state is produced by stimulating the current state, by exploring all possible cache transactions and by repeatedly applying the above rules. The algorithm for the symbolic state expansion process is shown in Figure 5.

5.4 Need for the Universe Constructor

In this section, we justify the introduction of the universe constructor v and of the progress expansion rule. When a request for an exclusive copy (such as `ReqO` or `ReqOC`) reaches the memory, all copies in the invalidation set must be invalidated and the expansion process needs to count whether or not the invalidation set is empty. Since all caches in the same state are specified by repetition constructors, the exact number of caches in a particular state is unknown.

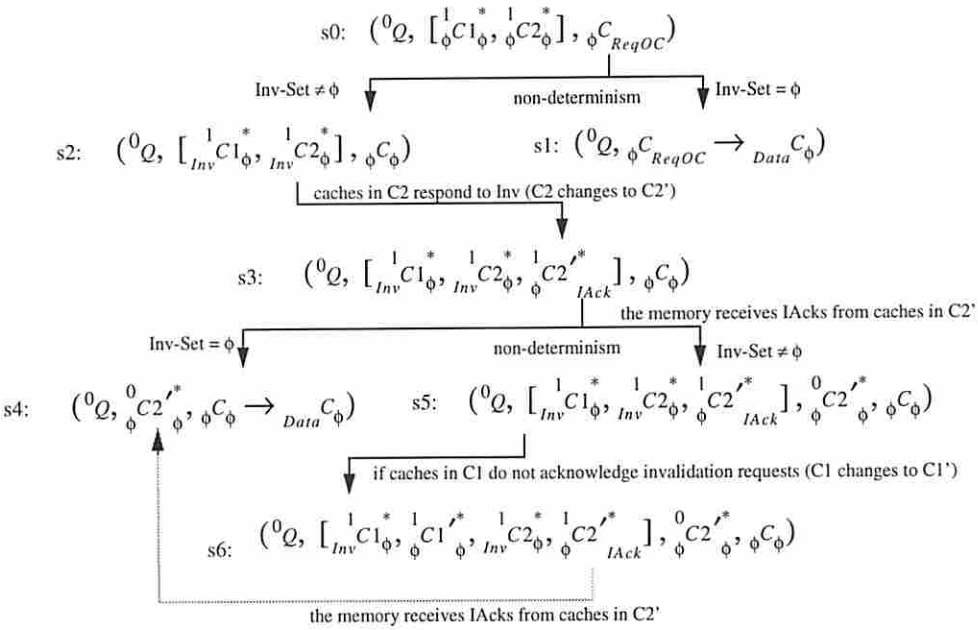


FIGURE 6. Expansion Step with Non-determinism.

Let's consider the following composite state with the invalidation set between brackets:

$$({}^{0}Q, [{}^{1}_{\phi}C1^{*}_{\phi}, {}^{1}_{\phi}C2^{*}_{\phi}], {}_{\phi}C_{ReqOC}) \qquad Q \in Q_{\mathcal{BM}},$$

where Q denotes all other base machines with presence bits reset. When the memory receives the request for and exclusive copy (`ReqOC`) from the cache in state C, it cannot determine whether or not the invalidation set is empty because the definition of * includes the case of null instance. To solve this shortcoming in the notation, we could add *non-determinism* to the expansion process. When the global state is expanded, two states, corresponding to an empty and to a non-empty invalidation set are generated. Expansion steps are shown in Figure 6 for a particular example.

1. In s0, suppose that memory receives a request for an exclusive copy from the cache in state C. Two states corresponding to an empty and to a non-empty invalidation set are generated. In s2,

invalidations are sent to caches in the invalidation set, whereas in s1, the requester obtains an exclusive copy and becomes the owner.

2. When expanding s2, caches in state $C2$ receive invalidations, respond with an invalidation acknowledgment, and $C2$ changes to $C2'$.

3. When the memory receives invalidation acknowledgments from caches in $C2'$ in s3, two states corresponding to an empty and to a non-empty invalidation set are again generated.

4. In global state s5, caches in state $C1$ do not acknowledge invalidations because of an incorrect design. This error goes undetected. In s6, when the `IAck` messages from caches in $C2'$ are received by the memory, the expansion procedure may consider the Inv-Set as empty again and make a transition to s4. However, the case where the Inv-Set is not empty is also covered by * and must be expanded too. So, either the process never stops or some errors are not detected.

The v constructor must be added to systematically exercise all possible states, yet preserve the power of ordering between 0 and * constructors. Briefly, a given state of the form $(x_1^*, x_2^*, \ldots, x_n^*)$ is considered as an intermediate state of $(\bar{x}_1^v, \bar{x}_2^v, \ldots, \bar{x}_n^v)$, which is generated after a sequence of state transitions on $(x_1^*, x_2^*, \ldots, x_n^*)$ such that $x_i \rightarrow \bar{x}_i$, where x_i's are arbitrary variables. When a transition is applied to the state $(x_1^*, x_2^*, \ldots, x_n^*)$, no state corresponding to a null case $(x_1^0, x_2^0, \ldots, x_n^0)$ is generated, whereas when the state $(\bar{x}_1^v, \bar{x}_2^v, \ldots, \bar{x}_n^v)$ is expanded, two states corresponding to a null case and a non-null case are generated. Afterwards, an Inv-Set may be considered as an empty set if and only if it has the form:

$$(\ldots, [{}^1q_1^v, {}^1q_2^v, \ldots, {}^1q_n^v], \ldots) \quad \text{where} \quad q_i \in Q_{\mathcal{BM}}$$

Let's examine the expansion steps using the v constructor and shown in Figure 7 to see how the procedure works.

1. In global state s1, the expansion process explores the path in which some $C2$ caches respond to invalidations. At this point, the expansion process only considers the case where the class $C2$ is not empty. As a result, in global state s2, the class of caches remaining in $C2$ is indicated by v, and, the next time they are expanded, the expansion process will consider the empty-set case.

2. In global state s3, the expansion process choose to expand the class of caches in state C1, considering only the case of the non-empty set. If caches in state $C1$ do not acknowledge invalidations, the process moves into state s4. According to the condition for emptiness of the invalidation set, the pending request for an exclusive copy in state s4 is never resumed (because of the caches in class $C1'$). This situation is easily detected as a livelock situation in our framework.

