Correctness of a Directory-Based Cache Coherence Protocol: Early Experience

Fong Pong and Michel Dubois

CENG Technical Report 93-23

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

May 1993

Correctness of a Directory-Based Cache Coherence Protocol: Early Experience

Fong Pong and Michel Dubois

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

May 1993

Abstract

Cache coherence protocols of increasing complexities call for automated verification tools which are both efficient and reliable. Most current approaches can only verify protocols at a high level of abstraction, and the model size is limited to a small number of interacting processes. Using a simple full-map directory scheme as example, we show that the verification of a simple protocol becomes overwhelmingly complex when implementation details are taken into account. One way to deal with the complexity is to impose conservative *handshaking* rules such as acknowledging every single message between caches and memory. Such a conservative approach slows down every transaction in order to avoid race conditions, which are relatively rare. The other approach explored in this paper is to apply verification techniques to the protocol without acknowledgements in order to determine the minimum set of messages needed for correctness. A new verification technique which is extremely efficient and is independent of the model size is applied and several non-obvious problems affecting the correctness of a protocol design are identified by the verification procedure.

Correctness of a Directory-Based Cache Coherence Protocol: Early Experience

Abstract

Cache coherence protocols of increasing complexities call for automated verification tools which are both efficient and reliable. Most current approaches can only verify protocols at a high level of abstraction, and the model size is limited to a small number of interacting processes. Using a simple full-map directory scheme as example, we show that the verification of a simple protocol becomes overwhelmingly complex when implementation details are taken into account. One way to deal with the complexity is to impose conservative *handshaking* rules such as acknowledging every single message between caches and memory. Such a conservative approach slows down every transaction in order to avoid race conditions, which are relatively rare. The other approach explored in this paper is to apply verification techniques to the protocol without acknowledgements in order to determine the minimum set of messages needed for correctness. A new verification technique which is extremely efficient and is independent of the model size is applied and several non-obvious problems affecting the correctness of a protocol design are identified by the verification procedure.

1 .0 Introduction

Cache coherence is one of the most important aspects of a shared-memory multiprocessor system. A coherence protocol maintains a globally consistent view of memory among processors. *Snooping* protocols [2] rely on broadcasting updates or invalidations to keep data copies consistent and therefore, their applicability is inherently limited to systems with interconnection supporting efficient broadcast, mostly shared-bus systems. It is well-known that a shared bus is *non-scalable* in the sense that a few processors can saturate it even when private caches reduce the memory traffic.

Directory-based protocols [3, 6, 23] are more apt at enforcing coherence in systems with a large number of processors. A directory keeps track of caches with a valid data copy so that coherence messages are sent to individual processors rather than broadcast to all processors, removing the need for an efficient broadcast medium. Although many performance studies of directory-based schemes have been published in the literature [1, 7, 18], protocol correctness issues are usually left out. Because of their more complex structures and of the difficulty of verifying protocols at lower implementation levels, automated verification tools based on efficient and reliable procedures are sorely needed.

In this paper we first describe an implementation of the *full-map* directory scheme proposed by Censier and Feautrier [6] down to the message-passing level. Even though the protocol is very simple at its behavior specification level the complexity of even the simplest implementation is overwhelming. One way to deal with the complexity is to impose conservative *handshaking* rules such as acknowledging every single message between caches and memory. Such a conservative approach slows down every transaction in order to avoid race conditions, which are relatively rare. The other approach explored in this paper is to apply verification techniques to the protocol without acknowledgements in order to determine the minimum set of messages needed for correctness. We demonstrate the successful application of a new verification technique [19] to the protocol implementation and discover several problems, which make a correct design difficult.

Our method is motivated by the observation that complex systems are often very *regular* and *symmetric*. In particular, in a cache-based multiprocessor, the behavior of all caches is characterized by the same finite-state process. Our symbolic state representations [19] exploit this regularity to yield a very efficient state expansion process without the *state explosion* problem [9, 11] plaguing other approaches. Additionally, the verification procedure is independent of the system size and is totally reliable. All 'possible' states are explored, as opposed to approaches verifying a '*small*' system; in these approaches design errors may appear for system sizes beyond the manageable model size and go undetected in the verification of the small model.

This paper is organized as follows. Section 2 briefly overviews previous work in the area of protocol verification. Section 3 provides a general and a more formal description of the protocol verified in this paper. In Section 4, our verification technique is formalized, and the verification results are then discussed in Section 5. Finally, we comment on the prospects offered by our approach in the field of protocol verification.

2 .0 Previous Work

Several authors have addressed the problem of protocol verification in different ways. Baer and Girault [4] introduced a Petri Net model that comprehensively specifies the underlying hardware structure. Rudolf and Segal [20] gave a correctness proof of a snooping protocol by enumerating all possible scenarios of reads and writes;each cache is modeled as a finite state automaton and a *product machine* is a collection of *n* finite state automata. Nanda and Bhuyan [17] presented a similar approach based on the composition of communicating finite state machines and on state enumeration. For most of complex protocols, their approaches are not feasible because of the enumeration complexity and of the state space explosion problem.

To overcome the state explosion problems, McMillan and Schwalbe [16] evaluate the truth value of protocol correctness conditions in temporal logic without constructing the global state diagram explicitly. However, the transition relations among global states must be constructed, which itself is not feasible for most complex systems [8]. By using *higher-order* logic, Loewenstein and Dill [15] show the *correspondence* (*simulation relation*) between state machines representing an implementation and specification behavior at a high-level abstraction. Although the logic is powerful, finding the simulation relation is difficult and inefficient.

To overcome the limitation that only protocols with few number of interacting processes can be verified in practice, reasoning about protocols with large number of processes is currently a topic of active research. In [5], reasoning about systems with many identical finite state processes is facilitated by finding a *correspondence* between two (global state) structures M and M' representing the system 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 smaller *base* state machine equivalently. In practice, this method is not feasible for protocols with complex structure. First, discovery of the correspondence relation between two structures relies on unrealistic human ingenuity. Second, construction of the global state diagram of the *base* machine may be a difficult procedure itself.

Recent work [13] suggests the use of partial-order relations between 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 potentials but the construction of the invariant process in [13] is not automated and requires considerable ingenuity, which is unrealistic for complex protocols.

3 .0 Full-Map Directory-Based Cache Coherence Protocol

Although many protocols have been proposed [1, 3, 23], we consider Censier and Feautrier's write-invalidate *full-map* directory protocol [6] to illustrate the approach. Every memory block is associated with a *vector of presence bits*, each of which indicates whether or not a cache has a copy of the block. When an invalidation occurs, invalidations are only sent to caches with their presence bits set.

