The Verification of Cache Coherence Protocols

Fong Pong and Michel Dubois

CENG Technical Report 92-20

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

May 1993

The Verification of Cache Coherence Protocols

Abstract

In this paper we introduce a verification technique for cache coherence protocols at the behavior level. Protocols are specified by a Finite State Machine (FSM) model. The global state space is the Cartesian product of an arbitrary number of individual cache state spaces and is symbolically expanded. A global FSM characterizing the protocol behavior is built and protocol verification becomes equivalent to finding whether or not the global FSM may enter erroneous states. State expansion only takes a few steps, contrary to current approaches. The verification procedure is applied to the verification of five existing protocols

1.0 Introduction

In a shared-memory multiprocessor, private caches are needed to reduce the effects of memory access latency and contention. Whereas private caches significantly improve system performance, they introduce the *cache coherence* problem. Multiple cached copies of the same memory word must be consistent at any time. A *cache coherence protocol* ensures that changes made to shared memory locations by any processor are visible to all other processors.

A cache coherence protocol is a set of rules coordinating communicating *entities* (usually cache and memory controllers) to enforce consistency among multiple data copies. Several protocols [1, 2, 6, 9] have been proposed, described and implemented; however, they have never been formally validated. The simplicity of these protocol, the lack of verification tools, and the complexity of current formal validation procedures may explain this state of affair. Informal techniques for protocol verification are based on time-consuming, error-prone testing procedures by engineers and require a great deal of ingenuity. As the complexity of protocols grows, it becomes extremely difficult to verify protocols by simply relying on human reasoning.

In a broad sense, the goal of validation is to verify that a protocol satisfies its specification and possesses the required invariant properties. Validation activities, including simulation studies, state reachability analysis and logical reasoning, should exist at all phases of design and implementation. Simulations are conceptually simple but suffer from *incompleteness* since a random test sequence must be run indefinitely to enter all reach-

able states. It is also very unlikely that validation procedures based on trace-driven simulations can detect most design errors. A protocol passing the test is only shown to be correct with respect to the particular simulation runs, and hence only partial correctness is demonstrated.

Early work by Baer and Girault [2] introduced a Petri Net model of cache protocols. This model comprehensively specifies the underlying hardware structure. The Petri Net model is valuable in capturing the synchronization between communicating hardware entities, and hence, it is an important methodology for mapping protocol designs to actual implementations. The construction of a Petri Net model as shown in [2] is difficult to automate and is very complex even though the protocol is simple. Moreover, the final step of verification is a lengthy, verbose analysis rather than a formal validation.

Reachability analysis is primarily based on exploring exhaustively all the possible interactions between entities interacting in the protocol. The system can be characterized by its *state*. From a given state, the exploration of all possible interactions between entities leads to a number of new states. States in which the protocol fails to preserve expected correctness properties are classified as *erroneous* states; otherwise states are *permissible*. If any erroneous state is reachable, the protocol is incorrect. The major difficulty of this technique is the *"state space explosion"* problem [5, 8]; normally the state exploration complexity blows out rapidly with the growing number and complexity of entities involved in the protocol. Reachability analysis has been widely adopted for the automated verification of communication protocols because violations of correctness conditions such as unspecified reception (state receiving unspecified messages) and deadlock (state without exits) are directly tied to the structure of the reachability graph [5, 7, 12]. However, in order to validate a cache coherence protocol, the values of data copies must also be tracked during the generation of reachable states. State models are adequate for state-oriented transition aspects, but they do not capture aspects associated with data values.

In [15], Rudolf and Segal presented a proof of a snooping protocol by enumerating the various scenarios of reads and writes. Each cache is considered as a finite state automaton and a *product machine* is a collection of n finite state automata. Nanda and Bhuyan [16] presented a similar approach which based on the composition of communicating finite state machines and on state enumeration. In an enumeration approach a large number of redundant states are visited and expanded during the state expansion procedure [16]. Enumerating states for complex protocols faces the state space explosion problem.

Another technique for validating protocols relies on logical proofs [5, 11, 13]. This approach can validate a full range of properties. Ideally, any property which can be formulated in logic can be verified, but proof and formulation of assertions reflecting the desired correctness properties are often error-prone and need considerable ingenuity. In some studies requiring great efforts, correctness conditions are still incorrectly and/or incompletely formulated. More importantly, this approach cannot deal with state-oriented transitions.

An approach combining the advantages of reachability analysis and of logical proof has recently been applied to the verification of communication protocols [3, 4, 5, 11, 14]. Reachability analysis based on state models with augmented variables and processing routines is used to expand major states, while logic properties are formulated and proved over associated context variables. This intermediate approach is suited to the verification of cache coherence protocols: on one hand the coherence activities are mainly reflected by the state changes of caches, which suggests the reachability analysis; on the other hand, the modeling of data aspects can be dealt with by augmenting the state description with context variables.

Recent work has focused on the state space explosion problem. Instead of enumerating the state space, a *symbolic* expansion of the state space is advocated in [10]. A complex system often exhibits a great deal of *regularity.* Ideally, a structure which exploits this regularity is able to compress the representation of a set of states by extracting their common features. For example, the regular expression $\{A,B,C\}^*\{A\}$ represents the set of all strings containing an arbitrary number of replications of $\{A,B,C\}$ and ending with A. This symbolic form provides a concise representation for the enumeration of an infinite set. An *n-ary* relation $\mathcal{R}$ with respect to sets $A_1, A_2, \ldots, A_n$ is any subset of $A_1 \times A_2 \times \ldots \times A_n$. We can characterize a relation $\mathcal{R}$ by a boolean function $f: A_1 \times A_2 \times \ldots \times A_n \rightarrow \{true, false\}$, such that $f(a_1, a_2, \ldots, a_n) = true$ if and only if the particular tuple $(a_1, a_2, \ldots, a_n)$ with $a_i \in A_i$ satisfies the relation $\mathcal{R}$. In this way [10], the enumeration of the state space is replaced by a search for sets of relations whose union is the global state space. Whether or not a state is a member of the set of states represented by $\mathcal{R}$ is indicated by the value of f corresponding to the state.

This paper introduces a new methodology for validating cache coherence protocols at the early design stage. Our method is based on reachability analysis. The system state is the composition of individual cache states as in [15, 16], but the system state space is symbolically expanded and represented. Abstracted from the details of their hardware

implementation, protocols are specified by finite state automata which characterize the behavior of the caches. Relying on the *symmetry* of the system, we derive equivalence relations among states; we then group all caches with equivalent states into classes. Each class is represented separately and its representation includes a set of caches with equivalent states. This state representation drastically reduces the complexity of the state expansion procedure. A global state graph which facilitates the interpretation of the protocol behavior is reported at the end of the procedure. The global state graph is useful not only to verify data consistency but also to demonstrate the similarities and disparities among protocols.

The first part of this paper provides a protocol model with its fundamental definitions. Equivalence relations leading to the state representation by classes are then introduced, followed by the symbolic construction of the global state graph. Finally, the methodology is applied to several protocols of moderate complexity described in [1].

2.0 Protocol Model

2.1 Finite State Automaton

In this paper, we employ a simple finite state machine (FSM) model to specify the protocols. The protocol model has the following assumptions:

1. Each protocol transition is *atomic*, that is, the time required for the change of states of all caches is zero.
2. Only the interactions between caches and main memory are modeled.

