Verification Techniques for Cache Coherence Protocols

FONG PONG

Sun Microsystems Computer Corporation

MICHEL DUBOIS

University of Southern California

In this article we present a comprehensive survey of various approaches for the verification of cache coherence protocols based on *state enumeration*, (*symbolic*) *model checking*, and *symbolic state models*. Since these techniques search the state space of the protocol exhaustively, the amount of memory required to manipulate the state information and the verification time grow very fast with the number of processors and the complexity of the protocol mechanisms. To be successful for systems of arbitrary complexity, a verification technique must solve this so-called *state space explosion* problem. The emphasis of our discussion is on the underlying theory in each method of handling the state space explosion problem, and formulating and checking the *safety* properties (e.g., data consistency) and the *liveness* properties (absence of deadlock and livelock). We compare the efficiency and discuss the limitations of each technique in terms of memory and computation time. Also, we discuss issues of *generality*, *applicability*, *automaticity*, and *amenity* for existing tools in each class of methods. No method is truly superior because each method has its own strengths and weaknesses. Finally, refinements that can further reduce the verification time and/or the memory requirement are also discussed.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*shared memory*; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*formal models*; B.4.4 [**Input/Output and Data Communications**]: Performance Analysis and Design Aids—*formal models*; *verification*

General Terms: Verification

Additional Key Words and Phrases: Cache coherence, finite state machine, protocol verification, shared-memory multiprocessors, state representation and expansion

1. INTRODUCTION

A cache-coherent shared-memory multiprocessor system provides programmers with a logical view that all processors have access to a shared global memory. This illusion is transparent to programmers in spite of the fact that main memory storage may be physically distributed and multiple data copies of the same memory location may exist in private caches. Although efficient caching techniques can significantly reduce memory access latency and interconnec-

Authors' addresses: F. Pong, Sun Microsystems Computer Corporation, Technology Development Group, 2550 Garcia Ave., MS UMPK 15-214, Mountain View, CA 94043; email: ⟨fong.pong@eng.sun.com⟩; M. Dubois, Department of EE-Systems, University of Southern California, Los Angeles, CA 90089-2562; email: ⟨dubois@paris.usc.edu⟩.

tion traffic, they introduce the *cache coherence* problem. Multiprocessors with private caches need hardware or software support to enforce data consistency, otherwise inconsistent data copies may be observed when a processor modifies the data copy in its private cache [Stenström 1990]. In many cases, this support is provided by a *cache coherence protocol* that defines a set of rules coordinating processors, cache controllers, and memory controllers.

The verification of cache coherence protocols is an important subject that has been neglected for a long time. Many protocols have been proposed and implemented;[1] however, their correctness has never been formally validated. The main reason for this state of affairs is that most existing protocols are relatively simple *snooping* protocols that use broadcast of updates or invalidations to keep data copies consistent. Their correctness can be established by careful inspection, thorough analysis [Baer and Girault 1985; Rudolf and Segall 1984], or simple techniques such as testing and simulations [Galles and Williams 1994; Lenoski et al. 1990]. The lack of efficient verification tools is also a reason. However, the need for high-performance and scalable machines has made cache protocols much more complex today. As faster larger systems are designed and built, the complexity of cache protocols will continue to increase. It is becoming impractical to verify a cache protocol by hand or by random simulations because a random test sequence must be run indefinitely to enter all reachable states. In some studies [McMillan and Schwalbe 1991; Pong et al. 1995], it has been shown that simulations are not very reliable in practice and, thus, there is a need for more efficient and reliable tools. As a result of these compelling facts, recent research has focused on techniques that can verify protocols completely.

This article provides a survey of techniques to verify cache coherence protocols by exploring all the possible sequences of interactions between components in a given protocol model. We are particularly interested in methods with mechanical verification procedures, specifically, methods based on *state enumeration*,[2] (*symbolic*) *model checking* [Browne et al. 1986; Clarke et al. 1986; McMillan 1992], and *symbolic state model* [Pong 1995]. In these techniques the protocol is characterized by its *state* and the verification is based on searching all reachable states exhaustively. From a given state, the exploration of all possible interactions among protocol components leads to a number of new states. The expansion process continues and converges when all reachable states have been produced. The major differences among the techniques surveyed in this article stem from the ways of representing and pruning the state space in order to overcome the *state space explosion problem* [Holzmann 1990]. In general, the state exploration complexity quickly blows up in terms of computation time and memory requirement with the increasing number and complexity of components in the protocol. To deal with this complexity, *symmetries*, *regularities*, and *homogeneities* in cache-based systems must be exploited to reduce the size of the state space.