3. On the other hand, if all caches in the invalidation set acknowledge memory, the expansion process takes another path through states s4', s5 and s6, as shown in Figure 7.

6 Protocol Error Detection

Since unexpected message reception errors are easy to detect, we only describe the more subtle livelock error detected during the course of this study. Second, we compare the performance of the SSM method, of Murφ-ns and of Murφ-s in terms of time complexity and memory usage, for the purpose of verifying the protocol introduced in this paper. For all three verification procedures, the expansion process starts with an initial state with no cached copy, empty message channels and free memory state; therefore, in the SSM method, the initial state is $({}^{0}_{\phi}\mathtt{I}^{+}_{\phi}, \mathtt{free})$.

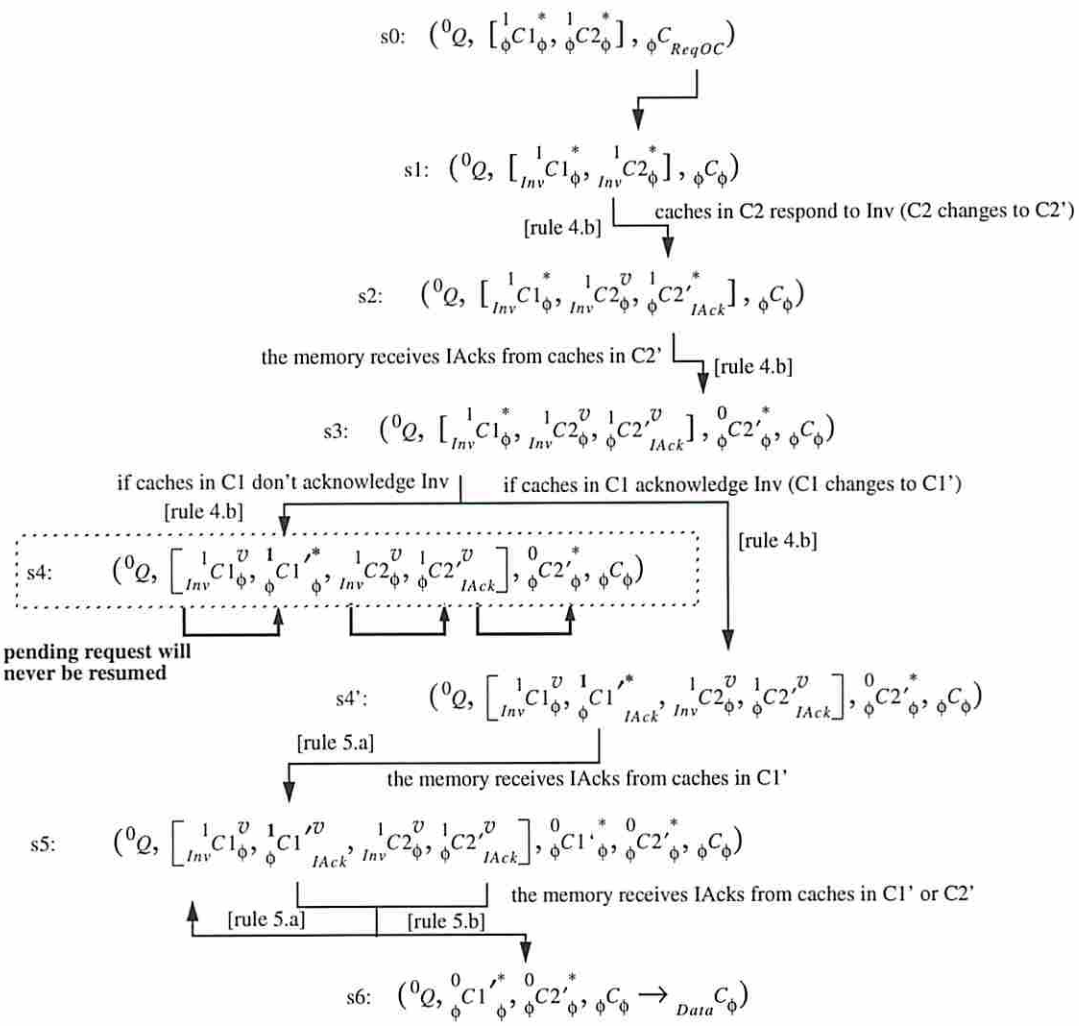


FIGURE 7. Resolution Provided by the v Constructor.

6.1 Livelock

The expansion steps leading to a livelock in the original protocol are now described. Initially, consider a system state with an owner and no request in progress (directory entry is free).

The state has the form (....., ${}^{1}_{\phi}O^{1}_{\phi}$,..., free). (Because other caches do not participate in the live-lock situation, we only specify the state of the owner and of the directory entry.)

Consider the following scenario:

1. The owner replaces its copy and writes the block back to memory. The generated system state is (...., ${}^{1}_{\phi}I^{1}_{\text{DOxMR}}$,..., free), indicating that a write-back message is in the output channel.
2. Next, the same cache experiences a write miss and sends a request for an exclusive copy. The new state is (...., ${}^{1}_{\phi}WMP^{1}_{\text{DOxMR, ReqOC}}$,..., free) and a race exists between the write-back and the ownership messages in the case of a non-FIFO network.
3. The memory may receive the ownership message before the write back message; in this case the memory state is changed to XOwnC and an invalidation (InvO) is sent to the cache because the memory still records that the cache is an owner. The resulting state is (...., ${}_{\text{InvO}}{}^{1}WMP^{1}_{\text{DOxMR}}$,..., XOwnC).
4. The cache receives the InvO message and changes its state to TxOI (Table 2). The system state becomes (...., ${}^{1}_{\phi}TxOI^{1}_{\text{DOxMR}}$,..., XOwnC).
5. Finally, when the memory receives the write-back message, it enters the synchronization state Synch1 and expect a synchronization message SAck from the cache (Figure 3). The system state is (...., ${}^{0}_{\phi}TxOI^{1}_{\phi}$,..., Synch1). However, the synchronization message will never be sent by the cache, which locks the directory entry forever.

In the SSM method, this error was successfully detected by reporting that a cycle exist between four global states without exits to a state outside the loop as shown in Appendix C (the global state transition diagram is not strongly connected). This error was not detected by the present Murφ system. Detection procedures for checking the connectivity of the global state diagram, become overwhelmingly complex when the size of the global state diagram is large.