Representing a cache coherence protocol by an FSM model is natural from the perspective of protocol designers. In the past, the FSM models have been extensively used to describe and specify cache coherence protocols at a logical level. Without loss of generality, formal definitions of the protocol model are as follows.

Definition 1 (FSM Model) *A deterministic finite state machine modeling a cache coherence protocol has structure* $\mathcal{M}=(Q, \Sigma, \mathcal{F}, \delta)$ where

Q is a finite set of state symbols,

Σ is the set of operations causing state transitions,

$\mathcal{F}$ is a characteristic function defined for each state and, which can be null; and

δ defines the state transition functions $\mathcal{F} \times Q \times \Sigma \rightarrow Q$.

Finally, the finite state machine must be *strongly connected*, that is, starting from any given state there exist at least one path leading to all other states.

Definition 2 (Global or System State) *With respect to a particular memory location, the global or system state is defined as the composition of each individual cache state*[1].

In several protocols, the next state of a cache depends only on its current state and on the type of request issued by the local processor; for these protocols the characteristic functions is null. The characteristic function $\mathcal{F}$ is introduced to include protocols whose transitions depend on the global state, and not just on the state of the local cache. For example, in the Illinois protocol [1], a read miss in a cache may bring in the desired cache block either in state *Valid-Exclusive* or *Shared*, depending on the presence of the block in other caches. Other protocols exhibiting this property are write-broadcast protocols such as the Dragon and the Firefly. We can define a relation and its characteristic (*sharing-detection*) function over all caches' states from the perspective of cache C_i as:

$$f_i(C_1, C_2, \ldots, C_n) = \begin{cases} true & \text{if } \exists C_j \neq C_i \text{ s.t. } state(C_j) \neq invalid \\ false & \text{otherwise} \end{cases}$$

and $\mathcal{F} = (f_1, f_2, \ldots, f_n)$.

We consider that the state *invalid* includes the cases where the block is in cache and has been invalidated or where the block is not present in cache. Note that there is one sharing-detection function per cache. If there is only one copy in caches then the function returns 0 for the cache with the valid copy and 1 for all other caches. In fact, the *sharing-detection* function as introduced here provides a more formal framework to model protocols than the framework partially specified in natural language given in [1]. Although $\mathcal{F}$ may be generalized to any function of the global state even unrelated to sharing, we limit ourselves to functions used in existing protocols, that is, $\mathcal{F}$ is either null or is the *sharing-detection* function.

1. Throughout this paper, we track the state of a single block. For simplicity we use the terms *cache state* and *memory state* to signify the state of block copies in a cache or in memory.

The FSM model specifies the protocol behavior in terms of global state transitions and is only the support for the protocol verification. To some extent, each cache state carries some semantic interpretation. For example, in the Illinois protocol, a cache block in the *Dirty* state means that the local copy has been modified and that main memory has an obsolete data copy, whereas a cache block in the *Shared* state indicates that the local copy is potentially shared with other caches and that all cached copies must be identical with the main memory copy. As a result, if different caches are in states *Dirty* and *Shared* in the same global state then a contradiction in the interpretation of cache states occurs. Similarly several caches in the *Dirty* state signal another contradiction. This suggests a primary verification procedure consisting of searching all reachable global states and proving that all reached states are *permissible* in the sense that individual cache states are compatible [16]. The problem of searching the global state space is therefore converted into the problem of finding an efficient model for the global FSM.

2.2 Model for Data Consistency

A cache coherence protocol must support correct execution of a program on a multiprocessor system. In general, there are two distinct requirements:

- The ordering of accesses must conform to a well defined consistency model and the parallel code must be written correctly for this model.
- The memory system must be *generally coherent*, i.e. if we assume that all accesses are atomic, the cache protocol must always return the latest value on each load.

We formulate this latter condition within the framework of the reachability expansion as follows.

Definition 3 (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.

Global states which violate this condition will be referred to as *erroneous* states. If any erroneous state is reachable, the protocol is incorrect.

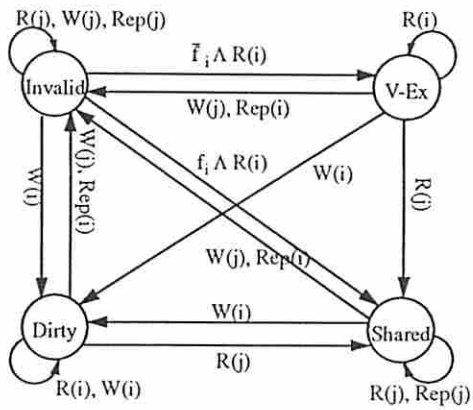


FIGURE 1. The Illinois Transition Diagram from Perspective of Cache C_i.

2.3 The Illinois Protocol

Figure 1 shows the state transition diagram of the Illinois protocol. We will use this protocol as a running example throughout the paper. The Illinois protocol distinguishes private and non-actively shared blocks from actively shared blocks such that invalidations for write hits on private and non-actively shared blocks can be avoided. On a read miss a block is loaded in states *Valid-Exclusive* or *Shared* depending on the value of the *sharing-detection* function *f*. There are four states for cached blocks: *Invalid*, *Valid-Exclusive* (not modified; only copy in caches), *Shared* (not modified, possibly copies in other caches) and *Dirty* (modified; only copy in caches). We can summarize the Illinois protocol as the following:

- State Symbols Q= {*Invalid*, *Valid-Exclusive*, *Shared*, *Dirty*}.
- Operations $\Sigma = \{R, W, Rep\}$, which stand for *read*, *write* and *replacement*.
- Cache algorithm from the perspective of cache C_i:
  1. **Read Hit**. No coherence actions need to be taken.
  2. **Read Miss**. If cache C_j has a dirty copy, C_j will supply the missing block and update the main memory at the same time; both C_i and C_j change their local state to *Shared*. Otherwise, C_i gets the missed block from other caches if any copies are cached and all caches with a copy of the block set their local state to *Shared*. If there is no cached copy, C_i gets the block from main memory and receives an exclusive copy (state *Valid-Exclusive*).

3. **Write Hit.** If the block is *Dirty*, no action is taken. If the write is *Valid-Exclusive*, its state changes to *Dirty*. Otherwise, all other caches with a copy must invalidate their local copies; and C_i receives a copy in state *Dirty*.

4. **Write Miss.** Like a read miss except that all other caches with a copy invalidate it and the block is loaded in the *Dirty* state.

5. **Replacement**. Only blocks in state *Dirty* need to be written back to main memory.

2.4 Augmenting the Global State with Context Variables

The definition of global state is primarily based on the cache block state. However to verify the protocol we need to augment it with attributes characterizing the value of data.

Definition 4 (Augmented Global State) *In a system with n caches, the global state of a block* $G=(q_1, q_2,..., q_n)$, $q_i \in Q$ *is augmented by* $M=(m_1, m_2,..., m_n)$, *where* m_i *denotes a set of context variables which characterize the block from the point of view of cache i.*

The global state is augmented in order to strengthen the FSM model by the introduction of auxiliary variables M. By a state transition $(G,M) \rightarrow^{\tau} (G',M')$, we actually mean that $(G \rightarrow^{\tau} G')$ and the discrepancy between M and M' reflects the environmental changes, and, in particular, the data aspects such as data transfers and values. Thus, the reachability graph is constructed over all reached global states G and associated Ms are considered attributes of these states.