To illustrate the various techniques, we apply them to a simple protocol example under the assumption of atomic memory accesses. It should be noted that although the protocol example is simple, the approaches surveyed in this article have been applied successfully to much more complex protocols and have been shown to work reasonably well in practice. We show through the example how the *safety* properties (e.g., data con-

[1] Please see Archibald and Baer [1986], Censier and Feautrier [1978], Dubois et al. [1991], Haridi and Hagersten [1989], James et al. [1990], Lenoski et al. [1990], and Sweazey and Smith [1986].

[2] Please see Bochmann and Sunshine [1980], Danthine [1980], Holzmann [1990], and Ip and Dill [1993a, b].

sistency) and the *liveness* properties (absence of deadlock and livelock) are formulated and checked in each method. We also discuss issues of *generality*, *applicability*, *automaticity*, and *amenity* for existing tools implemented for each class of methods. Generality and applicability refer to the type of cache protocols and properties that can be checked; automaticity is the degree to which the verification is carried out mechanically and amenity indicates how easy it is to use the tools. Refinements of these techniques that can further reduce the verification time and/or the memory requirement are also discussed.

The article is structured as follows. In Section 2, we start with an overview of shared-memory systems, of the cache coherence problem, and of the main sources of complexity in designing and verifying cache protocols. Section 3 overviews the construction of tractable protocol models based on the *finite state machine* (FSM) at different levels of abstraction as well as the correctness issues in cache coherence protocols. Section 4 introduces the snooping protocol example to illustrate the various approaches, which are described in Sections 5 to 7. In Section 8, we summarize the strengths and the weaknesses of each method and show a performance comparison by applying the methods to a directory-based protocol. Finally, we overview some other related methods in Section 9 and conclude in Section 10.

2. RESEARCH BACKGROUND—SHARED-MEMORY MULTIPROCESSORS

In a shared-memory multiprocessor system, all processors share a global memory address space. The physical shared memory is generally organized according to three models: the *Uniform-Memory-Access* model (*UMA*), the *NonUniform-Memory-Access* model (*NUMA*), and the *Cache-Only Memory Architecture* (*COMA*). When processors are associated with private caches, the UMA and NUMA models are also called CC-UMA and CC-NUMA, respectively.

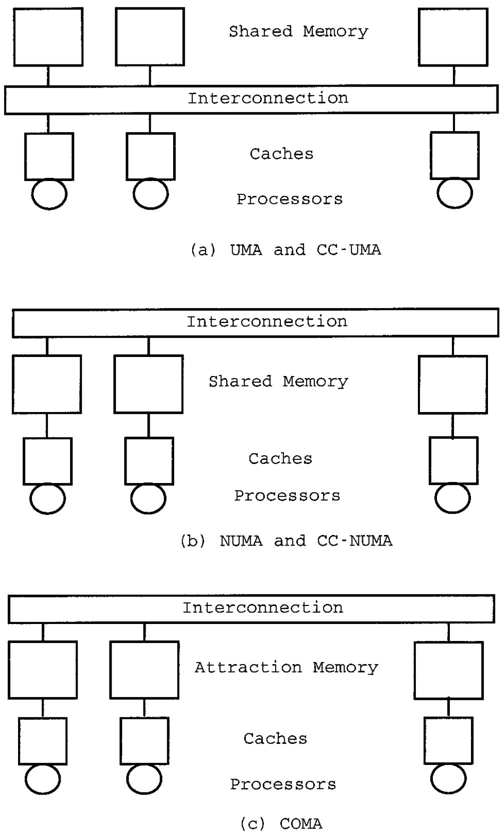


Figure 1. Shared-memory models: (a) UMA and CC-UMA; (b) NUMA and CC-NUMA; (c) COMA.