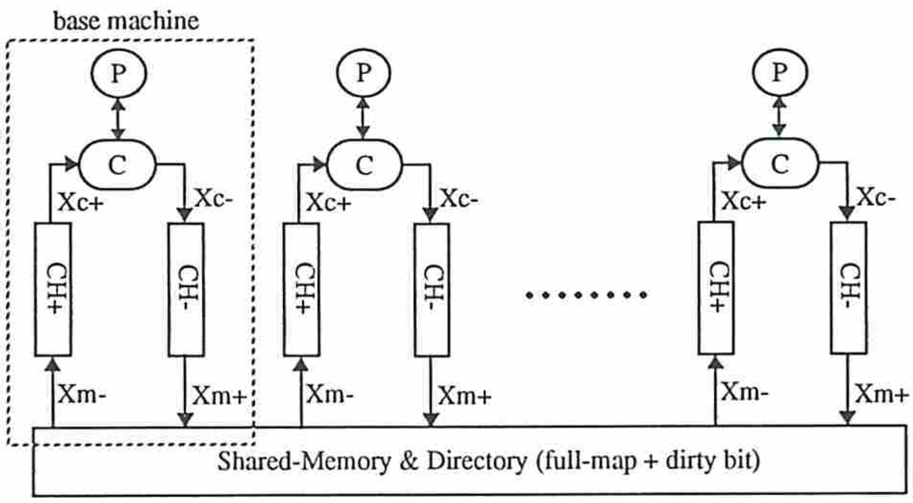

FIGURE 1. Architecture of a Directory-Based Coherence Protocol

The architecture, illustrated in Figure 1, consists of a central directory and many processor-cache connections. 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. In order to model the effect of memory event ordering, there is no restriction on the order in which messages are delivered, that is, messages may be received and consumed in an order different from the one in which they are issued. Each directory entry consists of a full-map presence bit vector and of an extra *dirty* bit indicating whether or not a dirty copy exists. Also, when a request being granted is in progress, the corresponding directory entry is locked, and is released when the coherent event is complete [18].

3.1 Cache States and Memory Commands

Caches can be in one of three states: *Invalid*, *Shared* (clean and potentially shared with other caches), and *Owner* (modified and only cached copy)[1]. Additionally, three transient states are required to model pending cache events in progress. They are *read-miss-pending*(*RMP*), *write-miss-pending*(*WMP*) and *write-hit-pending*(*WHP*). *RMP* and *WMP* should be clear, and *WHP* corresponds to the case that the local processor writes to a local clean copy and must invalidate remote copies first. For the rest of this paper, we also write *I*, *S* and *O* for *Invalid*, *Shared* and *Owner* to simplify the notations.

Caches and memory interact by two sets of *memory-to-cache* and *cache-to-memory* commands as summarized in Table 1. The 'Reject' message means that a request is aborted because a locked (busy) directory entry and must be retried. Alternatively the request could be queued at the memory module; this alternative can be easily modeled in the state expansion process by simply keeping the requests in the sending channels; requests are consumed only when the directory entry is free however.

TABLE 1. Memory Commands.

Type	Command	Functions
Memory To Cache(Σ_r)	Inv	Invalidate.
	UpdM	Request to update main memory copy. Receiving cache must be an Owner and changes to Shared state.
	Ownership	Ownership is transferred on the cache.
	Data	A data copy is supplied on the cache's request.
	Reject	Indicates that the request is rejected because of a locked directory entry.
Cache To Memory(Σ_s)	ReqSC	Request a Shared copy.
	ReqO	Request Ownership.
	ReqOC	Request Ownership and data copy.
	DxM	Owner cache updates main memory and relinquishes ownership on request. Owner cache transfers to Shared state.
	DOxM	Upon receiving an invalidation or being replaced, Owner updates main memory, relinquishes ownership and invalidates itself.
	Repl	Cache in Shared state executes a regular replacement.
	IAck	Acknowledge invalidation.

3.2 Cache Algorithm

The descriptions of the cache algorithm, given in Figure 2, is inspired by Hoare's CSP language [10], whose features are now briefly described. Indexed processes **Proc**,

1. Since not much implementation effort has ever been published in the literature, the protocol in this paper is only an empirical design. However, its descriptions follows general perceptions [6, 7, 18, 22].

Cache, **Replacement**, SCH, RCH, and **Memory** cooperatively specify the behavior of the protocol. A repetitive execution, indicated by a leading asterisk, 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 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.

```
1:  Proc_i :: *[ Cache_i!('R', addr, ∅); Cache_i?data [] Cache_i!('W', addr, data); Cache_i?wait ]
2:  Cache_i :: *[ Proc_i?(op, a, d); /* memory access requests from processor */
3:               blk.no:=BlockNo(a); /* the block contains location a */
4:                 [ op='R' →
5:                     [ Cache_i.st[blk.no]=O ∨ Cache_i.st[blk.no]=S → Proc_i!Cache.m[a] /* read hit */
6:                     [] Cache_i.st[blk.no]=I → Cache_i.st[blk.no]:=RMP; Replacement_i!'jumpstart';
7:                                               SCH_i!('ReqSC', blk); /* read miss */
8:                     ]
9:                 [] op='W' →
10:                    [ Cache_i.st[blk.no]=O → Cache_i.m[a]:=d; Proc_i!'done'; /* write hit on Owner copy */
11:                    [] Cache_i.st[blk.no]=S → Cache_i.st[blk.no]:=WHP; SCH_i!('ReqO', blk); /* on Shared copy */
12:                    [] Cache_i.st[blk.no]=I → Cache_i.st[blk.no]:=WMP; Replacement_i!'jumpstart';
13:                                              SCH_i!('ReqOC', blk); /* write miss */
14:                    ]
15:                ]
16:              [] RCH_i?(rmsg, blk); /* receive messages from receiving channel */
17:                [ rmsg='Inv' →
18:                    [ Cache_i.st[blk.no]=S → Cache_i.st[blk.no]:=I; SCH_i!('IAck', blk); /* ack. an invalidation */
19:                    [] Cache_i.st[blk.no]=O → Cache_i.st[blk.no]:=I; blk.data:=Cache_i.m[blk.no];
20:                                              SCH_i!('DOxM', blk); /* ack. by data & ownership */
21:                    ]
22:                [] rmsg='UpdM' → Cache_i.st[blk.no]=O → Cache_i.st[blk.no]:=S; blk.data:=Cache_i.m[blk.no];
23:                                                        SCH_i!('DxM', blk); /* update memory */
24:                [] rmsg='Ownership' → Cache_i.st[blk.no]=WHP → Cache_i.st[blk.no]:=O;
25:                                                                Cache_i.m[a]:=d; Proc_i!'done';
26:                [] rmsg='Data' → [ Cache_i.st[blk.no]=RMP → Cache_i.st[blk.no]:=S; Cache_i.m[blk.no]:=blk.data;
27:                                                             Proc_i!C_i.m[a];
28:                                 [] Cache_i.st[blk.no]=WMP → Cache_i.st[blk.no]:=O; Cache_i.m[blk.no]:=blk.data;
29:                                                             Cache_i.m[a]:=d; Proc_i!'done';
30:                                 ]
31:                [] rmsg='Reject' → [ Cache_i.st[blk.no]=RMP → SCH_i!('ReqSC', blk); /* re-send the request */
32:                                   [] Cache_i.st[blk.no]=WMP → SCH_i!('ReqOC', blk);
33:                                   [] Cache_i.st[blk.no]=WHP → SCH_i!('ReqO', blk);
34:                                   ]
35:                ]
36:              ]
37: Replacement_i:: [ Cache_i?start; eblk.no:=Evict(); /* replacement */
38:                     [ Cache_i.st[eblk.no]=O → Cache_i.st[eblk.no]:=I; eblk.data:=Cache_i.m[eblk.no]
39:                                               SCH_i!('DOxM', eblk); /* write back and release ownership */
40:                     [] Cache_i.st[eblk.no]=S → Cache_i.st[eblk.no]:=I; SCH_i!('Repl', eblk);
41:                     ]
42:                 ]
43: SCH_i :: *[ Cache_i?(smsg, sblk); InsertSCH_i(smsg, sblk); /* recording messages sent by cache */
44:            [] ¬EmptySCH_i → GetSCH_i(smsg, sblk); /* get the message emerging from the channel */
45:                             Memory!(smsg, sblk); /* propagating messages to memory*/
46:            ]
47: RCH_i :: *[ Memory?(rmsg, rblk); InsertRCH_i(rmsg, rblk); /* recording messages sent by memory */
48:            [] ¬EmptyRCH_i → GetRCH_i(rmsg, rblk); /* get the message emerging from the channel*/
49:                             Cache_i!(rmsg, rblk); /* propagating messages to cache */
50:            ]
```