The augmented context variables in definition 4 allows the caches to customize their views of the system in a flexible way. Caches may have different views of their local copies and of the memory copy because of the buffering of stores. For protocols in this paper, all caches will have the same view of the memory block because the effects of a write operation is instantaneous in these protocols (we have assumed atomic accesses throughout this paper). Therefore, a single global context variable can be used to represent the state of the memory copy. The more general notation has nonetheless been introduced here so that it will applicable to more complex protocols later on.

To formally model the Illinois protocol is relatively easy because the state of all the words in the block are the same as the state of the block containing it. Therefore, we can work on the block state or word state equivalently. The formal model associates each cache C_i with an auxiliary variable $cdata_i$ and the memory with auxiliary variable *mdata*

to keep track of consistency between memory and cached copies. Selecting a proper abstraction level, *cdata* takes values from domain *{nodata, fresh, obsolete}* and *mdata* from domain *{fresh, obsolete}*. Initially, all caches are in the *Invalid* state without data copies ($cdata_i$=*nodata*, for all *i*) and memory has the fresh copy noted as *mdata=fresh*. The value assignments to these variables during the state expansion conform to the protocol algorithm. Data inconsistency occurs when a processor can access its local copy indicated as an *obsolete* copy. With this notation, the data transfer aspects of the Illinois cache protocol from the perspective of C_i can be specified as follows.

1. **Read Miss.**

   if (there exist C_j in *Dirty* state)
   ($mdata = cdata_j$) /* update memory */
   ($cdata_i = mdata$) /* get data from memory */
   else if (there is no cached copy)
   ($cdata_i = mdata$) /* get data from memory */
   else /* arbitrarily choose C_j with a copy */
   ($cdata_i = cdata_j$)

2. **Write Hit.**

   if (C_i has a dirty copy)
   no action is taken
   else
   $\forall j\ cdata_j = nodata$ /* invalidated */
   $cdata_i = fresh$
   $mdata = obsolete$

3. **Write Miss.**

   if (there exist C_j in *Dirty* state)
   $cdata_i = cdata_j$ /* must be a fresh copy */
   else if (there exist C_j with a copy)
   $cdata_i = cdata_j$ /* must observe fresh copy */
   else /* no cached copy */
   $cdata_i = mdata$ /* must be fresh at this stage */

   $\forall j\ cdata_j = nodata$ /* invalidated */
   $cdata_i = fresh$
   $mdata = obsolete$

4. **Replacement.**

   if (C_i has a dirty copy)
   $mdata = cdata_i$
   $cdata_i = nodata$

3.0 Reachability Analysis - State Space Expansion

Since the reachability graph is constructed over all global states and value assignments to auxiliary variables M are irrelevant to state transitions, the following Sections concentrate on the transitions between global states alone.

3.1 Exhaustive Enumeration of the State Space

To completely verify the protocol at the finite automaton level, all states and transitions must be exhaustively simulated. Conventionally, an exhaustive search algorithm as shown in Figure 2 is used to explore the system state space. In this algorithm, a working list of newly produced states and an history list keeping track of visited states are maintained. At each step, all states directly reachable from the current state are generated and inserted in the working list if necessary. Redundant states are pruned by checking the new state against the visited states.

```
Algorithm: exhaustive search.

W : list of working states.
H : list of visited states.

while (W is not empty) do
begin
   get current state A from W and put A in H.
   for all A', A' is a successor state of A.
      if (A' ∉ W∪H)
         then add A' to W.
end.
```

FIGURE 2. Algorithm for exhaustive search.

Since the state space is enumerated explicitly, the number of caches must be exactly defined. As a result, the state space must be finite because the number of state symbols and cache events are also finite. Let's have a system with n caches, $|Q| = m$ state symbols and $|\Sigma| = k$ cache events. The maximum number of state in the system state space is $(m)^n$ states. However, the number of state visited in the expansion process is far more than $(m)^n$ states. For each state in the working list, we must generate all its directly reachable states although some of them may have been visited previously. Without any pruning effort, we need at least approximately $nk(m)^n$ state visits to complete the expansion process. If the *connectivity* information faithfully showing the path leading to a particular state from a given state is stored, the problem of limited memory capacity becomes appar-

ent. It is clear that the state space grows exponentially with the complexity of the protocol and the number of entities in the validation model. A quantitative analysis of this technique for verifying small- to medium-size protocols is presented in [8].

3.1.1 Pruning the State Space by Counting Equivalence

To keep the state space manageable, pruning of redundant states is necessary. Two system states $(q_1, q_2,..., q_n)$ and $(s_1, s_2,..., s_n)$ are strictly equivalent if and only if $q_i = s_i$, $q_i, s_i \in Q$, for all $1 \leq i \leq n$. This strict equivalence relation is certainly too conservative. As we mentioned before, all cache entities' behavior is characterized by a common FSM with deterministic transition functions. All $n!$ permutations of a state $(q_1, q_2,..., q_n)$ are equivalent in the validation process because the order of the tuple is not important. For example, in the validation of a system with three caches, the tuples *(shared, shared, invalid)* and *(shared, invalid, shared)* should represent equivalent states.

This equivalence based on symmetry arguments suggests a concrete way to represent a set of equivalent states by the number of caches in each state. A state for a system with n caches can be represented as $\Pi_{i=1}^{|Q|} q_i^{k_i}$, where k_i denotes the number of caches in state $q_i \in Q$ and $\Sigma_{i=1}^{|Q|} k_i = n$.

Definition 5 (Counting Equivalence) *Two system states* $\Pi_{i=1}^{|Q|} q_i^{k_i}$ *and* $\Pi_{i=1}^{|Q|} q_i^{l_i}$ *are equivalent if* $k_i = l_i$ *for all* i.

The relaxed equivalence of Definition 5 is a first step in the right direction. Equivalence classes can be further broadened.

3.2 Symbolic Expansion

The conventional expansion algorithm has three efficiency problems: the time-consuming operation of comparing states, the efficiency of convergence (termination) of the expansion process, and the large consumption of memory resources. Searching through the history and working lists against the new state becomes intolerable for large state spaces. The number of visited states that can be maintained is also limited by the memory size.

Another technical problem is the number of caches to model in a validation procedure. It is not clear at first that a protocol correct for a system with n caches would also be correct for a system with n' caches, $n \neq n'$. The verification procedure should deal with any n in order to validate the protocol for any system.

3.2.1 Composite State

To some extent, the number of caches in a particular cache state plays an important role in judging whether or not a system state is permissible. For example, several caches in the *Dirty* state signals data inconsistency. Similarly if a cache is in *Shared* state, the local copy is clean and *perhaps* present in other caches. In theory, an infinite number of caches could have clean copies without affecting the protocol correctness. In all these cases, the actual number of copies is really unimportant. What is critical in all protocols that we are aware of is whether there are 0, 1 or several copies in a given state[2]. These possibilities can be represented by the following set of repetition operators.

Definition 6 (Repetition Operator)

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

In a *composite* system state, caches in the same state are grouped into a cache state *class* and specified by one of the above repetition operators. For example, a composite state $(q_1^*, q_2^+, q_3,...)$ indicates that there are exactly one cache in state q_3, one or more caches in state q_2 and none or some caches in state q_1. Formally, the definition for a composite state is as follows.