In the UMA model, the physical memory is uniformly shared by all processors (Figure 1(a)) and all processors have equal access time to every memory location. In a NUMA model, shown in Figure 1(b), each processor is physically associated with a fraction of the globally shared memory address space; thus the memory access time varies with the location of the memory word. The memory module that contains memory location *a* is commonly referred to as the *home* memory of *a* and the home memory is located at the *home node* or the *home* [Lenoski et al. 1990].

In the COMA model (Figure 1(c)), the distributed memory modules are referred to as the *attraction* memories [Haridi and Hagersten 1989] and act as caches of very large capacity. Data can be replicated freely in the attraction memories of different processor nodes, as if they were caches. A commercial COMA, the KSR-1, has also been called ALLCACHE [Rothnie 1992] to emphasize the fact that all memories behave as caches.

The per-processor caches are needed in all these models in order to reduce the long memory access latency and the network traffic. When the local processor generates a memory reference, it always checks its local cache first. If the data are found in the cache, the access is satisfied locally; otherwise, a cache miss occurs.

Due to the caching of data, multiple copies of the same memory location may exist in the system. When a processor modifies its local copy, *data inconsistency* may occur [Stenström 1990; Yen et al. 1985]. This is known as the *cache coherence* problem. Multiprocessors with private caches need hardware or software support to enforce data consistency. In many cases, support is provided by a *cache coherence protocol* that defines a set of rules coordinating processors, cache controllers, and memory controllers. In the CC-UMA model, coherence is maintained through a snoopy protocol on a bus. In the CC-NUMA model, coherence is maintained by directories centralized at the home node [Censier and Feautrier 1978; Lenoski et al. 1990] or distributed in a list linking all caches [James et al. 1990; Nowatzyk et al. 1994]. Coherence can sometimes be maintained in software; however, in this article, we only consider hardware-based protocols.

2.1 Cache Coherence Protocols

In all existing cache coherence protocols, several read-only copies of the same memory location can exist in the system at the same time. When multiple copies exist in different caches, they must be identical. We say that these copies are *Shared*. Usually, copies in the *Shared* states must be *Clean*, meaning that they are also identical to the memory copy. When a processor writes to the memory location, the protocol must guarantee the consistency of the copies. The two policies for maintaining data consistency are *write-invalidate* and *write-update*. In a write-invalidate protocol, a processor must invalidate all other copies before it can update its own copy. In this case, we say that the processor has an *Exclusive* and *Dirty* copy, and sometimes this processor is referred to as the *Owner* of the block. The owner must supply the copy of the block to any processor experiencing a miss. In a write-update protocol data consistency is maintained by updating all remote data copies instead of invalidating them.

Cache protocols must enforce that all processors observe all stores to the *same* memory location in the same order [Scheurich and Dubois 1987]. Memory accesses to the same memory location must be performed in program order. As a result, when all processors cease to write and all stores in propagation are performed, all data copies of the same memory location must be identical. This is the so-called *general coherency property* [Scheurich 1989], which is an essential requirement for correctly enforcing *memory consistency models* in a cache-based shared-memory multiprocessor system.

2.2 Memory Consistency Models

The *memory consistency model* refers to the logical model of memory access orderings offered by the memory system to the programmer or to the compiler. In a uniprocessor system, data are accessed by one processor, and hence, the memory system presents a very simple behavioral model to programmers—*a load access to a memory location always returns the value written by the latest store to the same memory location*. It is natural for programmers to extend the simple uniprocessor memory model for multiprocessor systems by imaging the executions of applications on a primitive architecture as shown in Figure 2.

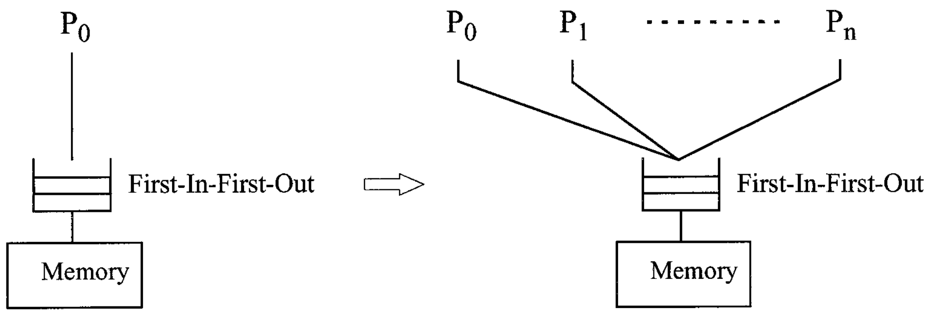


Figure 2. Natural transition between memory models for uniprocessor and multiprocessors.