The livelock condition originates from the fact that memory does not check the presence bits when it receives an ownership request (see Table 3 and line 99 of the algorithm in Appendix A). The livelock can be removed by the following correction to the protocol. When the memory receives a ReqOC message, it checks whether the processor identifier of the message corresponds to the current owner. If it does, the memory state is changed to the synchronization state Synch1 directly (following the state diagram in Figure 3). Later, when the write-back message arrives, the memory updates its copy of the block, supplies the cache with the copy of the block and unlocks the directory entry, according to Table 3.

6.2 Comparison with the Murφ System

For verification methods using exhaustive search, the time complexity and memory usage are two critical concerns closely related to the size of the system state space. Generally, an exhaustive search algorithm performs three fundamental operations.

1. Generate a new state, if there is any left; otherwise the algorithm terminates and reports the final set of global states.
2. Compare the new state against the set of previously visited states.
3. Keep the new state for future expansion if the new state has not been visited before.

The most time-consuming operation is comparing the new state to previously visited states. The time complexity grows in proportion to the size of the search state space (the set of states generated and analyzed during the procedure), while the memory usage increases with the size of the global state space (the set of states saved and reported at the end). As a matter of fact, the search space is a direct expansion of the global state space; therefore, reducing the size of the global state space is particularly important. The comparison with the Murφ system is relevant because Murφ is an efficient verification tool based on state enumeration, incorporating state encoding to reduce memory usage and hash tables to speed up the search and comparison operations. The comparison between SSM and Murφ can assess the performance advantages afforded by the drastic reduction of the global state space in the symbolic state model.

TABLE 4. Comparison between SSM, Murφ-ns and Murφ-s

Method	Number of processors	Size of global state space	Size of search space	Verification time (seconds)	Memory usage (Mbytes)
Murφ-ns	2	619	1672	1.5	0.02
	3	12195	49731	33.6	0.42
	4	252369	1455228	1028.6	11.6
	5	excessive memory usage (over 200Mbytes)			
Murφ-s	2	332	911	1.2	0.009
	3	3261	13765	22.8	0.11
	4	38925	234444	1395.6	1.79
	5	559899	4540185	142985.2	29.96
SSM	any n >1	123	25631	167	1.6

Table 4 shows performance comparisons between Murφ-ns, Murφ-s, and SSM running on a SPARCstation 10 Model 30 with 128 MBytes of memory for the verification of the protocol. The information in Table 4 includes (1) the number of processors in the verification model, (2) the size of the global state space, (3) the number of states generated during the verification (the search space), (4) the verification time, and (5) the memory usage.

We make the following observations. First, for small-scale systems with less than five processors, the time complexity and the memory usage of Murφ-ns and Murφ-s are tolerable. Second, the sizes of both the global state space and the search space of Murφ-s are significantly less than those of Murφ-ns, but there is little difference in the times taken by both methods. In the case of four processor systems, we observed that Murφ-s may take longer than Murφ-ns. The extra overhead due to the state permutation mappings in Murφ-s may explain this unexpected phenomenon. As compared to Murφ, the SSM method is very efficient with 167 seconds of verification time and 1.6M bytes of memory consumption. (No state encoding was applied.) Moreover, the small size of the global state space (123 global states) demonstrates that the regular structure and symmetry of homogeneous multiprocessors yields compact state representations. Third, the main advantage of exploiting symmetry in Murφ is the reduction of the size of the global state space and of the memory requirement. However, as shown in Table 4, this advantage dwindles quickly as more processors are added in the model and both the verification time and the memory usage increase drastically.

The fact that the performance of classical enumeration techniques is acceptable for small system sizes raises the question of whether more elaborate approaches such as the SSM method are really needed. The number of processors which should be included in a state enumeration model in order to cover all possible sequences of memory accesses and all possible errors is indicated by the number of different base machines observed during the SSM expansion. Since the final set of essential states (collecting base machines in different states) reported in the SSM represents all possible states the system can be in at any time instance, essential states with a maximal number of base machines in different states represent the most complex case. In the verification using SSM, the most complex essential states consisted of 25 base machines in different states. This means that a system model of at least 25 processors is required to obtain a 100% confidence of error coverage for a state enumeration method. From Table 4, in the case of Murφ-ns), we observe (roughly) a 30 times increase in the size of the search space each time one more process is added to the model. If this trend continues up to 25 processors, an approximation of the asymptotic size of the search space is of the order of 10^{37} for a model with 25 processors. The time and the memory space needed by a verification of such complexity is prohibitive on any existing machine.

The SSM method has another advantage in the detection of livelocks. Because the number of global states reported is relatively small (123 states in this case), the time complexity of checking the connectivity of the global state transition diagram is more manageable than in Murφ.

7 Conclusion

We have presented and verified a directory-based cache coherence protocol for non-FIFO networks. By adding transient states, the protocol eliminates the need to acknowledge all messages as normally required to avoid races among messages between memory and processors. We do not claim that this protocol is optimal. However, this study demonstrates that the number of

messages exchanged in the protocol can be reduced without compromising correctness.

The verification of the protocol was done by the Murφ system and the SSM method. Whereas the complexity of the state space search (for a 100% confidence of error coverage) is of the order of 10^{37} for a system model of at least 25 processors, the SSM method symbolically characterized the entire state space by 123 global states, and verified the protocol in 25,631 state searches within three minutes. Before we obtained a correct design, several errors were found, including *unspecified message reception* (states in which the protocol receives unspecified messages) and *livelock* (the protocol circles around a loop of states indefinitely).

Generally speaking, from the study, we found that the Murφ system is effective in verifying small-scale systems with manageable complexity. The largest number of processors successfully included in the model was five with about 40 hours of execution time and 30M bytes of memory usage. When more processors were added, both the time and space complexity increase exponentially. The more important finding is that the conventional state enumeration method (Murφ-ns) performs as well as the method exploiting system symmetry (Murφ-s) for small-scale system models. Occasionally, because of extra overhead in exploiting the system symmetry, we observed that the Murφ-s system may take more verification time than the Murφ-ns system. The livelock error of Section 5.1 was not detected by the Murφ system but it was detected by the SSM method.

Overall, the SSM method offers three advantages. First, it overcomes the state explosion problem plaguing state enumeration methods. Second, since the entire global state space is symbolically represented by a small number of essential states, the time complexity of checking the connectivity of the global state transition diagram (needed for livelock detection) is more manageable than in traditional state enumeration methods with a large global state transition diagram. Third, it is also unique in the sense that it verifies the protocol for *any* system size.