FIGURE 2. Cache Algorithm.

```
Memory ::
    *[$SCH_i$?(cmd, dblk);
        [ Memory.st[dblk.no]=free → /* directory entry is free */
            [ cmd='ReqSC' →
                    [ dirtybit=1 → /* an Owner copy exists */
                            Memory.st[dblk.no]:=XData; reqc:=i;
                            $RCH_{owner}$!('UpdM', dblk); /* update memory copy */
                    [] dirtybit=0 → presencebit[i]:=1; dblk.data:=Memory.m[dblk.no];
                                $RCH_i$!('Data', dblk); /* memory provides the data */
                    ]
            [] cmd='ReqO' ∨ cmd='ReqOC' →
                    [ for all ($j \neq i \wedge$ presencebit[j]=0) → /* no other cached copies exist */
                        dirtybit:=1;
                        [  cmd='ReqO' → $RCH_i$!('Ownership', dblk);
                        [] cmd='ReqOC' → presencebit[i]:=1; dblk.data:=Memory.m[dblk.no];
                                        $RCH_i$!('Data', dblk);
                        ]
                    [] ¬(for all ($j \neq$ reqc $\wedge$ presencebit[j]=0)) → /* there exists some cached copies */
                        reqc:=i;
                        [  cmd='ReqO' → Memory.st[dblk.no]:=XOwn;
                        [] cmd='ReqOC' → Memory.st[dblk.no]:=XOwnC;
                        ]
                        for all ($j \neq$ reqc $\wedge$ presencebit[j]=1)  $RCH_j$!('Inv', dblk);
                    ]
            [] cmd='Repl' → presencebit[i]:=0;
            [] cmd='DOxM' → presencebit[i]:=0; dirtybit:=0; Memory.m[dblk.no]:=dblk.data;
            ]
        [ ¬(Memory.st[dblk.no]=free) → /* directory entry is locked */
            [  (cmd='ReqSC' ∨ cmd='ReqO' ∨ cmd='ReqOC') → $RCH_i$!('Reject', dblk);
            [] cmd='DxM' →  dirtybit:=0; /* owner no longer exists */
                        Memory.m[dblk.no]:=dblk.data; /* update memory copy */
                        Memory.st[dblk.no]=XData → $RCH_{reqc}$!('Data', dblk); /* supplies data */
                        presencebit[reqc]:=1; reqc:=null; /* requesting cache is reset */
                        Memory.st[dblk.no]:=free;
            [] cmd='DOxM' → dirtybit:=0; Memory.m[dblk.no]:=dblk.data; /* update memory copy */
                          presencebit[i]:=0; presencebit[reqc]:=1; /* Owner invalidated */
                          [ Memory.st[dblk.no]=XData → $RCH_{reqc}$!('Data', dblk); /* supplies data */
                          [] Memory.st[dblk.no]=XOwnC → dirtybit:=1; $RCH_{reqc}$!('Data', dblk);
                          ]
                          reqc:=null; Memory.st:=free;
            [] cmd='Repl' → presencebit[i]:=0;
            [] cmd='IAck' → presencebit[i]:=0;
                        for all ($j \neq$ reqc $\wedge$ presencebit[j]=0) → /* no other cached copies */
                        [ [  Memory.st[dblk.no]=XOwn → dirtybit:=1; $RCH_{reqc}$!('Ownership', dblk);
                          [] Memory.st[dblk.no]=XOwnC → dirtybit:=1; /* new Owner */
                                                    dblk.data:=Memory.m[dblk.no];
                                                    $RCH_{reqc}$!('Data', dblk);
                          ]
                          reqc:=null; Memory.st[dblk.no]:=free;
                        ]
            ]
        ]
    ]
```

FIGURE 2. Cache Algorithm (cont'd).

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

The cache algorithm is simple. A read hit or a write hit on an *Owner* copy do not generate any coherence event. A read miss first changes the local cache state to *RMP*, and a request for a *Shared* copy is sent down the sending channel. A write hit on a *Shared* copy initiates a *ReqO* request and changes the local state to *WHP*. A write miss requests an *Owner* copy with a *ReqOC* command and the local cache state is *WMP*. An access miss also calls for block replacement to make room for the new data block. The replacement procedure is simple if the cache is in the *Shared* state: a simple *Repl* message is sent to the memory to reset its corresponding presence bit. By contrast, an *Owner* must write back the data block as well as relinquish ownership by a *DOxM* command.

Upon receiving messages, the cache controller responds as follows. For each *Inv* request, the cache invalidates its local copy and acknowledges by an *IAck*; additionally, an *Owner* must also write back the block of data to the main memory by a *DOxM* command. For each *UpdM* request received, the *Owner* responds by a *DxM* command and retains the block in *Shared* state. When ownership is acquired, the cache in the *WHP* state resumes the pending write and becomes the new *Owner*. When the data copy arrives for the cache in *RMP* (*WMP*), the cache obtains a *Shared* copy (becomes a new *Owner*). Finally, a *Reject* notice forces the cache controller to re-send the previous request.