In the multiprocessor model of Figure 2, there is a single centralized memory module and only one copy of data for every memory location. Accesses from processors are serially scheduled to the memory and are performed one at a time according to their order in the program. Although the model is direct and straightforward, it is very restrictive and has negative effects on system performance. For instance, concurrent reads are unnecessarily serialized.

A major challenge for system architects is to design more flexible memory systems supporting multiple data paths to the same memory location. Typical architectures providing this concurrency are the cache-coherent shared-memory models in Figure 1. In such systems, processors may update different cached copies of the same memory location. Since a store is no longer an atomic event and the propagation of stores to different memory locations can overlap in time, the notion of "latest store" has lost its meaning.

From the perspective of programmers, the execution of a concurrent program on a cache-coherent multiprocessor system must be indistinguishable from the results obtained from the idealized primitive architecture. Thus the most prominent memory model is called *Sequential Consistency* [Lamport 1979]. The sequential consistency model requires that the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each processor were executed in the order specified in the program.

Many researchers have developed conditions for implementing sequential consistency correctly in cache-coherent shared-memory systems. The most widely adopted set of (sufficient) conditions are:

(1) A write-access to a cache block is globally performed when all other cache copies are either invalidated or updated.
(2) A read access is globally performed when the store defining the value returned by the load is globally performed.
(3) Each processor globally performs its memory accesses in program order.

Although these conditions are sufficient but not necessary, they yield the most viable hardware solutions. Other conditions may require the system to support efficient broadcast devices [Afek et al. 1989; Brown 1990] or special hardware aids [Adve and Hill 1990b], and the performance gain may not be worthwhile relative to their increased hardware complexity.

Many memory consistency models such as *Weak Ordering* [Dubois and Scheurich 1990; Dubois et al. 1986], *Release Consistency* [Gharachorloo et al. 1990], and others [Sindhu et al. 1992] have recently been proposed to relax the hardware constraint of executing one memory access at a time in order to implement sequential consistency. They assume that all synchronizations among parallel threads are done through explicit, hardware-recognizable synchronization primitives. The software must be written so that all shared data that are not synchronization primitives are accessed in critical or semicritical sections enforced by explicit synchronization. The implication for weakly ordered memory systems is that a processor globally performs all its preceding loads and stores before it issues a synchronization operation, and that a processor issues no memory load or store following a synchronization point until the synchronization operation is successfully completed. Thus, in protocols under relaxed memory models, cache coherence can be *delayed* until the points of synchronizations [Dubois et al. 1991; Lenoski et al. 1990] and aggressive latency tolerance techniques [Dahlgren et al. 1994; Gharachorloo et al. 1991] can be exploited.

2.3 Latency Tolerance Techniques

Latency tolerance techniques refer to the mechanisms that can reduce the penalty of cache coherence events. Important techniques include *nonbinding prefetch*, *pipelined memory accesses*, *delayed consistency*, and *write posting*.

Nonbinding prefetch [Gupta et al. 1991] is a technique valid for all memory consistency models and which preloads data into local caches before processors actually access the data. Pipelined memory accesses [Gharachorloo et al. 1991], delayed consistency [Dubois et al. 1991], and write posting [Scheurich and Dubois 1991] are mechanisms reserved for systems with relaxed memory models. Because data consistency is only required at synchronization points in relaxed memory models, the processor is not blocked when a store misses in the cache. Stores are most often buffered and load hits can bypass pending stores [Lenoski et al. 1990]. At a synchronization point, all pending stores must be completed.

In the context of relaxed memory models, data consistency is not enforced on-the-fly. The effects of stores can be delayed until the next synchronization point. Moreover, processors can ignore received invalidations or updates until the next synchronization point. Such delays cut down on the coherence traffic. A protocol supporting delayed consistency was proposed by Dubois et al. [1991].