Whereas classical state enumeration approaches will probably be sufficient to verify protocols for systems with small numbers of processors, methods based on symbolic state representations such as SSM will be critical in the future for the design of complex protocols in large-scale multiprocessors.

Acknowledgment

We want to acknowledge the contributions of David L. Dill and C. Norris Ip who provided invaluable information on the Murφ system.

References

[1] Agarwal, A., et al., "An Evaluation of Directory Schemes for Cache Coherence", *Proc. of the 15th Int'l Symposium on Computer Architecture, June 1988,* pp. 280-289.

[2] Archibald, J., "The Cache Coherence Problem in Shared-Memory Multiprocessors", Ph.D Dissertation, University of Washington, Feb. 1987.

[3] Archibald, J. and Baer, J.-L., "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Trans. on Computer Systems*, Vol.4, No.4, Nov. 1986, pp. 273-298.

[4] Archibald, J. and Baer, J.-L., "An Economic Solution to the Cache Coherence Problem", *Proc. of the 11th Int'l Symposium on Computer Architecture*, June 1984, pp. 355-362.

[5] Baer, J.-L. and Girault, C., "A Petri Net Model for a Solution to the Cache Coherence Problem", *Proc. of the 1st Conf. on Supercomputing Systems*, 1985, pp. 680-689.

[6] Browne, M.C., Clarke, E.M. and Grumberg, O., "Reasoning about Networks with Many Identical Finite State Processes", *Information and Computation 81, 13-31 (1989).*

[7] Censier, L.M. and Feautrier, P., "A new solution to coherence problems in multicache systems", *IEEE Trans. on Computers*, Vol. C-27, No. 12, Dec. 1978, pp. 1112-1118.

[8] Chaiken, D., et al., "Directory-Based Cache Coherence in Large Scale Multiprocessors", *IEEE Computer*, Vol. 23, No. 6, pp. 49-9, June 1990.

[9] Coudert, O., Madre, J.C. and Berthet, C., "Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams", *Computer-Aided Verification*, Springer-Verlag, 1990.

[10] Danthine, A.S., "Protocol Representation with Finite-State Models", *IEEE Trans. on Communications*, Vol. COM-28, No. 4, Apr. 1980, pp. 632-642.

[11] Dill, D.L., et al., "Protocol Verification as a Hardware Design Aid", *Int'l Conf. on Computer Design: VLSI in Computers and Processors*, Oct. 1992, pp. 522-525.

[12] Hoare, C.A.R., "Communicating Sequential Processes", *Commun. ACM*, Vol. 21, No. 8, Aug. 1978, pp. 666-677.

[13] Holzmann, G.J., "Algorithms for Automated Protocol Verification", *AT\&T Technical Journal*, Jan./Feb., 1990.

[14] Ip, C. N. and Dill, D.L., "Better Verification Through Symmetry", *Proc. of the IFIP WG10.2 Conf. on Computer Hardware Description Languages and their Applications*, Apr. 1993, pp. 87-100.

[15] Kakuda, Y., Wakahara, Y. and Norigoe, M., "An Acyclic Expansion Algorithm for Fast Protocol Verification", *IEEE Trans. on Software Engineering*, Vol. 14, No. 8, Aug. 1988, pp. 1059-1070.

[16] Kurshan, R.P. and McMillan, K., "A Structural Induction Theorem for Processes", *Eight Principles of Distributed Computing*, 1989, pp. 239-247.

[17] Lenosky, D., *et al.*, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor", *Proc. of the 17th Int'l Symposium on Computer Architecture*, June 1990, pp. 148-159.

[18] Loewenstein, P. and Dill, D.L., "Verification of a Multiprocessor Cache Protocol using Simulation Relations and Higher-Order Logic", *Computer-Aided Verification*, Springer-Verlag, 1990.

[19] McMillan, K.L. and Schwalbe, J., "Formal Verification of the Gigamax Cache Consistency Protocol", *Proc. of the ISSM Int'l Conf. on Parallel and Distributed Computing,* Oct. 1991.

[20] McMillan, K.L., "Symbolic Model Checking: An Approach to the State Explosion Problem", Ph.D Dissertation, Carnegie Mellon University, May 1992.

[21] Nanda, A.K. and Bhuyan, L.N., "A Formal Specification and Verification Technique for Cache Coherence Protocols", *Proc. of the 1992 Int'l Conf. on Parallel Processing*, pp. I-22-I-26.

[22] O'Krafka, B.W. and Newton, A.R., "An Empirical Evaluation of Two Memory-Efficient Directory Methods", *Proc. of the 17th Annual Int'l Symp. on Computer Architecture*, June 1990.

[23] Pong, F. and Dubois, M., "The Verification of Cache Coherence Protocols", *Proc. of the 5th Annual Symp. on Parallel Algorithm and Architecture,* June 1993, pp. 11-20.

[24] Pong, F. and Dubois, M., "Correctness of a Directory-Based cache Coherence Protocol: Early Experience", *Proc. of the 5th Annual Symp. on Parallel and Distributed Processing*, Dec. 1993, pp. 37-44.

[25] Rudolf, L. and Segall, Z., "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors", *Proc. of the 11th Int'l Symposium on Computer Architecture, June 1984,* pp. 340-347.

[26] Scheurich, C. and Dubois, M., "Correct Memory Operation of Cache-Based Multiprocessors", *Proc. of the 14th Int'l Symposium on Computer Architecture*, June, 1987, pp.234-243.

[27] Stenström, P., "A Survey of Cache Coherence Schemes for Multiprocessors", *IEEE Computer*, Vol. 23, No. 6, pp. 49-9, June 1990.

[28] Yen, W.C., Yen, W.L. and Fu, K.-S., "Data Coherence Problem in a Multicache System", *IEEE Trans. on Computers*, Vol. C-34, No. 1, Jan. 1985.

[29] Zafiropulo, P., West, C.H., Rudin, H., *et al.,* "Towards Analyzing and Synthesizing Protocols", *IEEE Trans. on Communications,* Vol. COM-28, No. 4, Apr. 1980, pp. 651-660.