Definition 7 (Composite State) *A composite state represents the composition of cache states in a system with an arbitrary number of cache entities. It is constructed over cache state classes of the form $(q_1^{r_1}, q_2^{r_2}, ..., q_n^{r_n})$, where $n = |Q|$ and $r_i \in [0, 1, +, *]$*[3].

The representation of a composite state carries all the information needed to verify the protocol. For example, in the Illinois protocol, there are two possible sources of data inconsistencies.The first possibility is that cache(s) in the *Shared* state coexist with a cache in the *Dirty* state and the second possibility is that more than one cache are in the *Dirty* state. It is clear that the first case can be formulated by the cache state classes appearing in the composite state (e.g. the composite state *(Dirty, Shared,...)*). The second case is cov-

2. Note that if a protocol was ever invented in which two dirty copies are permissible, the methodology is still applicable provided that we add this new possibility to the list of repetition operator.

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

ered by the repetition operator over state classes with multiple Dirty copies such as *(Dirty$^+$,...), (Dirty*,...)*. The definition of composite state expands state equivalence relations beyond the strict counting equivalence of definition 5.

3.2.2 Information Ordering and Pruning

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 $q \in Q$. q^+ reveals that one cache is in state q (which is always permissible) or that multiple caches are in state q (which may indicate a data inconsistency condition). The star operator adds the possibility of null instance. Therefore, we say that q^{r_1} is *weaker* than q^{r_2} if $r_1 < r_2$, where $q \in Q$ and $r_1, r_2 \in [1, +, *]$. The null instance can be ordered with respect to *, i.e., $0 < *$.

Definition 8 (Structural Covering) *We say that composite state S_2 structurally covers composite state S_1, or $S_1 \leq S_2$, if*

$$\forall q^{r_1} \in S_1 \quad \exists q^{r_2} \in S_2 \rightarrow q^{r_1} \leq q^{r_2} \text{ i.e. } r_1 \leq r_2$$

where r1, r2 are repetition operators.

Definition 9 (Containment) *We say that composite state S_2 contains composite state S_1, or $S_1 \subseteq_{\mathcal{F}} S_2$, if*

$$S_1 \leq S_2 \text{ and } \mathcal{F}(S_1) = \mathcal{F}(S_2)$$

where $\mathcal{F}$ is a characteristic function defined in the FSM model of Section 2.1.

The definition of *structural covering* extends the order between repetition operators to composite states. Furthermore, a composite state S_2 *contains* S_1 if S_1 is structurally covered by S_2 and if S_1 and S_2 have the same value of the characteristic function $\mathcal{F}$.

An interesting consequence of containment is that if $S_1 \subseteq_{\mathcal{F}} S_2$ 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 is a *monotonic* operator on the set of composite states S, that is, if $S_1 \subseteq_{\mathcal{F}} S_2$, then $\tau(S_1) \subseteq_{\mathcal{F}} \tau(S_2)$ where τ is an operator representing a cache 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 con-

tained in a new state are discarded. At the end of the expansion process all visited states are *essential* states.

Definition 10 (Essential State) *Composite state S is essential if and only if there does not exist a composite state* $\overline{S}$ *such that* $S \subseteq_{\mathcal{F}} \overline{S}$.

As a result, the state space at the end of the expansion process is simply decomposed into several families (which may be overlapping) represented by essential composite states.

3.2.3 Rules and Algorithm for the Expansion Process

We need now to define the set of operations applied to construct a composite state in the state generation process. In the following, the slash '/' symbol signifies "or" selection.

1. **Aggregation**:

   (a). $(q^0, q^r) \equiv q^r$.

   (b). $(q, q^{1/+/*}) \equiv q^+$.

   (c). $(q^+, q^r) \equiv q^+$.

   Aggregation rules are equivalence rules between composite states obtained by merging cache states.

2. **Coincident Transition**: ${q_1}^r \rightarrow^{\tau} {q_2}^r$, where $r \in [1, +, *]$ and τ is an observed transition originated by cache $\{C\text{: state}(C) \neq q_1\}$. It states that all caches in state q_1 change state coincidentally because of a transition originated by another cache not in state q_1.

3. **One-step Transition**: $q_1 \rightarrow^t q_2$ and ${q_1}^{+/*} \rightarrow^t (q_2, {q_3}^*)$, where t is a transition originated by cache $\{C\text{: state}(C) = q_1\}$. The next state of C is q_2 and all other caches in the state class q_1 change state to q_3. For example, one cache modifying its local copy in the *Shared* state changes to the *Dirty* state and the copies in all other caches in the same *Shared* state are invalidated.

4. **N-steps Transitions**: This rule specifies the repetitive application of the same transition N times, where N is an arbitrary positive integer.

   (a) $(Q, {q_1}^+)_{\mathcal{F}} \rightarrow^t (Q, {q_2}^1, {q_1}^*)_{\mathcal{F}} \rightarrow^t (Q, {q_2}^2, {q_1}^*)_{\mathcal{F}} \rightarrow^t \ldots \rightarrow^t (Q, {q_2}^+, {q_1}^*)_{\mathcal{F}}$, where t is a transition originated by cache $\{C\text{: state}(C) = q_1\}$ and $q_1 \rightarrow^t q_2$. It states that the same transition t can be applied infinitely many times as long as there are caches in state q_1, and every generated state has the same value defined over the characteristic function $\mathcal{F}$.

Every application of the transition brings down the number of caches in state q_1 by one and increases the number of caches in state q_2. The transition t has no effect on other caches denoted as Q in the tuple. One trivial example is a series consecutive replacements; another example is the case where $|Q| = 0$, q_1 is *Invalid* and q_2 is *Shared* following consecutive read misses.

(b). $(Q, q_1^+)_{\mathcal{F}} \rightarrow^t (Q, q_2^1, q_1^+)_{\mathcal{F}} \rightarrow^t \ldots \rightarrow^t (Q, q_2^+, q_1^+)_{\mathcal{F}} \rightarrow^t \ldots \rightarrow^t (Q, q_2^+, q_1^{1/0})_{\overline{\mathcal{F}}}$. The same interpretation is made as in (a), except that the derivation ends with a composite state of different value $\overline{\mathcal{F}}$. This rule is used to model the effect of the *sharing-detection* function introduced in Section 2.1.

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. Before we formalize the algorithm for symbolic state expansion and protocol verification, we first prove, as promised, the monotonicity of the expansion process.

Lemma 1 *The immediate successor $\overline{S}_1$ originated from state*

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

is structurally covered by $\overline{S}_2$ originated from state

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

if $r_j = \bar{r}_j$ for all $j \neq i$, the same cache event $t \in \Sigma$ is applied to S_1 and S_2, and $\mathcal{F}(S_1)=\mathcal{F}(S_2)$.

Proof: The proof is direct. First of all, conforming to the condition $\mathcal{F}(S1)=\mathcal{F}(S2)$, caches in the same state in S_1 and S_2 will get to the same next cache state under common cache event t. Let's ignore caches in state q_i first, that is, $\{S_1 - q_i\} \equiv \{S_2 - q_i^r\}$. Since both states are exactly the same, any transition t will bring them to the same state $S1' \equiv S2'$. Next, we augment S_1' and S_2' by adding the effect of applying t on caches in state q_i in S_1 and S_2. The only change is the number of caches in the particular state $\{q_j: q_i \rightarrow^t q_j\}$. It is obvious that $\{S_1: q_i \rightarrow^t q_j\} \leq \{S_2: q_i^{1/+/*} \rightarrow^t q_j^{1/+/*}\}$, i.e., $\{S_1: q_i\} \leq \{S_2: q_i^{1/+/*}\}$ results in $\{\overline{S}_1: q_j\} \leq \{\overline{S}_2: q_j^{1/+/*}\}$ after the transition. We can thus conclude that $\overline{S}_1 \leq \overline{S}_2$.