Write posting [Scheurich and Dubois 1991] is based on the fact that the processor does not need to be stalled if it always reads data stored by itself—when a write miss occurs, the modified word is made available to the local processor while the write miss is waiting for the completion of invalidations. As long as the processor reads the values created locally, it does not need to stall.

Although latency tolerance techniques are useful in improving processor efficiency and system scalability, they exacerbate the complexity of cache protocols. It becomes harder to predict the protocol behavior due to the fact that more concurrency is exploited and the verification of cache coherence protocols becomes a truly challenging problem.

2.4 Summary

Cache coherence is a critical requirement for correct memory behavior of a shared-memory system using private caches. However, the verification of cache coherence does not mean verification of correct memory orderings. The major impact of memory models on cache protocol design is to specify when cache coherence should be enforced, that is, when the effect of stores should propagate and when they should be taken into account at the receiving end. In the restricted sequential consistency model, cache coherence is maintained at each memory reference. On the other hand, more architectural optimization techniques can be exploited in relaxed models such as weak ordering and release consistency where cache coherence only needs to be enforced at synchronization points.

In the sequel of the article, we focus on the verification of cache protocols, or the cache coherence problem. We also limit our focus to protocols designed for CC-UMA or CC-NUMA systems, but not for COMA systems. In addition to the requirement of data consistency, a COMA system needs to ensure that a valid data copy of every memory location always exists somewhere in the system [Haridi and Hagersten 1989]. It should be noted, however, that the verification methods surveyed in this article can be adapted to verify COMA systems.

The state of the art in verification tools is that we can only verify properties related to coherence. Thus the verification of the memory consistency model is beyond the scope of this article. We assume that every processor node (including processor, cache, controllers, and latency tolerance hardware) issue read requests and write requests to the *same* block in an arbitrary order and we verify that coherence is maintained. We do not model the latency tolerance hardware in each processor, but stores are not necessarily atomic. One restriction is that only one request to the block per processor can be outstanding at a time (multiple requests to *different* blocks may be outstanding, however.) Coherence can also be verified when the model incorporates the latency tolerance hardware and, in the conclusion of this survey, we indicate how to do that.

3. VERIFICATION OF CACHE PROTOCOLS BASED ON THE FSM MODEL

The goal of a formal protocol verification procedure is to validate an abstract protocol model by proving that it adheres to a given specification. The specification is a list of correctness properties required from the protocol.

3.1 Model Abstraction and Specification Using FSMs

Although there is a variety of ways to specify a protocol model, we are interested in methodologies that employ finite state machines (FSMs) to form protocol models. Because cache protocols are essentially composed of component processes such as memory and cache controllers that exchange messages and respond to "events" generated by processors, a finite state machine model with such "events" as its inputs is a natural model. Specifically, we focus on verifying cache protocols where the behavior of an individual protocol component C is modeled as a finite state machine FSM_C and the protocol machine is composed of all FSM_Cs. Inputs to these machines are processor-generated events and messages for maintaining data consistency.

In general, the protocol models are abstracted representations. They are often kept simple to make the complexity of verification manageable, while preserving properties of interest. It is clear that the quality of a verification is only as good as the quality of the model. Therefore, a good compromise between quality and simplicity is necessary to make a verification model practically useful.

Cache protocols can be modeled at three different levels of abstraction (Table I). The highest level of abstraction aims at verifying the *behavior* of protocols (the composite behavior of all caches with respect to a given cache line). At this level, protocol transactions [Pong and Dubois 1993a] are assumed *atomic* and the interconnection network is not modeled. This behavioral model can be refined by modeling *nonatomic* memory accesses, at the *message passing* or *system* level. At the lowest abstraction level, called the *architectural* level, implementation details of the protocol are included in the model.