Appendix A The Cache Algorithm

```
1:  Proc_i :: *[ Cache_i!('R', addr, ∅); Cache_i?data [] Cache_i!('W', addr, data); Cache_i?wait ]
2:
3:  Cache_i ::
4:  *[ Proc_i?(op, a, d); /* memory access requests from processor */
5:      blk.no:=BlockNo(a); /* the block contains location a */
6:      [ op='R' →
7:          [ Cache_i.st[blk.no]=O ∨ Cache_i.st[blk.no]=S → Proc_i!Cache.m[a] /* hit */
8:          [] Cache_i.st[blk.no]=I → Replacement_i!'go'; Cache_i.st[blk.no]:=RMP; SCH_i!('ReqSC', blk); /* miss */
9:          ]
10:     [] op='W' →
11:         [ Cache_i.st[blk.no]=O → Cache_i.m[a]:=d; Proc_i!'resume'; /* write hit on Owner copy */
12:         [] Cache_i.st[blk.no]=S → Cache_i.st[blk.no]:=WHP; SCH_i!('ReqO', blk); /* on Shared copy */
13:         [] Cache_i.st[blk.no]=I → Replacement_i!'go'; Cache_i.st[blk.no]:=WMP; SCH_i!('ReqOC', blk); /* miss */
14:         ]
15:     ]
16: [] RCH_i?(rmsg, blk); /* receiving messages */
17:     [ rmsg='Inv' →
18:         [ Cache_i.st[blk.no]=I → SCH_i!('IAck', blk); /* ack. an invalidation */
19:         [] Cache_i.st[blk.no]=S → Cache_i.st[blk.no]:=I; SCH_i!('IAck', blk);
20:         [] Cache_i.st[blk.no]=RMP → Cache_i.st[blk.no]:=TxSI; /* entering transient state */
21:         [] Cache_i.st[blk.no]=WMP → SCH_i!('IAck', blk);
22:         [] Cache_i.st[blk.no]=WHP → Cache_i.st[blk.no]:=WMP; SCH_i!('IAck', blk);
23:         ]
24:     [] rmsg='InvO' →
25:         [ Cache_i.st[blk.no]=I → SCH_i!('SAck', blk); /* synchronizing */
26:         [] Cache_i.st[blk.no]=O → Cache_i.st[blk.no]:=I; /* write back */
27:                                   blk.data:=Cache_i.m[blk.no]; SCH_i!('DOxMU', blk);
28:         [] Cache_i.st[blk.no]=RMP → SCH_i!('SAck', blk);
29:         [] Cache_i.st[blk.no]=WMP ∨ Cache_i.st[blk.no]=WHP → Cache_i.st[blk.no]:=TxOI; /* transient */
30:         ]
31:     [] rmsg='UpdM' →
32:         [ Cache_i.st[blk.no]=I → SCH_i!('SAck', blk); /* synchronizing */
33:         [] Cache_i.st[blk.no]=O → Cache_i.st[blk.no]:=S; /* write back */
34:                                   blk.data:=Cache_i.m[blk.no]; SCH_i!('DxM', blk);
35:         [] Cache_i.st[blk.no]=RMP → SCH_i!('SAck', blk);
36:         [] Cache_i.st[blk.no]=WMP ∨ Cache_i.st[blk.no]=WHP → Cache_i.st[blk.no]:=TxOS; /* transient */
37:         ]
38:     [] rmsg='O-ship' →
39:         [ Cache_i.st[blk.no]=WHP → Cache_i.st[blk.no]:=O; Cache_i.m[a]:=d; Proc_i!'resume';
40:         [] Cache_i.st[blk.no]=TxOI → Cache_i.m[a]:=d; blk.data:=Cache_i.m[blk.no]; Cache_i.st[blk.no]:=I;
41:                                      SCH_i!('DOxMU', blk); Proc_i!'resume';
42:         [] Cache_i.st[blk.no]=TxOS → Cache_i.m[a]:=d; blk.data:=Cache_i.m[blk.no]; Cache_i.st[blk.no]:=S;
43:                                      SCH_i!('DxM', blk); Proc_i!'resume';
44:         ]
45:     [] rmsg='Data' →
46:         [ Cache_i.st[blk.no]=RMP → Cache_i.st[blk.no]:=S; Cache_i.m[blk.no]:=blk.data; Proc_i!C_i.m[a];
47:         [] Cache_i.st[blk.no]=WMP → Cache_i.st[blk.no]:=O; Cache_i.m[blk.no]:=blk.data;
48:                                     Cache_i.m[a]:=d; Proc_i!'resume';
49:         [] Cache_i.st[blk.no]=TxOI → Cache_i.m[blk.no]:=blk.data; Cache_i.m[a]:=d; blk.data:=Cache_i.m[blk.no];
50:                                      Cache_i.st[blk.no]:=I; Proc_i!'resume'; SCH_i!('DOxMU', blk);
51:         [] Cache_i.st[blk.no]=TxSI → Cache_i.m[blk.no]:=blk.data; Proc_i!C_i.m[a]; Cache_i.st[blk.no]:=I;
52:                                      SCH_i!('IAck', blk);
53:         [] Cache_i.st[blk.no]=TxOS → Cache_i.m[blk.no]:=blk.data; Cache_i.m[a]:=d; blk.data:=Cache_i.m[blk.no];
54:                                      Cache_i.st[blk.no]:=S; Proc_i!'resume'; SCH_i!('DxM', blk);
55:         ]
```