3.2.2 Channel and Memory Processes

The channels for sending and receiving messages merely behave as communication mediums between caches and the main memory, propagating messages between them. Several auxiliary functions such as *InsertSCH*, and *GetSCH* should be clear and do not need further explanation.

The memory process is more complex. The directory entry for a data block is made of a presence bit vector (*presencebit*) and of a dirty bit (*dirtybit*). The directory entry can be in one of four states: *free, XData, XOwn* and *XOwnC*. A *free* directory indicates that there is no request in progress. If not free, the directory entry is locked because a request is pending for the block; locking of the entry is needed to maintain a critical section on each memory block. States *XData* and *XOwn* record a pending request for a *Shared* copy and for ownership transfer (*ReqO*) respectively. (The *XOwnC* state is similar to *XOwn* except that a data copy is sent along with the ownership right.) The identifier of the requesting cache is kept in variable *reqc*.

The detailed descriptions of memory behavior is given in Figure 2. What is important is that there is no explicit acknowledgment by caches to release a locked entry in this design [7, 22]. Also, when there is no cached *Owner* copy, the main memory responds immediately to requests for *Shared* copies by providing its own data without locking the directory entry [1, 6, 23]. This aggressive design that allows *multiple reader*s and an *exclusive writer* is important in reality for performance reasons, although it certainly exposes the protocol to the potential danger of race among messages. Ideally, handshake messages between caches and memory should be reduced to the extent that they guarantee a correct design, that is, the system should run at full speed and resolve race conditions when they occur, at the expense of more complex protocol controls. To design such complex protocols, verification techniques at different level of the protocol design are needed.

4 .0 Correctness Issues of the Protocol

4.1 Data Consistency

A cache coherence protocol must enforce consistency between multiple data copies. In systems where writes are not atomic, data consistency is typically enforced by allowing one and only one update in progress at any time. 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 [19].

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.*

4.2 Unspecified Message Reception

Unspecified message reception is caused by incomplete specification of the protocol. This type of flaw is very likely to happen at the early design stages because of unforeseen interleaving of memory events. Therefore, it often favors verification methodologies that help synthesize and verify the protocol incrementally. This also explains why a logic proof is not suitable: A proof of an incorrect protocol is nothing more an existence evidence than a constructive expostulation. On the other hand, a state machine model is able to show the path leading to the erroneous state. 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 a state receives messages that are not specified in the protocol specification.

4.3 Deadlock and Livelock

The detection of deadlocks and livelocks are the most challenging errors to detect. Within the framework of reachability analysis, deadlock means that the protocol enters a state without possible exits; for example, some processor is permanently blocked from accessing a given memory block. A livelock is a situation where processes interacting in the protocol could theoretically make progress but, because of a fortuitous timing of events they circle around a loop of states without making progress. In directory schemes, the livelock problem is likely to happen because the directory entry may be locked and requests must be rejected at times. A careless design may result in the directory locked permanently, and consequently no access to the given memory block can proceed.

5 .0 Symbolic State Model

The state expansion algorithm in this paper is an extension of our earlier work[19]. For the paper to be self-contained, we repeat the needed concepts here, together with appropriate extensions.

5.1 Protocol Model

Given the architecture and protocol model of Figures 1 and 2, we now formally define the constituent finite-state machines. Following the conventions in [12, 24], message transmission is represented by a negative sign, and message reception by a plus sign.

Definition 2. (Receiving Channel) *The receiving channel machine recording received messages in transit to the cache has structure* $\mathcal{RChM}=(Q_r, \Sigma_r, Xm\text{-}, \delta 1_r, Xc+, \delta 2_r)$, *where*

> Q_r: *state symbols,*
> Σ_r: *set of memory-to-cache commands(Table 1),*
> $\delta 1_r$: $Xm\text{-} \times Q_r \rightarrow Q_r$, $Xm\text{-} \in \Sigma_r$, *(recording commands sent by memory),*
> $\delta 2_r$: $Q_r \times \Sigma_r \rightarrow Q_r \times Xc+$, $Xc+ \in \Sigma_r$, *(signals to cache controller),*

Xm- and Xc+ are the messages issued by the memory and consumed by the cache respectively.

Definition 3. (Sending Channel) *The sending channel recording the messages issued by the cache and in transit to memory has structure* $\mathcal{SChM}=(Q_s, \Sigma_s, Xc\text{-}, \delta 1_s, Xm+, \delta 2_s)$, *where*

> Q_s: *state symbols,*
> Σ_s: *set of cache-to-memory commands (Table 1),*
> $\delta 1_s$: $Xc\text{-} \times Q_s \rightarrow Q_s$, $Xc\text{-} \in \Sigma_s$,*(recording messages sent by cache),*
> $\delta 2_s$: $Q_s \times \Sigma_s \rightarrow Q_s \times Xm+$, $Xm+ \in \Sigma_s$, *(signaling memory controller),*

Xc- and Xm+ represent the messages issued by the cache and consumed by the memory respectively.

At each state expansion step, a receiving (sending) channel may record the command sent by the memory (cache), or may propagate a command to the corresponding cache (the memory).

Definition 4. (Cache Machine) *The state machine characterizing the cache behavior has structure* $\mathcal{CM}=(Q_c, \Sigma_r, \Sigma_s, Xc+, \delta 1_c, Xc\text{-}, \delta 2_c)$, *where*

> Q_c: *cache state symbols,*
> Σ_r, Σ_s: *commands as defined in definitions 2 and 3,*
> $\delta 1_c$: $Xc+ \times Q_c \rightarrow Q_c \times (\varnothing \cup Xc\text{-})$, $Xc+ \in \Sigma_r, Xc\text{-} \in \Sigma_s$,
> $\delta 2_c$: $Q_c \times \Sigma_s \rightarrow Q_c \times Xc\text{-}$, $Xc\text{-} \in \Sigma_s$,

Xc+ and Xc- are the messages consumed and produced by the cache respectively.

The behavior of the cache controller is given in definition 4. Upon receiving a message, a cache controller may or may not respond by generating outgoing messages according to $\delta 1_c$. Also, the cache may issue requests, for instance, for a shared copy, which is described by $\delta 2_c$. Finally, we have the definitions for the main memory and for the global states as follows.

Definition 5. **(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 2 and 3,*

δ_{mr}: $\quad Xm+ \times Q_m \rightarrow Q_m \times (\varnothing \cup Xm-)\,,\quad Xm+ \in \Sigma_s, Xm- \in \Sigma_r \quad ,$

Xm+ and Xm- are the caches-to-memory and memory-to-caches commands respectively.