Most contemporary methodologies focus on verifying cache protocols at the intermediate system level for several reasons. First, the assumption of atomic

Table I. Model Abstraction at Different Levels

Level	Characteristics
Behavior	1. Atomic protocol transactions take zero state transition time. 2. Aggregate behavior of all caches is checked. 3. Interconnections are not modeled.
System / Message Passing	1. Non-atomic protocol transactions are specified by sequences of messages exchanged among protocol components. 2. FSMs for characterizing the behavior of protocol components (e.g., cache controllers) are augmented with transient states to model intermediate behavior. 3. Interconnection networks are abstracted by message channels.
Architecture	1. The model mimics the architecture implementing the protocol in details. 2. More hardware components such as message buffers are added.

memory accesses at the behavior level is far from realistic. Except for some snooping protocols implemented on a circuit-switched bus, atomicity of stores is generally not guaranteed. Second, verifying at the behavioral level is akin to verifying the protocol specification, which is not the same as verifying the implementation. The same protocol specification may be implemented differently by different manufacturers. Finally, it is practically infeasible to verify an entire, finely modeled implementation because of its unmanageable complexity.

On the other hand, validation of the protocol at the system level has moderate complexity and is adequate in most cases. The concurrency of memory events is nicely modeled by FSMs communicating through messages. Message passing between FSMs helps the understanding of interactions among protocol components. Moreover, a FSM modeled at the system level facilitates a hierarchical verification methodology. In a hierarchical scheme, the protocol is first verified at the message-passing level and is then broken down into smaller critical pieces for successive validations in lower abstraction levels. For example, at the system level, the model only deals with the outputs generated by a cache controller in response to input messages it receives. The implementation of the cache controller is hidden in a black box whose behavior is characterized by FSM_C. Later, the correspondence between inputs and outputs of FSM_C can be used as a basis for validating its hardware implementation.

3.2 Type of Protocol Errors

While verifying cache protocols against their specification, there are two basic *safety* properties ("bad things will never occur") and one *liveness* property ("good things will occur in the future") to verify. They are defined as follows.

—*Data consistency.* In a cache coherent system, data consistency among multiple data copies is typically enforced by allowing one and only one store in progress at any time for each block [Scheurich and Dubois 1987]. Concurrent accesses to the same block can be executed on different data copies but must appear to have executed atomically in some sequential order. The cache protocol must always return the latest value on each load.

—*Incomplete protocol specification.* When possible state transitions have been omitted in the protocol specification, it may happen that some protocol component receives a message that is not specified in its current state. Such *unexpected message reception* is an error condition. Since the message is not specified, the subsequent behavior of the protocol is not defined and is unpredictable.

—*Absence of deadlock and livelock.* A deadlock occurs when the protocol enters a state without possible exits, such that all FSMs remain indefinitely in their current state. Deadlock must be avoided because the system is blocked forever. On the other hand, a livelock is a situation where protocol components keep exchanging messages but the system is not making any useful progress. In the context of cache coherence protocols, it is customary to consider deadlock as a special case of livelock and to adopt the following definition for livelocks [McMillan and Schwalbe 1991; Pong 1995].

Definition 1. *(Livelock and Deadlock)* In the context of coherence protocols, a livelock (including deadlock) is a condition in which a given block is locked by one processor so that some processor is permanently prevented from accessing the block.

Given the preceding definition, a livelock (including deadlock) occurs when some processor is locked out from reaching the *shared* or the *exclusive* (*dirty*) state [McMillan and Schwalbe 1991]. It is important to notice that the preceding definition is derived from the service (functionality) supported by the cache protocols. It does not deal with livelocks and deadlocks due to particular implementation choices. For instance, deadlocks or livelocks are caused by finite message buffers or by unfair processing of memory requests. Such situations are not detected. Rather, we only consider *protocol-intrinsic* livelocks and deadlocks. A typical example is a processor waiting for a message that is never sent by another processor in the protocol specification [Pong 1995].

4. A SIMPLE PROTOCOL EXAMPLE

We illustrate the various techniques with the verification of a simple multiprocessor system with two coherent caches supported by a write-invalidate protocol (Figure 3). One single memory location is considered and it is assimilated to a single cache block. A cache block can be in three stable states: Invalid (I), Shared (S; clean copy potentially shared with other caches), and Dirty (D; modified and only cached copy).

The behavior of each cache is specified by FSMs C_1 and C_2 as shown in Figure 3. For the rest of this article, we closely investigate techniques to verify properties of the protocol over the *aggregate* behavior of C_1 and C_2. This aggregate behavior is characterized by the *protocol machine*. Although this protocol is simplistic, it is adequate for the purpose of illustration.