FIGURE A.1. Cache Algorithm.

```
56:       [] rmsg='NAck' →
57:          [  Cache_i.st[blk.no]=RMP → SCH_i!('ReqSC', blk); /* re-issue the request */
58:          [] Cache_i.st[blk.no]=WMP → SCH_i!('ReqOC', blk);
59:          [] Cache_i.st[blk.no]=WHP → SCH_i!('ReqO', blk);
60:          [] Cache_i.st[blk.no]=TxOI → Cache_i.st[blk.no]:=WMP; SCH_i!('SAck', blk); SCH_i!('ReqOC', blk);
61:          [] Cache_i.st[blk.no]=TxSI → Cache_i.st[blk.no]:=RMP; SCH_i!('IAck', blk); SCH_i!('ReqSC', blk);
62:          [] Cache_i.st[blk.no]=TxOS → Cache_i.st[blk.no]:=WMP; SCH_i!('SAck', blk); SCH_i!('ReqOC', blk);
63:          ]
64:       ]
65:  ]

66:  Replacement_i::
67:  [ Cache_i?start; eblk.no:=Evict(); /* replacement */
68:       [  Cache_i.st[eblk.no]=O → Cache_i.st[eblk.no]:=I; eblk.data:=Cache_i.m[eblk.no]; SCH_i!('DOxMR', eblk);
69:       [] Cache_i.st[eblk.no]=S → Cache_i.st[eblk.no]:=I;
70:        ]
71:  ]

72:  SCH_i :: *[  Cache_i?(smsg, sblk); InsertSCH_i(smsg, sblk); /* record messages sent by the cache */
73:             [] ¬EmptySCH_i → GetSCH_i(smsg, sblk); /* get the message emerging from the channel */
74:                               Memory!(smsg, sblk); /* send message to the memory*/
75:             ]

76:  RCH_i :: *[  Memory?(rmsg, rblk); InsertRCH_i(rmsg, rblk); /* record messages sent by the memory */
77:             [] ¬EmptyRCH_i → GetRCH_i(rmsg, rblk); /* get the message emerging from the channel*/
78:                               Cache_i!(rmsg, rblk); /* send message to the cache */
79:             ]

80:  Memory ::
81:  *[SCH_i?(cmd, dblk);
82:       [ Memory.st[dblk.no]=free → /* directory entry is free */
83:          [ cmd='ReqSC' →
84:             [ dirtybit=1 → Memory.st[dblk.no]:=XData; reqc:=i; RCH_owner!('UpdM', dblk);
85:             [] dirtybit=0 → presencebit[i]:=1; dblk.data:=Memory.m[dblk.no]; RCH_i!('Data', dblk);
86:             ]
87:          [] cmd='ReqO' ∧ presencebit[i] ≠ 1 → RCH_i!('NAck', dblk); /* ReqOC expected */
88:          [] (cmd='ReqO' ∧ presencebit[i] = 1) ∨ cmd='ReqOC' →
89:             [ for all (j ≠ i ∧ presencebit[j]=0) → /* no other cached copies exist */
90:                   dirtybit:=1;
91:                   [  cmd='ReqO' → RCH_i!('O-ship', dblk);
92:                   [] cmd='ReqOC' → presencebit[i]:=1; dblk.data:=Memory.m[dblk.no]; RCH_i!('Data', dblk);
93:                   ]
94:             [] ¬(for all (j ≠ reqc ∧ presencebit[j]=0)) → /* there exists some cached copy */
95:                   reqc:=i;
96:                   [  cmd='ReqO' → Memory.st[dblk.no]:=XOwn;
97:                                   for all (j ≠ reqc ∧ presencebit[j]=1)  RCH_j!('Inv', dblk);
98:                   [] cmd='ReqOC' → Memory.st[dblk.no]:=XOwnC;
99:                                   [ dirty=1 → RCH_owner!('InvO', dblk); /* invalidate owner */
100:                                  [] dirty=0 → for all (j ≠ reqc ∧ presencebit[j]=1) RCH_j!('Inv', dblk);
101:                                  ]
102:                  ]
103:          ]
104:       [] cmd='DOxMR' → presencebit[i]:=0; dirtybit:=0; Memory.m[dblk.no]:=dblk.data;
105:       ]
106:
```

FIGURE A.1. Cache Algorithm (cont'd).

```
107:    [ ¬(Memory.st[dblk.no]=free) → /* directory entry is locked */
108:        [  (cmd='ReqSC' ∨ cmd='ReqO' ∨ cmd='ReqOC') → RCH_i!('NAck', dblk); /* reject further requests */
109:        [] cmd='DxM' →
110:            Memory.st[dblk.no]=XData → dirtybit:=0; Memory.m[dblk.no]:=dblk.data; RCH_reqc!('Data', dblk);
111:                                                    presencebit[reqc]:=1; reqc:=null; Memory.st[dblk.no]:=free;
112:        [] cmd='DOxMR' →
113:            presencebit[i]:=0; dirtybit:=0; /* owner no longer exists */
114:            Memory.m[dblk.no]:=dblk.data; /* update memory copy */
115:            [ Memory.st[dblk.no]=XData → Memory.st[dblk.no]:=Synch2; /* synchronization state */
116:            [] Memory.st[dblk.no]=XOwnC → Memory.st[dblk.no]:=Synch1;
117:            [] Memory.st[dblk.no]=Synch1 →
118:                        RCH_reqc!('Data', dblk); /* supply data */
119:                        presencebit[reqc]:=1; reqc:=null; /* requesting cache is reset */
120:                        dirtybit:=1; /* new owner */ Memory.st[dblk.no]:=free;
121:            [] Memory.st[dblk.no]=Synch2 →
122:                        RCH_reqc!('Data', dblk); /* supply data */
123:                        presencebit[reqc]:=1; reqc:=null; /* requesting cache is reset */
124:                        Memory.st[dblk.no]:=free;
125:            ]
126:        [] cmd='DOxMU' →
127:            Memory.st[dblk.no]=XOwnC →
128:                        Memory.m[dblk.no]:=dblk.data; /* update memory copy */
129:                        RCH_reqc!('Data', dblk); /* supply data */
130:                        presencebit[i]:=0; presencebit[reqc]:=1; /* new owner */
131:                        reqc:=null; /* requesting cache is reset */ Memory.st[dblk.no]:=free;
132:        [] cmd='IAck' → presencebit[i]:=0;
133:                        for all (j ≠ reqc ∧ presencebit[j]=0) → /* no other cached copies */
134:                            [ [  Memory.st[dblk.no]=XOwn → dirtybit:=1; RCH_reqc!('O-ship', dblk);
135:                              [] Memory.st[dblk.no]=XOwnC → dirtybit:=1; /* new Owner */
136:                                                            dblk.data:=Memory.m[dblk.no];
137:                                                            presencebit[reqc]=1;
138:                                                            RCH_reqc!('Data', dblk);
139:                              ]
140:                              reqc:=null; Memory.st[dblk.no]:=free;
141:                            ]
142:        [] cmd='SAck' →
143:            presencebit[i]:=0; dirtybit:=0;
144:            [ Memory.st[dblk.no]=XData →
145:                        Memory.st[dblk.no]:=Synch2; /* enter a synchronization state */
146:            [] Memory.st[dblk.no]=XOwnC →
147:                        Memory.st[dblk.no]:=Synch1; /* enter a synchronization state */
148:            [] Memory.st[dblk.no]=Synch1 →
149:                        dblk.data:=Memory.m[dblk.no]; RCH_reqc!('Data', dblk); /* supply data */
150:                        presencebit[reqc]:=1; reqc:=null; /* requesting cache is reset */
151:                        dirtybit:=1; /* new owner */ Memory.st[dblk.no]:=free;
152:            [] Memory.st[dblk.no]=Synch2 →
153:                        dblk.data:=Memory.m[dblk.no]; RCH_reqc!('Data', dblk); /* supply data */
154:                        presencebit[reqc]:=1; reqc:=null; /* requesting cache is reset */
155:                        Memory.st[dblk.no]:=free;
156:            ]
157:        ]
158:    ]
159: ]
```

FIGURE A.1. Cache Algorithm (cont'd).