Definition 6. **(Base Machine)** *The base machine is a composition of the cache machine and its two corresponding channel machines, that is,* $\mathcal{BM}_i=(\mathcal{CM}_i \# \mathcal{RChM}_i \# \mathcal{SChM}_i)$*.*

Definition 7. **(Protocol Machine)** *The protocol machine is defined as a composition of all base machines and of the memory machine, that is,* $\mathcal{PM}=(\mathcal{BM}_1 \# \mathcal{BM}_2 \# \,........\, \# \mathcal{BM}_n \# \mathcal{MM})$ *for a system with n caches.*

The memory controller consumes the messages from the cache and responds according to the given block state and the message type. A pictorial illustration of the base machine and the protocol machine is given in Figure 1. Finally, the state of the protocol machine is also referred to *global state* in this paper.

5.2 Equivalence Relations and State Pruning

Since all base machines are characterized by the same finite state process, we can map global states to more abstract states, regardless of the number of base machines in a particular state by the following set of repetition operators. (For a more detailed justification see [19].)[2]

Definition 8. **(Repetition Operator)**

1. *The* **Singleton** *(q^1) states that there is one and only one* $\mathcal{BM}$ *in state* $q \in Q_{\mathcal{BM}}$[3]*. This operator can be omitted.*
2. *The* **Positive** *or* **Plus-operator** *(q^+) indicates that at least one* $\mathcal{BM}$ *is in state* $q \in Q_{\mathcal{BM}}$*.*
3. *The* **Star-operator** *(q^*) extends the plus-operator by including the case of null instance.* q^* *means that none or some* $\mathcal{BM}$*s are in state* $q \in Q_{\mathcal{BM}}$*.*

2. NOTE TO THE REVIEWER (will be removed from manuscript): Technical report [19] can be accessed through anonymous ftp from "usc.edu", directory "pub/CENG", file "CENG-92-20.ps.Z".

3. $Q_{\mathcal{BM}}$ and $Q_{\mathcal{PM}}$ denote the set of state symbols for the base machine and the protocol machine, respectively.

In a *composite* system state, base machines in the same state are grouped into a state *class* and specified by one of the above repetition operators as follows.

Definition 9. (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* $(q_1^{r_1}, q_2^{r_2}, \ldots, q_n^{r_n}, q_{\mathcal{MM}})$, *where* $n = |Q_{\mathcal{BM}}|$ *is all the possible state that a base machine can stay,* $q_i \in Q_{\mathcal{BM}}$, $r_i \in [0, 1, +, *]$[4] *and* $q_{\mathcal{MM}} \in Q_m$ *is the memory machine state.*

Repetition operators can be ordered by the possible states they specify. The resulting order is $1 < + < *$, and we also write $q^1 < q^+ < q^*$ where $q \in Q$. The null instance can be ordered with respect to $*$, i.e., $0 < *$. We say that q^{r_1} is *weaker* than q^{r_2} if $r_1 < r_2$, where $q \in Q$ and $r_1, r_2 \in [1, +, *]$.

Definition 10. (Containment) *We say that composite state* S_2 *contains composite state* S_1, *or* $S_1 \subseteq S_2$, *if*

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

where r1, r2 are repetition operators.

An interesting consequence of containment is that if $S_1 \subseteq S_2$ and if S_1 is not permissible then S_2 is also not permissible. S_1 could thus be discarded during the verification process provided we keep S_2. We will show that the expansion process based on expansion rules in Section 5.3 is a *monotonic* operator on the set of composite states S, that is, if $S_1 \subseteq S_2$, then $\tau(S_1) \subseteq \tau(S_2)$, or $\tau(S_1) \subseteq S_2$ where the operator τ is a memory event. As the expansion process progresses, new composite states are created. A new state is discarded if it is contained in a visited state. Similarly all visited states contained in a new state are discarded. At the end of the expansion process all visited states are *essential* states.

Definition 11. (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 simply decomposed into several families of states (which may be overlapping) represented by essential composite states.

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

5.3 Rules and Algorithm for the Expansion Process

The set of applicable operations to composite state in the state generation process is defined as follows, where '/' signifies "or" selection.

1. **Aggregation:** $(q^0, q^r) \equiv q^r$, $(q^*, q^*) \equiv q^*$, $(q, q^{1/+/*}) \equiv q^+$, and $(q^+, q^{1/+/*}) \equiv q^+$, 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 $r \in [1, +, *]$ and τ is an observed transition. For instance, the memory directory sends an invalidation signal to each of caches with a valid copy.
3. **One-step Transition:** $q_1 \rightarrow^t q_2$ and $q_1^{+/*} \rightarrow^t (q_2, q_3^*)$, 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.
4. **N-steps Transitions**: This rule specifies the repetitive application of the same transition N times, where N is an arbitrary positive integer.

   $(Q, q_1^+) \rightarrow^t (Q, q_2^1, q_1^*) \rightarrow^t (Q, q_2^2, q_1^*) \rightarrow^t ... \rightarrow^t (Q, q_2^+, q_1^*)$. It states that the same transition t can be applied infinitely many times as long as there are base machines in state q_1 and $q_1 \rightarrow^t q_2$. 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).
5. **Progress Transitions:** Two additional dual rules with similar interpretations as N-steps transitions are required for the progress of the expansion process.

   (a). $(Q, q_1^*) \rightarrow^t ... \rightarrow^t (\overline{Q}, q_2^*, q_1^0)$, if $t \in \{Inv, IAck\}$ and $q_2^{+/*} \in Q$ before any fire of t starts and

   (b). $(Q, q_1^*) \rightarrow^t ... \rightarrow^t (Q, q_2^*, q_1^*)$, otherwise.

A good way to explain the dual progress rules is by examples. Consider the trivial case in which a pending write can be resumed only after all other cached copies have been successfully invalidated. In terms of the expansion process, this means that the number of base machines with a valid data copy must eventually reach zero so that the expansion process can make progress. Therefore, we have rule (a). Rule (b) is introduced to guarantee that the expansion process retains the monotonicity property. We also notice that the final step of rule (a) *may* result in changes ($\overline{Q}$) for base machines in Q. This covers, for example, the case where the memory has received all required *IAck* messages and responds to the cache in the write pending state.

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. First, we prove, as promised, the monotonicity of the expansion process.

Lemma 1 *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

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

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

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. There are three cases.

(a). $t \in \Sigma_r$ indicates that the cache receives a memory-to-cache command. This case is trivial because we know $q_j \leq q_j^{1/+/*}$, provided $q_i \rightarrow^t q_j$.

(b). $t \in \Sigma_s$ is a request by the cache. As in case (a), $q_j \leq q_j^{1/+/*}$ after the transition, provided $q_i \rightarrow^t q_j$.