Lemma 2 *The claim $\overline{S}_1 \leq \overline{S}_2$ holds if $S_1 \subseteq_{\mathcal{F}} S_2$, that is, $r_j \leq r_j$ for all j and $\mathcal{F}(S_1)=\mathcal{F}(S_2)$.*

Proof: The result extends the conclusion of lemma 1 and the proof is similar.

The result of lemma 2 indicates that if $S_1 \subseteq_{\mathcal{F}} S_2$, the immediately reachable state $\overline{S}_1$ of S_1 is structurally covered ($\leq$) by the immediately reachable state $\overline{S}_2$ of S_2. To demonstrate the *monotonicity*, we still need to show that $\overline{S}_1$ and $\overline{S}_2$ have the same characteristic functions, that is, $\mathcal{F}(\overline{S}_1)=\mathcal{F}(\overline{S}_2)$. We prove this in Appendix A1.

Corollary 1 If $\mathcal{F}$ is null and $S_1 \subseteq_{null} S_2$, then for every $\overline{S}_1$ reachable from S_1 there exists $\overline{S}_2$ reachable from S_2 such that $\overline{S}_1 \subseteq_{null} \overline{S}_2$.

Proof: Because $\mathcal{F}$ is null, the transition functions depend only on local cache state and intended operations. By definition 9, the relation of containment $\subseteq_{null}$ is characterized by the relation of structural covering $\leq$ alone. As a result, the claim is just a recursive induction from lemma 2.

Corollary 1 demonstrates that protocols whose behavior does not depend on any characteristic function exhibits the *monotonous* property of our symbolic expansion process. Therefore, during the expansion process, S_1 can be discarded provided we can demonstrate this monotonous property because all successors originated from S_1 can be generated by expanding the family originated from S_2.

Corollary 2 Protocols employing the *sharing-detection* function also exhibit the monotonicity of expansion. If $\mathcal{F}$ is the *sharing-detection* function and $S_1 \subseteq_{\mathcal{F}} S_2$, then for every $\overline{S}_1$ reachable from S_1 there exists $\overline{S}_2$ reachable from S_2 such that $\overline{S}_1 \subseteq_{\mathcal{F}} \overline{S}_2$.

Proof: Direct consequence of the proof in Appendix A.1.

The preceding results suggest a very efficient expansion process to obtain essential states as shown in Figure 3. Two lists are used to keep non-expanded and visited states. At each step, a new state is produced by expanding the current state, and then a pruning process based on the realization of monotonicity 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, as we now show.

Theorem 1 *The essential composite states generated by the proposed algorithm are complete. They symbolically characterize all states which can be produced by an exhaustive expansion process such as the algorithm depicted in Figure 2.*

Proof: The number of caches needs to be explicitly specified in the validation model for an enumeration approach, while the symbolic approach employs canonical form to characterize states. Consider states u, v (derived from u by transition τ) in the enumera-

tion approach and composite states s, t (derived from s by transition τ) in the symbolic form and s symbolically characterize u. t also characterizes v, because the same transition functions are applied and the information is accumulated during the generation of composite states.

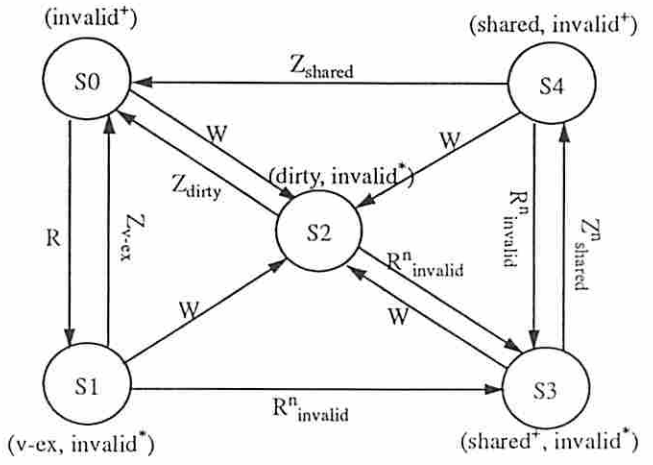
```
Algorithm : essential states generation.

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' ⊆_ℱ P or A' ⊆_ℱ Q or A' ⊆_ℱ A)
                  then discard A'.
               else begin
                       remove P from W if P ⊆_ℱ A'.
                       remove Q from H if  Q ⊆_ℱ A'.
                       add A' to W.
                       if (A ⊆_ℱ A') then discard A  and terminate
                         all FOR loops starting a new run.
                    end
           end
       end
   end
end.
```

FIGURE 3. Algorithm for generating essential states.

4.0 Verification of the Illinois Protocol

We have coded the algorithm of Figure 3 and, in this section, we take as example the Illinois protocol to demonstrate the symbolic expansion algorithm. The other four protocols described in [1] are verified in Appendix B. We start the expansion process by an initial state ($Invalid^+$), in which no cache has a block copy. After a small number of expansion steps, the essential states and the global state transition diagram are reported. The global transition diagram built upon the symbolic essential states highlights our tech-

nique and facilitates the verification of the data consistency and the demonstration of similarities and disparities between protocols.



state	sharing ($\mathcal{F}$)	cdata	mdata
(Invalid$^+$)	(false)	(nodata)	fresh
(V-Ex, Invalid$^+$)	(false, true)	(fresh, nodata)	fresh
(Dirty, Invalid*)	(false, true)	(fresh, nodata)	obsolete
(Shared$^+$, Invalid*)	(true, true)	(fresh, nodata)	fresh
(Shared, Invalid$^+$)	(false, true)	(fresh, nodata)	fresh

FIGURE 4. The Illinois Global Transition Diagram.

After 22 state visits (see Appendix A.2), five essential states, (*Invalid*$^+$), (*V-Ex*, *Invalid**), (*Dirty*, *Invalid**), (*Shared*$^+$, *Invalid**) and (*Shared*, *Invalid*$^+$) are reported and the global transition diagram is shown in Figure 4, where *R*, *W* and *Z* stand for read, write and replacement respectively. An optional subscript identifies the state of the cache originating the transition. A superscript *n* means that the transition is the result of an **N-steps transitions**. The values of the sharing-detection function and of the auxiliary variables showing the status of the cached data copies and of the memory copy are listed. It is clear that the data consistency requirement is satisfied because processors always have the most recent data values if a valid copy is in their cache, as shown in the table in Figure 4.

We need to distinguish between states *s3*:(*Shared*$^+$, *Invalid**) and *s4*:(*Shared*, *Invalid*$^+$) because of different observations of data sharing from the perspective of caches in state class *Shared*. *s4* is structurally covered by *s3* but is not contained in *s3*. By the definition of the *sharing-detection* function in Section 2.1, the boolean values returned are

(*true*, *true*), (*false*, *true*) for *s3* and *s4* respectively. The state *s0:*($Invalid^+$) is also an essential state and is not contained by another state for the same reason.