Appendix B Proof of Theorem 1

Lemma 1. *The aggregation process is monotonic, that is, if* $q^{r_{11}} \le q^{r_{21}}$ *and* $q^{r_{12}} \le q^{r_{22}}$, *then we have* $\{q^{r_{11}}, q^{r_{12}}\} = q^{r_1} \le \{q^{r_{21}}, q^{r_{22}}\} = q^{r_2}$, *where* $q \in Q_{\mathcal{BM}}$, $r_i \in [0, 1, +, *, v]$

Proof: Immediate from the Aggregation rule in Section 5.3.

Lemma 2. *The immediate successor* $\overline{S}_1$ *originated from state*

$$S_1 = (q_1^{r_1}, q_2^{r_2}, \ldots, q_{i-1}^{r_{i-1}}, q_i^{i=1}, q_{i+1}^{r_{i+1}}, \ldots, q_n^{r_n}, q_{\mathcal{MM}1})$$

is contained by state $\overline{S}_2$ *originated from state*

$$S_2 = (q_1^{\bar{r}_1}, q_2^{\bar{r}_2}, \ldots, q_{i-1}^{\bar{r}_{i-1}}, q_i^{1/+/*/v}, q_{i+1}^{\bar{r}_{i+1}}, \ldots, q_n^{\bar{r}_n}, q_{\mathcal{MM}2}),$$

if $r_j = \bar{r}_j$ for all $j \neq i$, if the same event $t \in (\Sigma_r \cup \Sigma_s)$ is applied to S_1 and S_2, and if $q_{\mathcal{MM}1} = q_{\mathcal{MM}2}$.

Proof: The proof is direct for the case that the event t is applied to base machines in state q_j, where $j \neq i$. We only need to consider the effect of applying t to base machines in state q_i in S_1 and S_2. To simplify the notation, all equivalence classes q_j ($j \neq i$) are denoted as Q. There are three cases.

Case (a). $t \in \Sigma_r$ (see Table 1) indicates that the cache receives a memory-to-cache command. We have the following transitions (by application of the N-steps rule), provided $q_i \rightarrow^t q_j$:

(a.1). $S_1=(Q, q_i, q_{\mathcal{MM}1}) \rightarrow^t \overline{S}_1=(Q, q_i^0, \{q_j, q_j^{r_j}\}, q_{\mathcal{MM}1})$,

(a.2). $S_2=(Q, q_i^+, q_{\mathcal{MM}2}) \rightarrow^t \overline{S}_2=(Q, q_i^*, \{q_j^+, q_j^{\bar{r}_j}\}, q_{\mathcal{MM}2})$, and

(a.3). $S_2=(Q, q_i^{*/v}, q_{\mathcal{MM}2}) \rightarrow^t \overline{S}_2=(Q, q_i^v, \{q_j^*, q_j^{\bar{r}_j}\}, q_{\mathcal{MM}2})$.

By lemma 1 (the aggregation process is monotonic) and the antecedent of this lemma ($r_j = \bar{r}_j$), we know that $\overline{S}_1 \subseteq \overline{S}_2$.

Case (b). $t \in \Sigma_s$ (see Table 1) corresponds to a memory access by a processor. The same argument as in case (a) can be made.

Case (c). $t \in \Sigma_s$ and the expansion step is taken by the memory that responds to a cache-to-memory command. The following argument is based on two assumptions: (1) the protocol allows one and only one owner in the system at any time, (2) the owner is responsible for supplying the most up-to-date data to remote requests. First of all, let's consider the following case:

(c.1). t is a request for a data copy emerging from the sending channel of a base machine in state q_i when an owner exists. We have

(c.1.1). $S_1=(Q, q_{own}, q_i, q_{MM1}) \rightarrow^t \overline{S}_1=(Q, q_{own}', q_i^0, q_{req}, q_{MM1}')$,

(c.1.2). $S_2=(Q, q_{own}, q_i^+, q_{MM2}) \rightarrow^t \overline{S}_2=(Q, q_{own}', q_i^*, q_{req}, q_{MM2}')$, and

(c.1.3). $S_2=(Q, q_i^{*/v}, q_{MM2}) \rightarrow^t \overline{S}_2=(Q, q_{own}', q_i^v, q_{req}, q_{MM2}')$,

where one base machine from class q_i is recorded as the requester, and $q_{own} \rightarrow q_{own}'$ means that a request is sent to the owner. We also know that $q_{MM1}' = q_{MM2}'$, because the memory responds to the same type of request, thus, $\overline{S}_1 \subseteq \overline{S}_2$.

(c.2). For all other cases, similar arguments as for (c.1) can be made. Here, we limit ourselves to the most complex case corresponding to a request for an exclusive copy. Also, base machines in state q_i are the only element in the Invalidation-Set and the transition t is caused by an acknowledgment message (such as IAck) in response to an invalidation request. Provided ${}^1q_i \rightarrow^t {}^0q_j$ (the acknowledgment message is removed from the sending channel of q_i causing the reset of the presence bit), we have

(c.2.1). $S_1=({}^0Q, {}^1q_i, q_{req}, q_{MM1}) \rightarrow^t \overline{S}_1=({}^0Q, {}^0q_j, q_{req} \rightarrow q_{own}, q_{MM1}')$,

(c.2.2). $S_2=({}^0Q, {}^1q_i^+, q_{req}, q_{MM2}) \rightarrow^t S_2'=({}^0Q, {}^1q_i^*, {}^0q_j^+, q_{req}, q_{MM2}) \rightarrow^t$

$S_2''=({}^0Q, {}^1q_i^v, {}^0q_j^+, q_{req}, q_{MM2}) \rightarrow^t \overline{S}_2=({}^0Q, {}^0q_j^+, q_{req} \rightarrow q_{own}, q_{MM2}')$

and $\overline{S}_2'=({}^0Q, {}^1q_i^v, {}^0q_j^+, q_{req}, q_{MM2})$, by the non-determinism of the progress rule.