(c). $t \in \Sigma_s$ and the expansion step is taken by the memory that responds to a cache-to-memory command. The same arguments can be made as above except that a little care is needed for the case that t is an invalidation acknowledgment because, if machines in class q_i are the last ones with valid copies to prevent the progress of a pending write, the write will complete after the receptions of *IAck* from machines in q_i. In such a case, we have the following transition, by applying t to S_1:

$$S_1=(Q_1, q_i, q_{\mathcal{MM}1}) \rightarrow^t \overline{S}_1=(Q_1', q_j, q_{\mathcal{MM}1}')$$

and, the following transitions by applying t to S_2:

(1). $S_2=(Q_2, q_i, q_{\mathcal{MM}2}) \rightarrow^t \overline{S}_2=(Q_2', q_j, q_{\mathcal{MM}2}')$

(2). $S_2=(Q_2, q_i^+, q_{\mathcal{MM}2}) \rightarrow^t (Q_2, q_i^*, q_j^+, q_{\mathcal{MM}2}) \rightarrow^t \overline{S}_2=(Q_2', q_i^0, q_j^+, q_{\mathcal{MM}2}')$

(3). $S_2=(Q_2, q_i^*, q_{\mathcal{MM}2}) \rightarrow^t (Q_2, q_i^*, q_j^*, q_{\mathcal{MM}2}) \rightarrow^t \overline{S}_2=(Q_2', q_i^0, q_j^*, q_{\mathcal{MM}2}')$

where Q_1, Q_2 denotes all other irrelevant base machines in the tuples, and $Q_1 \leq Q_2$ by the lemma statement. Clearly, (2) and (3) are the results of applying the progress rules and it may need two expansion steps to end up in a state $\overline{S}_2$ containing $\overline{S}_1$. Also, needless to say, $q_{\mathcal{MM}1}'$ must be equal to $q_{\mathcal{MM}2}'$ such that the memory directory is free. Finally, $Q_1' \leq Q_2'$. From (a), (b), and (c), we can thus conclude that $\overline{S}_1 \subseteq \overline{S}_2$.

Lemma 2 *The claims $\overline{S}_1 \subseteq \overline{S}_2$ or $\overline{S}_1 \subseteq S_2$ hold if $S_1 \subseteq S_2$, that is, $r_j \leq \bar{r}_j$ for all j and $q_{\mathcal{MM}1}=q_{\mathcal{MM}2}$.*

Proof: The result extends the conclusion of lemma 1 and the proof is similar. Let's consider that the transition t is applied to base machines in state class q_i. Only the special cases $S_1=(Q, q_i^+, q_{\mathcal{MM}1})$ and $S_2=(Q, q_i^*, q_{\mathcal{MM}2})$ dealing with the progress rules need to be justified. The result of applying the transition t such that $q_i \rightarrow^t q_j$ is

(1). $S_1=(Q, q_i^+, q_{\mathcal{MM}1}) \rightarrow^t \overline{S}_1=(Q, q_i^*, q_j^+, q_{\mathcal{MM}1})$, and

(2). $S_2=(Q, q_i^*, q_{\mathcal{MM}2}) \rightarrow^t \overline{S}_2=(Q, q_i^*, q_j^*, q_{\mathcal{MM}2})$, or

(3). $S_2=(Q, q_i^*, q_{\mathcal{MM}2}) \rightarrow^t \overline{S}_2=(Q', q_i^0, q_j^*, q_{\mathcal{MM}2}')$, by the progress rule (a).

By (1) and (2), it is clear that $\overline{S}_1 \subseteq \overline{S}_2$. By (1) and (3), we know that $\overline{S}_1 \subseteq S_2$ because the antecedent for applying the progress rule (a) is that $q_j^{+/*}$ must exist in Q already, and therefore, $\overline{S}_1 \subseteq S_2$. In either case, the successive states obtained by expanding S_1 will be contained by states obtained by expanding S_2.

By a recursive induction from lemma 2, we can show that the expansion process based on the symbolic structure is *monotonous*.

Corollary 1 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 $\overline{S}_1 \subseteq \overline{S}_2$.

The preceding results suggest a very efficient expansion process to obtain essential states as shown in Figure 3. Two lists keep track of non-expanded and visited states. At each step, a new state is produced by expanding the current state, and then a pruning process justified by the monotonicity property removes contained states. The final output reported in list H is the set of essential states. All possible states are included in the reported essential states [19].

```
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 $\in$ A
       for all applicable operations $\tau$ on v
           A $\rightarrow^{\tau}$ A'.
           for any state P $\in$ W and Q $\in$ H
               if (A' $\subseteq_{\mathcal{F}}$ P or A' $\subseteq_{\mathcal{F}}$ Q or A' $\subseteq_{\mathcal{F}}$ A)
                  then discard A'.
               else begin
                       remove P from W if P $\subseteq_{\mathcal{F}}$ A'.
                       remove Q from H if  Q $\subseteq_{\mathcal{F}}$ A'.
                       add A' to W.
                       if (A $\subseteq_{\mathcal{F}}$ A') then discard A  and terminate
                         all FOR loops starting a new run.
                    end
           end
       end
   end
   insert A to H if A is fully expanded and is not contained.
end.
```

FIGURE 3. Algorithm for Generating Essential States.

6 .0 Some Errors Found During Verification

In this section, we describe some representative errors found in the verification process of the protocol of Figure 2. For the sake of simplifying the description of these errors, let's consider a system with three processors, P_1, P_2 and P_3. A illustrative sequence of expansion steps showing possible flaws is listed in the Appendix.

6.1 Data Inconsistency

The algorithm given in Figure 2 will end up with data inconsistency conditions in the following case. Initially, P_1 has a clean shared copy, and the following steps are then taken.

1. P_1 evicts its copy and issues a *Repl* to the memory.
2. At some later time, P_1 has a read miss to the replaced block and issues a *ReqSC* to the memory.
3. Both commands *Repl* and *ReqSC* are in transit. The memory receives the *ReqSC* request first and responds by supplying its data copy.
4. Next, the memory receives the *Repl* and reset the corresponding presence bit for P_1.
5. P_1 receives the data copy in *Shared* state; however, its presence bit was reset in the memory directory. As a result, when P_2 modifies the block, P_1's copy is not invalidated and an inconsistency can be detected in P_1's cache.

This bug and many other similar bugs found are caused by the simple design that the memory does not do any check on the presence bits when the *ReqSC* request is received as shown in line 58 of Figure 2. As a matter of fact, many other flaws of the same kind prove that the state of the presence bit defines a very strong *antecedent* condition for a cache request being legal. For example, when the memory receives a *ReqO* request, the presence bit for the requesting cache must be set. This strong connection between state of the presence bit and the cache request should be an essential support for a correct design, but it does not suffice.

6.2 Unspecified Reception

In the cache algorithm's simplest form, many unspecified message reception flaws were found as in Table 2.

TABLE 2. Unexpected Receptions.

cmd\state	I	RMP	WMP	WHP
Inv	X	X	X	X
UpdM	X	X	X	

Rather than list the sequences leading to them all, a typical example to illustrate this problem is shown below. Initially, P_1, P_2 and P_3 have a shared copy.