The definition of the *plus* (+) operator needs some explanation when the *sharing-detection* function $\mathcal{F}$ is employed. In fact, $Shared^+$ in state *s3*: ($Shared^+$, $Invalid^*$) with $\mathcal{F}(s3)$ = (*true*, *true*) denotes that there are at least two caches in the *Shared* state when *s3* is first constructed. This does not violate the original definition of the *plus* operator because the additional information is carried by the value of $\mathcal{F}$ associated with the composite state. We use the additional information carried by the value of $\mathcal{F}$ to distinguish between *s3* and *s4*. Certainly, we could explicitly express the effect of $\mathcal{F}$ by defining another operator that denotes two or more caches in a particular cache state. However, more expansion rules would be introduced and more essential states would be generated. On the other hand, the original definition of the *plus* operator provides a general notation and the characteristic function $\mathcal{F}$ shows the subtle difference.

States *s3* and *s4* are also reported for the Firefly protocol (see Appendix B), however, these two protocols do have different behavior. Following a write in the block in the *Shared* state in (*Shared*, $Invalid^+$) the system moves into state (*Dirty*, $Invalid^*$) for the Illinois protocol and (*Exclusive*, $Invalid^*$) for the Firefly protocol. The Firefly protocol employs a write broadcast scheme to keep copies consistent as long as multiple data copies exist, that is, the detection of data sharing is done whether or not the block is present in the cache. On the other hand, the Illinois protocol needs the sharing information only when a read miss occur, i.e. at a time when the block is not present in the cache. So, the *sharing-detection* function for the Illinois protocol could be changed to

$$f(C_1, C_2, \ldots, C_n) = \begin{cases} true & if\ \exists C_i \text{ s.t. } state(C_i) \neq invalid \\ false & \text{otherwise} \end{cases}$$

rather than the function introduced in Section 2.1. This would further--albeit marginally--simplify the global state diagram of the Illinois protocol because the state (*Shared*, $Invalid^+$) would then be contained by ($Shared^+$, $Invalid^*$) after the refinement.

5.0 Conclusion

In this paper, we have introduced a simple method for validating cache coherence protocols at the behavior level. By exploiting equivalence relations among global states, we can symbolically represent and generate the system state space rather then enumerate it. The reduction in complexity of the verification procedure over existing approaches is so drastic that we can contemplate efficient verification of much more complex protocols

with large number of cache states, such as relaxed consistency protocols, protocols with locked states and protocols for hierarchically organized machines.

We did not find any consistency problem in the five protocols that we have have examined. This was somewhat expected. The protocols are relatively simple and have been time-tested (ie, we would know by now if there was a problem.)

Extension of our work beyond more complex protocols could be the definition of a formal specification language capable of describing both the protocol behavior and the processes implementing it. This formal specification model should facilitate greater automatisation of the verification activities, which would reduce the possibility of errors. It is also possible to extend the model and the terminology to include more details of the protocol implementation having to do with ordering and consistency models.

References

[1] J. Archibald and J.-L. Baer " Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model ", *ACM Trans. on Computer Systems*, Vol.4, No4, Nov. 1986, pp. 273-298.

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

[3] G.V. Bochmann " Combining Assertions and States For the Validation of Process Communication ", *Proc. of the IFIP*, 1978.

[4] G.V. Bochmann and J. Gecsei " A Unified Method for the Specification and Verification of Protocols ", *Proc. of the IFIP*, 1977, Toronto.

[5] G.V. Bochmann and C.A. Sunshine " Formal Methods in Communication Protocol Design ", *IEEE Transactions on Communications*, Vol. COM-28, No. 4, Apr. 1980, pp. 624-631.

[6] L.M. Censier and P. Feautrier " A new solution to coherence problems in multicache systems ", *IEEE Trans. on Comp.*, C-27.12, Dec. 1978, pp. 1112-1118.

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

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

[9] R.H. Katz, S.J. Egger and *et al.* " Implementing a Cache Consistency Protocol ", *Proceedings of the 12th International Symposium on Computer Architecture*, 1985, pp.276-283.

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

[11] K. Tarnay " Protocol Specification and Testing ", 1991, Plenum Press, New York.

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

[13] Brent T. Hailpern " Verifying Concurrent Processes Using Temporal Logic ", *Lecture Notes in Computer Science,* Springer-Verlag, New York, 1982.

[14] E. M. Clarke, E. A. Emerson and A. P. Sistla " AutomaticVerification of Finite-State Concurrent Systems using Temporal Logic Specifications ", *ACM Trans. on Programming Languages and Systems,* Vol. 8, No. 2, April 1986.

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

[16] Ashwini K. Nanda and Laxmi N. Bhuyan " A Formal Specification and Verification Technique for Cache Coherence Protocols ", *Proceedings of the 1992 International Conference on Parallel Processing*, pp. I-22-I-26.

Appendix A

A.1 Proof of Corollary 2.

The proof is not formal but is based on the enumeration of all possible scenarios. The sketch of the proof is as follows.

Proof: By lemma 1 and lemma 2, we already know that $\overline{S}_2$ must structurally cover $\overline{S}_1$, that is, $\overline{S}_1 \leq \overline{S}_2$. It further remains to show that $\mathcal{F}(\overline{S}_1)=\mathcal{F}(\overline{S}_2)$. Consider the value of $\mathcal{F}$ returned by a composite state; there are three possibilities:

1. ***v1 = (false, false,...., false)***: no cached copy exists and the only possible composite state is $(Invalid^{+})$.
2. ***v2 = (true, true,..., true, false)***: one and only one cached copy exists. The composite state must have the form as $(Invalid^{+/*}, q)$, where $q \in (Q\text{-}\{Invalid\})$.
3. ***v3 = (true, true,..., true)***:there are several valid copies in the caches. The composite state has the structure $(Invalid^{+/*}, q_1^{r_1}, q_2^{r_2}, \ldots, q_m^{r_m})$, where $q_i \in (Q\text{-}\{Invalid\})$ for all i, and there must exist either a q_i with $r_i = +$, or *arbitrary* q_i, q_j with $r_i = r_j = 1$, or any combination of above.

Since $S_1 \subseteq_{\mathcal{F}} S_2$, S_1 and S_2 return the same values of $\mathcal{F}$, either ***v1***,***v2***, or ***v3***.

Case 1: $\mathcal{F}(S_1) = \mathcal{F}(S_2) =$ ***v1***.

In this case, S_1 and S_2 are identical and their successive states $\overline{S}_1$ and $\overline{S}_2$ are also identical, which implies $\mathcal{F}(\overline{S}_1)=\mathcal{F}(\overline{S}_2)$.

Case 2: $\mathcal{F}(S_1) = \mathcal{F}(S_2) =$ ***v2***.

Either S_1 and S_2 are identical or $S_1=(Invalid^{+}, q)$ is contained by $S_2 = (Invalid^{*}, q)$. In both cases, the same transition τ results in $\overline{S}_1$ and $\overline{S}_2$ having the same value of $\mathcal{F}$. This could be shown by contradiction. Assume that $\mathcal{F}(\overline{S}_1) =$ ***v1*** and $\mathcal{F}(\overline{S}_2) \neq$ ***v1***. This can happen only if τ causes $(q \rightarrow^{\tau} Invalid, Invalid \rightarrow^{\tau} Invalid)$ in S_1 and τ has different effects on S_2, which is impossible because $S_1 \subseteq_{\mathcal{F}} S_2$. Similar contradictions could be shown to refute the other two cases, say $\mathcal{F}(\overline{S}_2) \neq \mathcal{F}(\overline{S}_1) =$ ***v2***, $\mathcal{F}(\overline{S}_2) \neq \mathcal{F}(\overline{S}_1) =$ ***v3***, and thus we could conclude $\mathcal{F}(\overline{S}_1)=\mathcal{F}(\overline{S}_2)$.