(c.2.3). $S_2=({}^0Q, {}^1q_i^*, q_{req}, q_{MM2}) \rightarrow^t S_2'=({}^0Q, {}^1q_i^v, {}^0q_j^*, q_{req}, q_{MM2}) \rightarrow^t$

$\overline{S}_2=({}^0Q, {}^0q_j^*, q_{req} \rightarrow q_{own}, q_{MM2}')$ and $\overline{S}_2'=({}^0Q, {}^1q_i^v, {}^0q_j^*, q_{req}, q_{MM2})$.

(c.2.4). $S_2=({}^0Q, {}^1q_i^v, q_{req}, q_{MM2}) \rightarrow^t \overline{S}_2=({}^0Q, {}^0q_j^*, q_{req} \rightarrow q_{own}, q_{MM2}')$ and $\overline{S}_2'=({}^0Q, {}^1q_i^v, {}^0q_j^*, q_{req}, q_{MM2})$.

We also know that $q_{MM1}' = q_{MM2}'$ because the memory responds to the same type of pending request; thus, $\overline{S}_1 \subseteq \overline{S}_2$. For cases (c.2.2-4), we have seen that it may take several expansion steps to end up in a state $\overline{S}_2$ containing $\overline{S}_1$, which does not affect the validity of the methodology.

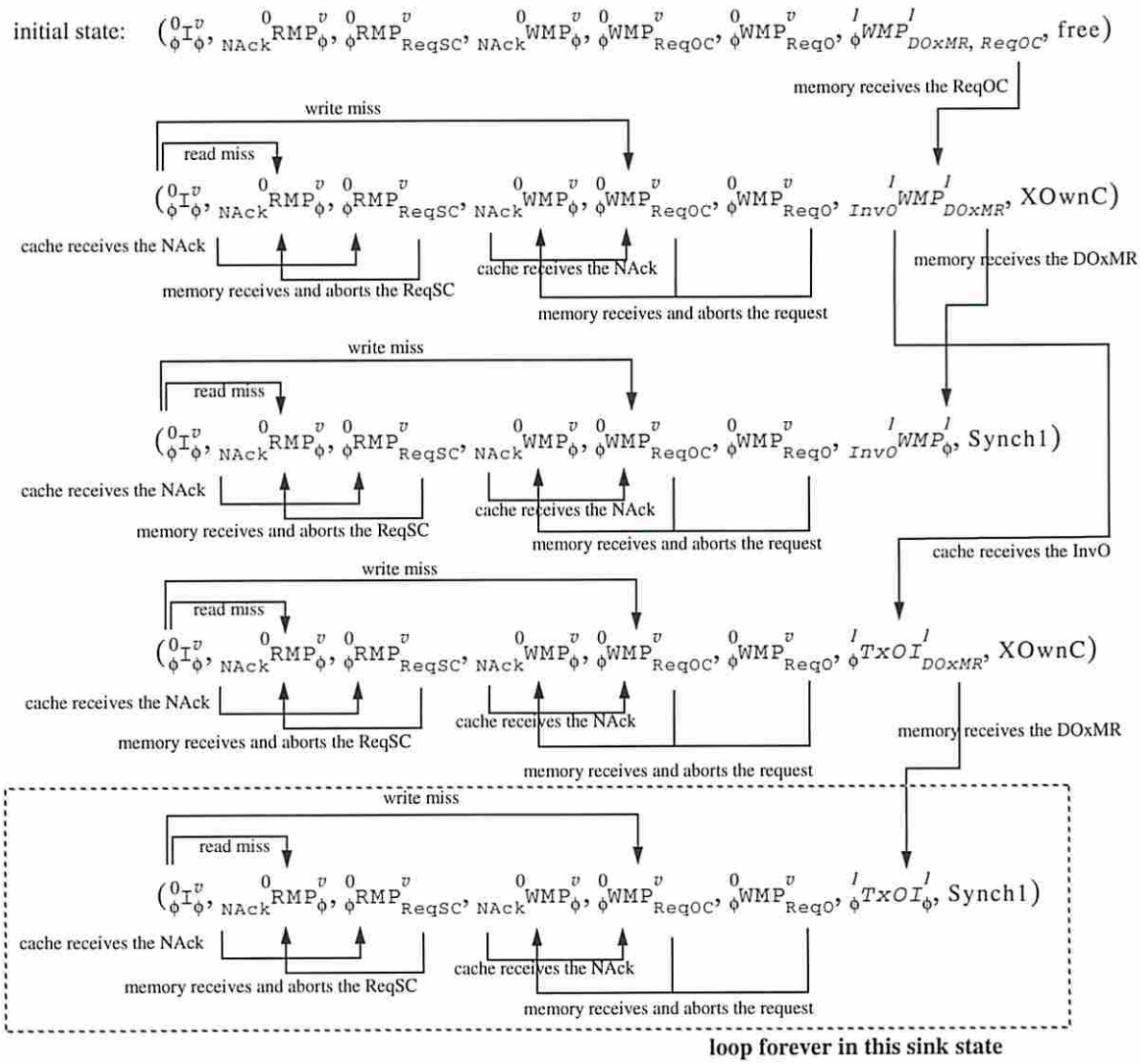
Lemma 3. *The claim $\overline{S}_1 \subseteq \overline{S}_2$ holds if $S_1 \subseteq S_2$, that is, $r_j \leq \overline{r}_j$ for all j and $q_{MM1}=q_{MM2}$.*

Proof: The result extends the conclusion of Lemma 2 and the proof is similar.

Finally, by a recursive induction from Lemma 3, we can prove Theorem 1.

Appendix C Constituent Global States in the Found Livelock

In the following, we present the livelock condition found in the protocol verification. The owner is italicized (this cache is also the requester for an exclusive copy in this case). Directed arrows specify transitions.



Note that the last *sink* state above actually represents a set of states which constitute a closed loop in a traditional state enumeration method.