1. P_1 evicts its copy, turns its state to *Invalid*, and a *Repl* to the memory is sent.
2. P_3 attempts to modify its copy and a *ReqO* is sent after its local state has changed to *WHP*.
3. When the memory receives the *ReqO* from P_3, invalidations are sent to P_1 and P_2.

4. P_1, in the *Invalid* state, thus receives an invalidation request, which is an unspecified transition in the algorithm of Figure 2.

An intuitive way to resolve this bug is to have P_1 ignore the invalidation request because P_1 can expect that the *Repl* command sent before will serve the same purpose. Unfortunately, this simple resolution may lead the system to a livelock condition as shown next.

6.3 Livelock

Following the example of the previous section, assume that a *Repl* command has been issued by P_1 and is in transit, an invalidation request is heading to P_2, and the directory entry is locked by the request from P_3.

Suppose P_2 receives the invalidation and acknowledges the memory first, the *Repl* from P_1 then arrives some time later. At this stage, no cached copies (except P_3) exist, but the system is livelocked. This bug is caused by the fact that when the memory receives a replacement request, it only resets the corresponding presence bit as shown in line 91 of Figure 2. Therefore, if the last cached copy is evicted by a replacement, the system cannot make progress even though all cached copies are cleaned.

6.4 Difficulties in Obtaining a Correct Design

The major difficulty of making a correct protocol design is the fact that there are many unforeseen interleaving of memory events. Races among messages make the discovery of random protocol flaws possible. Sometimes, the resolution of one flaw may produce other problems as we show below.

Initially, both processors P_1 and P_2 have a shared copy and request ownership (*ReqO*) independently. Suppose that the memory receives the request from P_2 first, sends an invalidation to P_1, and locks the directory entry. Next, processor P_1 in *WHP* state receives the invalidation unexpectedly. If P_1 ignores the invalidation request, the directory entry will be locked by P_2 permanently. On the other hand, if P_1 responds by an *IAck*, a race between the *IAck* and the previously sent *ReqO* begins. If the *IAck* message arrives at the memory first, the memory unlocks the directory entry and resumes the request from P_2. When the *ReqO* request from P_1 arrives, the corresponding presence bit for P_1 has been reset and the *ReqO* command is not appropriate any more. This bug can be resolved, as commented before, by using the presence bit state as an antecedent condition, so that the memory rejects the request and force P_1 to make a new request. However, this does not suffice to a correct design.

Next, let's consider an initial configuration such that P_1 is an owner and P_2 issues a *ReqOC* because of a write miss. When the memory receives the request from P_2, an invalidation is sent to P_1. Before the invalidation arrives at P_1, which evicts the target block and a *DOxM* message is sent to the memory. Henceforth, a problem of *ambiguity* occurs: the *DOxM* command sent to the memory can be a result of owner replacement, or a result of response to an invalidation request. If the memory treats both cases as the same, the invalidation previously sent to P_1 can cause further problems. Although it is always possible to introduce more message types to avoid such ambiguities, the protocol structure will get more and more complex and will increase implementation costs.

7 .0 Conclusion and Prospects

In this paper, we have investigated the application of a new verification technique to a directory-based cache coherence protocol at a low implementation level. The verification technique is very promising. All listed errors are found in a few tens or houndreds state visits, in a few seconds of computation time. What is important is that the verification procedure allows us to fix protocol flaws and re-run the procedure quickly. It results in a fast design process.

The protocol structure is basically simple; however, it does have overwhelming complexities when the implementation details are taken into account. After we constructed a formal model for the cache algorithm, several unexpected flaws were found, most of them caused by unforeseen interleavings of memory events. Our results suggest that there should exist a strong antecedent and consequent relation between the presence bits at the memory directory and the cache requests. Also, we believe that the protocol should be designed with a minimum number of messages exchanged in order to have good performance. Mechanisms that synchronize the cache state and the memory state explicitly are only required when the protocol control is under the threat of entering an erroneous state.

Although there is no further concrete evidence guiding the protocol designers to design a error-free protocol, it is vital to understand the possible sources of protocol flaws. We are planning to further apply our methodology to the protocol of Figure 2 to provide guidelines for designing correct and efficient directory-based protocols in the future.

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. 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.

[3] 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.

[4] 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.

[5] 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).*

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

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

[8] 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.

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

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

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Pong, F. and Dubois, M., " The Verification of Cache Coherence Protocols ", to appear, *5th Annual Symp. on Parallel Algorithm and Architecture.* Also, Technical Report CENG-92-20, University of Southern California, 1992.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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 An illustrative sequence of state expansion steps.

In this appendix, we show a path taken by the state expansion process and comment on possibly observed design flaws in the raw design. We use the notation ${}^{p}_{R}C^{r}_{S}$ to represent the state of base machine including cache state C, states of receiving channel R and sending channel S. Superscript p denotes the status of corresponding presence bit at main memory and superscript r is the repetition operator. Then, a global state groups base machines and the memory state in a tuple as in definition 9. Finally, the owner cache (from the perspective of main memory) is **bold-faced**; and the cache with granted request in progress is ***italicized***.

S1: (Initial state) $({}^{0}_{\phi}\mathrm{I}^{+}_{\phi}$, free) -> ${}^{0}_{\phi}\mathrm{I}^{+}_{\phi}$ write miss ->

S2: $({}^{0}_{\phi}\mathrm{I}^{*}_{\phi}$, ${}^{0}_{\phi}\mathrm{WMP}^{+}_{\mathrm{ReqOC}}$, free) -> ${}^{0}_{\phi}\mathrm{I}^{*}_{\phi}$ read miss ->

S3: $({}^{0}_{\phi}\mathrm{I}^{*}_{\phi}$, ${}^{0}_{\phi}\mathrm{RMP}^{*}_{\mathrm{ReqSC}}$, ${}^{0}_{\phi}\mathrm{WMP}^{+}_{\mathrm{ReqOC}}$, free) -> memory receives ReqOC from ${}^{0}_{\phi}\mathrm{WMP}^{+}_{\mathrm{ReqOC}}$ ->

S4: $({}^{0}_{\phi}\mathrm{I}^{*}_{\phi}$, ${}^{0}_{\phi}\mathrm{RMP}^{*}_{\mathrm{ReqSC}}$, ${}^{1}_{\mathbf{Data}}\mathbf{WMP}_{\phi}$, ${}^{0}_{\phi}\mathrm{WMP}^{*}_{\mathrm{ReqOC}}$, free) -> memory receives ReqSC from ${}^{0}_{\phi}\mathrm{RMP}^{*}_{\mathrm{ReqSC}}$ ->

S5: $({}^{0}_{\phi}\mathrm{I}^{*}_{\phi}$, ${}^{0}_{\phi}\mathrm{RMP}^{*}_{\mathrm{ReqSC}}$, ${}^{\mathbf{0}}_{\phi}\boldsymbol{RMP}_{\phi}$, ${}^{1}_{\mathbf{Data,\,UpdM}}\mathbf{WMP}_{\phi}$, ${}^{0}_{\phi}\mathrm{WMP}^{*}_{\mathrm{ReqOC}}$, XData) -> memory receives ReqOC from ${}^{0}_{\phi}\mathrm{WMP}^{*}_{\mathrm{ReqOC}}$ ->

Note that the (presumed owner) cache in WMP state can receive UpdM request unexpectedly.