Case 3: $\mathcal{F}(S_1) = \mathcal{F}(S_2) = v3$.

Let's consider the values of $\mathcal{F}$ returned by the generated composite state $\overline{S}_1$.

(a). $\mathcal{F}(\overline{S}_1) = v1$. This can happen if the transition τ is repeatedly applied to S_1 on caches in state q with $(q \rightarrow^{\tau} Invalid)$ and S_1 has the structure $(Invalid^{+/*}, q^{+})$. S_2 should have the same structure and result in $\mathcal{F}(\overline{S}_2) = v1$ because the transition τ removes all cached copies in S_1 and it will have the same effect when applied to S_2, and thus $\mathcal{F}(\overline{S}_1) = v1$.

(b). $\mathcal{F}(\overline{S}_1) = v2$. This can happen if a transition τ is applied to S_1 and removes all cached copies except in the cache which originates τ. Because $S_1 \subseteq_{\mathcal{F}} S_2$, τ must have the same effects when applied to S_2, and hence $\mathcal{F}(\overline{S}_2) = v2$.

(c). $\mathcal{F}(\overline{S}_1) = v3$. In this case, $\overline{S}_1$ carries the information that two or more than two cached copies exist. Since we know that $\overline{S}_1 \leq \overline{S}_2$, or $\overline{S}_1$ is structurally covered by $\overline{S}_2$, two or more than two cached copies exist in $\overline{S}_2$, that is, $\mathcal{F}(\overline{S}_2) = v3$.

From the above, we can conclude that $\mathcal{F}(\overline{S}_1) = \mathcal{F}(\overline{S}_2)$ in all cases.

A.2 Expansion Steps for The Illinois Protocol.

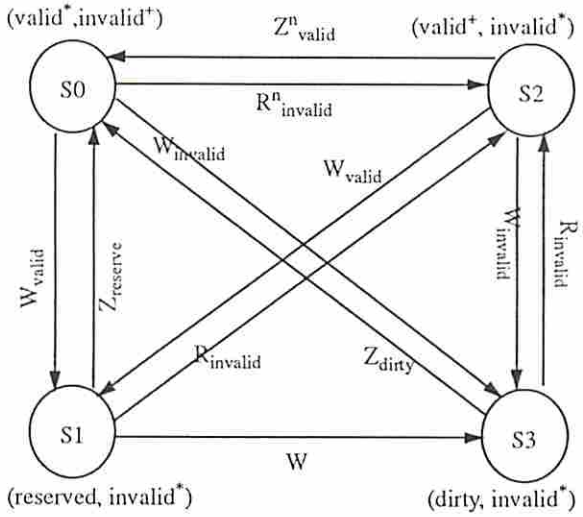
The intermediate steps for exploring the Illinois state space are listed below.

$$
\begin{array}{lll}
(Inv^{+}) \rightarrow & W_{inv} & \rightarrow (Dirty, Inv^{*}) \\
(Inv^{+}) \rightarrow & R_{inv} & \rightarrow (V\text{-}Ex, Inv^{*}) \\
(Dirty, Inv^{*}) \rightarrow & Rep_{dirty} & \rightarrow (Inv^{+}) \\
(Dirty, Inv^{*}) \rightarrow & W_{dirty} & \rightarrow (Dirty, Inv^{*}) \\
(Dirty, Inv^{*}) \rightarrow & R_{dirty} & \rightarrow (Dirty, Inv^{*}) \\
(Dirty, Inv^{*}) \rightarrow & W_{inv} & \rightarrow (Dirty, Inv^{+}) \\
(Dirty, Inv^{*}) \rightarrow & R^{n}_{inv} & \rightarrow (Shared^{+}, Inv^{*}) \\
(V\text{-}Ex, Inv^{*}) \rightarrow & Rep_{v\text{-}ex} & \rightarrow (Inv^{+}) \\
(V\text{-}Ex, Inv^{*}) \rightarrow & W_{v\text{-}ex} & \rightarrow (Dirty, Inv^{*}) \\
(V\text{-}Ex, Inv^{*}) \rightarrow & R_{v\text{-}ex} & \rightarrow (V\text{-}Ex, Inv^{*}) \\
(V\text{-}Ex, Inv^{*}) \rightarrow & W_{inv} & \rightarrow (Dirty, Inv^{+}) \\
(V\text{-}Ex, Inv^{*}) \rightarrow & R^{n}_{inv} & \rightarrow (Shared^{+}, Inv^{*}) \\
(Shared^{+}, Inv^{*}) \rightarrow & Rep^{n}_{shared} & \rightarrow (Shared, Inv^{+}) \\
(Shared^{+}, Inv^{*}) \rightarrow & W_{shared} & \rightarrow (Dirty, Inv^{*}) \\
(Shared^{+}, Inv^{*}) \rightarrow & R_{shared} & \rightarrow (Shared^{+}, Inv^{*}) \\
(Shared^{+}, Inv^{*}) \rightarrow & W_{inv} & \rightarrow (Dirty, Inv^{+}) \\
(Shared^{+}, Inv^{*}) \rightarrow & R^{n}_{inv} & \rightarrow (Shared^{+}, Inv^{*}) \\
(Shared, Inv^{+}) \rightarrow & Rep_{shared} & \rightarrow (Inv^{+}) \\
(Shared, Inv^{+}) \rightarrow & W_{shared} & \rightarrow (Dirty, Inv^{+}) \\
(Shared, Inv^{+}) \rightarrow & R_{shared} & \rightarrow (Shared, Inv^{+}) \\
(Shared, Inv^{+}) \rightarrow & W_{inv} & \rightarrow (Dirty, Inv^{+}) \\
(Shared, Inv^{+}) \rightarrow & R^{n}_{inv} & \rightarrow (Shared^{+}, Inv^{*})
\end{array}
$$

Appendix B The Verification of all Protocols

B.1 The Write-Once Protocol

The Write-Once protocol is mainly characterized by the introduction of the *Reserved* cache state. A first time write to a clean and potentially shared block is a write-through to memory and it updates the main memory copy as well as the local copy. The local copy becomes *Reserved,* which indicates an exclusive copy in the system and saves following write invalidations. The result of applying our algorithm to the write-once protocol is shown in Figure 5. A processor always accesses fresh data, and therefore data consistency is maintained.