S6: $({}^{0}_{\phi}\mathrm{I}^{*}_{\phi}$, ${}^{0}_{\phi}\mathrm{RMP}^{*}_{\mathrm{ReqSC}}$, ${}^{\mathbf{0}}_{\phi}\boldsymbol{RMP}_{\phi}$, ${}^{1}_{\mathbf{Data,\,UpdM}}\mathbf{WMP}_{\phi}$, ${}^{0}_{\phi}\mathrm{WMP}^{*}_{\mathrm{ReqOC}}$, ${}^{0}_{\mathrm{Reject}}\mathrm{WMP}^{*}_{\phi}$, XData) -> Memory receives ReqSC from ${}^{0}_{\phi}\mathrm{RMP}^{*}_{\mathrm{ReqSC}}$ ->

S7: $({}^{0}_{\phi}\mathrm{I}^{*}_{\phi}$, ${}^{0}_{\phi}\mathrm{RMP}^{*}_{\mathrm{ReqSC}}$, ${}^{0}_{\mathrm{Reject}}\mathrm{RMP}^{*}_{\phi}$, ${}^{\mathbf{0}}_{\phi}\boldsymbol{RMP}_{\phi}$, ${}^{1}_{\mathbf{Data,\,UpdM}}\mathbf{WMP}_{\phi}$, ${}^{0}_{\phi}\mathrm{WMP}^{*}_{\mathrm{ReqOC}}$, ${}^{0}_{\mathrm{Reject}}\mathrm{WMP}^{*}_{\phi}$, XData) -> presumed owner in **WMP** state receives Data ->

S8: $({}^{0}_{\phi}I^{*}_{\phi}, {}^{0}_{\phi}RMP^{*}_{ReqSC}, {}^{0}_{Reject}RMP^{*}_{\phi}, {}^{0}_{\phi}\mathit{RMP}_{\phi}, {}^{1}_{UpdM}O_{\phi}, {}^{0}_{\phi}WMP^{*}_{ReqOC}, {}^{0}_{Reject}WMP^{*}_{\phi}$, XData) -> owner cache performs replacement ->

S9: $({}^{0}_{\phi}I^{*}_{\phi}, {}^{0}_{\phi}RMP^{*}_{ReqSC}, {}^{0}_{Reject}RMP^{*}_{\phi}, {}^{0}_{\phi}\mathit{RMP}_{\phi}, {}^{1}_{UpdM}\mathbf{I}_{DOxM}, {}^{0}_{\phi}WMP^{*}_{ReqOC}, {}^{0}_{Reject}WMP^{*}_{\phi}$, XData) -> memory receive DOxM from ${}^{1}_{UpdM}\mathbf{I}_{DOxM}$->

The replacement by owner in S8 may result in the case that cache in Invalid state (${}^{1}_{UpdM}\mathbf{I}_{DOxM}$) in S9 receives UpdM command unexpectedly.

S10: $({}^{0}_{\phi}I^{*}_{\phi}, {}^{0}_{\phi}RMP^{*}_{ReqSC}, {}^{0}_{Reject}RMP^{*}_{\phi}, {}^{1}_{Data}RMP_{\phi}, {}^{0}_{UpdM}I_{\phi}, {}^{0}_{\phi}WMP^{*}_{ReqOC}, {}^{0}_{Reject}WMP^{*}_{\phi}$, free) -> memory receives ReqOC from ${}^{0}_{\phi}WMP^{*}_{ReqOC}$ ->

S11: $({}^{0}_{\phi}I^{*}_{\phi}, {}^{0}_{\phi}RMP^{*}_{ReqSC}, {}^{0}_{Reject}RMP^{*}_{\phi}, {}^{1}_{Data, Inv}RMP_{\phi}, {}^{0}_{UpdM}I_{\phi}, {}^{0}_{\phi}WMP^{*}_{ReqOC}, {}^{0}_{Reject}WMP^{*}_{\phi}, {}^{0}_{\phi}\mathit{WMP}_{\phi}$, XOwnC) -> write miss occurs at ${}^{0}_{UpdM}I_{\phi}$ ->

S12: $({}^{0}_{\phi}I^{*}_{\phi}, {}^{0}_{\phi}RMP^{*}_{ReqSC}, {}^{0}_{Reject}RMP^{*}_{\phi}, {}^{1}_{Data, Inv}RMP_{\phi}, {}^{0}_{UpdM}WMP_{ReqOC}, {}^{0}_{\phi}WMP^{*}_{ReqOC}, {}^{0}_{Reject}WMP^{*}_{\phi}, {}^{0}_{\phi}\mathit{WMP}_{\phi}$, XOwnC) -> ${}^{1}_{Data, Inv}RMP_{\phi}$ receiving data ->

Note that cache in WMP state can receive UpdM command unexpectedly.

S13: $({}^{0}_{\phi}I^{*}_{\phi}, {}^{0}_{\phi}RMP^{*}_{ReqSC}, {}^{0}_{Reject}RMP^{*}_{\phi}, {}^{1}_{Inv}S_{\phi}, {}^{0}_{UpdM}WMP_{ReqOC}, {}^{0}_{\phi}WMP^{*}_{ReqOC}, {}^{0}_{Reject}WMP^{*}_{\phi}, {}^{0}_{\phi}\mathit{WMP}_{\phi}$, XOwnC) -> ${}^{1}_{Inv}S_{\phi}$ evicts the block ->

S14: $({}^{0}_{\phi}I^{*}_{\phi}, {}^{0}_{\phi}RMP^{*}_{ReqSC}, {}^{0}_{Reject}RMP^{*}_{\phi}, {}^{1}_{Inv}I_{Repl}, {}^{0}_{UpdM}WMP_{ReqOC}, {}^{0}_{\phi}WMP^{*}_{ReqOC}, {}^{0}_{Reject}WMP^{*}_{\phi}, {}^{0}_{\phi}\mathit{WMP}_{\phi}$, XOwnC)

As described in section 6.3, if the cache in Invalid state (${}^{1}_{Inv}I_{Repl}$) ignores the invalidation signal and expect the previously sent Repl to serve the same purpose, the system will enter a livelocked suitation according to the algorithm in Figure 2.