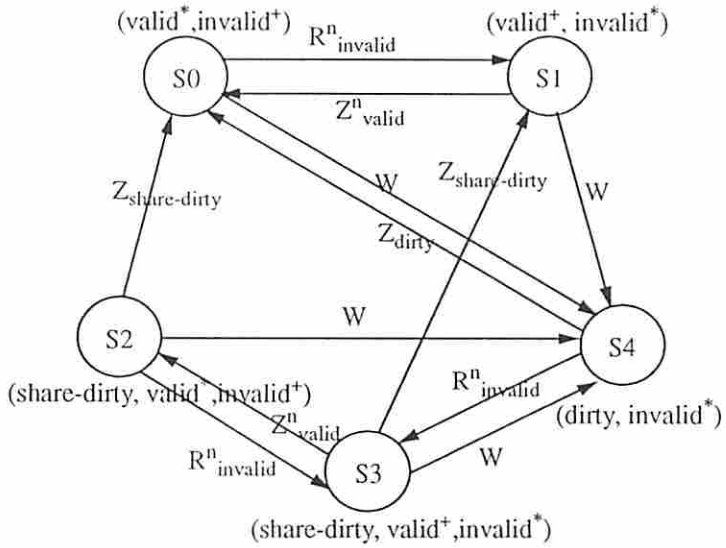


state	cdata	mdata
(valid*, invalid+)	(fresh, nodata)	fresh
(reserved, invalid*)	(fresh, nodata)	fresh
(valid+, invalid*)	(fresh, nodata)	fresh
(dirty, invalid*)	(fresh, nodata)	obsolete

FIGURE 5. The Write-Once Protocol after 22 State Visits.

B.2 The Berkeley Protocol

The Berkeley protocol introduced the notion of *ownership*. On a cache miss, the required data is always supplied by the owner, who has the latest data. In the Berkeley protocol, caches in *Dirty* or *Shared-Dirty* states are owner; otherwise, the main memory is the owner. It is a write-invalidate protocol and the memory copy is updated using a write-back policy when a copy in states *Dirty* or *Shared-Dirty* are victimized.The global state transition diagram is shown as Figure 6. A block can be in state *Shared-Dirty* in only one cache, but it may also be present in state *Valid* in other caches.

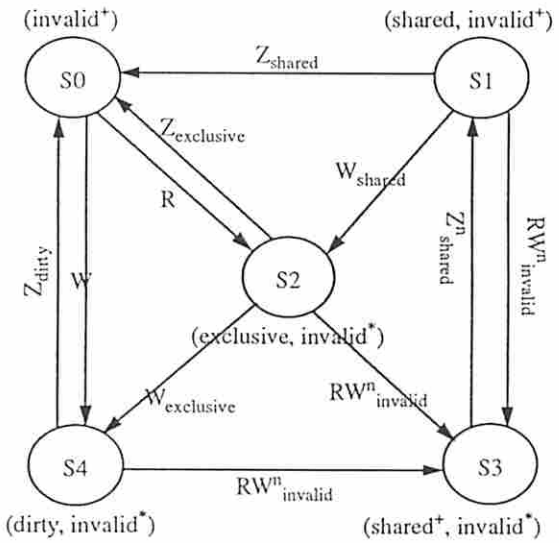


state	cdata	mdata
(valid*, invalid+)	(fresh, nodata)	fresh
(valid+, invalid*)	(fresh, nodata)	fresh
(shared-dirty, valid*, invalid+)	(fresh, fresh, nodata)	obsolete
(shared-dirty, valid+, invalid*)	(fresh, fresh, nodata)	obsolete
(dirty, invalid*)	(fresh, nodata)	obsolete

FIGURE 6. The Berkeley Protocol after 33 State Visits.

B.3 The Firefly Protocol

Figure 7 shows the global state transition diagram for the Firefly protocol. The Firefly protocol is a write-broadcast with dynamic detection of sharing. As long as there exists more than one cached copy, writes are broadcast to memory and to other caches. The *Exclusive* state indicates that no other cache has a copy and that writes need no longer be broadcasted. A write-back occurs when a dirty copy is selected for replacement.



state	sharing($\mathcal{F}$)	cdata	mdata
(invalid$^+$)	(false)	(nodata)	fresh
(shared, invalid$^+$)	(false, true)	(fresh, nodata)	fresh
(exclusive, invalid*)	(false, true)	(fresh, nodata)	fresh
(shared$^+$, invalid*)	(true, true)	(fresh, nodata)	fresh
(dirty, invalid*)	(false, true)	(fresh, nodata)	obsolete

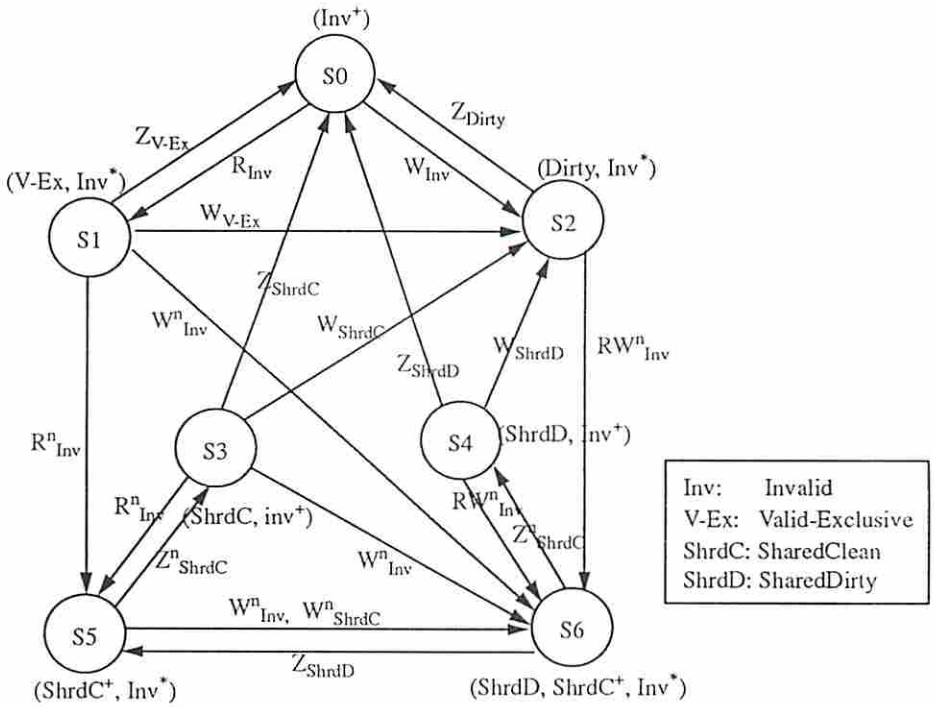
FIGURE 7. The Firefly Protocol after 22 State Visits.

B.4 The Dragon Protocol

In Fig. 8, the output of our algorithm is shown for the Dragon protocol. The Dragon protocol is a write-broadcast as the Firefly protocol . One difference is the updates

to shared blocks are not immediately reflected at main memory. The difference is the existence of the *SharedDirty* state.

The cache which performed the latest write to the shared block is in the *Shared-Dirty* state and is responsible for supplying the block on misses in remote caches and for updating main memory on replacement. A *SharedDirty* copy is not consistent with the main memory copy, which is apparent from the table in Fig. 8.



state	sharing(ℱ)	cdata	mdata
(Inv+)	(false)	(nodata)	fresh
(V-Ex, Inv*)	(false, true)	(fresh, nodata)	fresh
(Dirty, Inv*)	(false, true)	(fresh, nodata)	obsolete
(ShrdC, Inv+)	(false, true)	(fresh, nodata)	fresh
(ShrdD, Inv+)	(false, true)	(fresh, nodata)	obsolete
(ShrdC+, Inv*)	(true, true)	(fresh, nodata)	fresh
(ShrdD, ShrdC+, Inv*)	(true, true, true)	(fresh, fresh, nodata)	obsolete

FIGURE 8. The Dragon Protocol after 35 State Visits.