It is worthwhile mentioning that modeling a cache block with a single word is sufficient to verify the property of cache coherence for most existing protocols, because the granularity of coherence is a cache block. However, for protocols with word-based granularity of coherence [Dubois et al. 1991], a cache block with multiple words must be modeled [Pong and Dubois 1996]. In this case, the methodology must be adapted, but it remains fundamentally the same.

5. STATE ENUMERATION METHODS

5.1 Reachability or Perturbation Analysis

One important class of verification techniques is called *reachability* or *perturbation* analysis, which explores the system or global state space completely. Correctness properties of the protocol are verified on the set of reachable global states. The procedure starts with a model that uses finite state machines to describe the behavior of components in the protocol. A global state is defined as the composition of the states of all components. In our example, a global state, denoted as (s_1, s_2), where $s_1, s_2 \in$

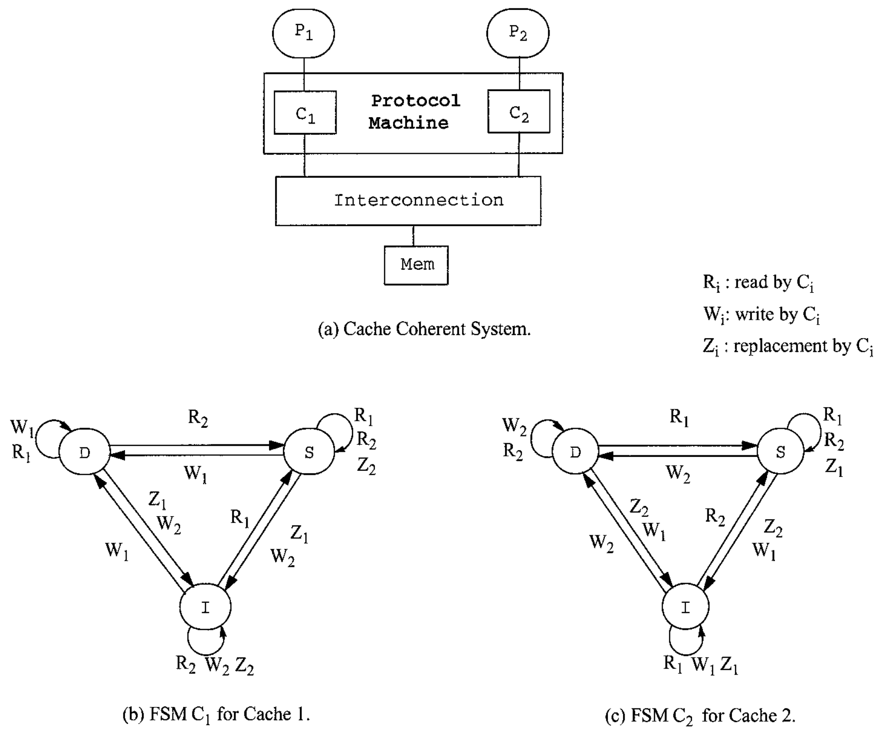


Figure 3. Coherent system with two caches: (a) cache coherent system; (b) FSM C_1 for Cache 1; (c) FSM C_2 for Cache 2.

{I, S, O}, is simply a composition of the states of the two caches.

A perturbation method is based on an exhaustive search algorithm (shown in Figure 4) to explore the system state space [Holzmann 1990]. In this algorithm, a working list (W) of newly produced states and a history list (H) of visited states are maintained. The expansion process starts with a given initial state, from which all possible transitions are exercised, leading to a number of new states. A new state is inserted into the working list if it is generated for the first time. This process is repeated for every new state until no new state is generated, as shown in the flattened *reachability* (*state*) graph of Figure 5. At the end, all reachable global states are contained in H.

```
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' $\notin W \cup H$)
         then add A' to W.
end.
```

Figure 4. Algorithm for exhaustive search.

The state expansion algorithm in Figure 4 implements an *interleaving* model; namely, every state transition is the result of the execution of a *single* memory event. Events such as processors issuing independent requests that may occur simultaneously are serialized and taken in different state expansion steps. Therefore all reachable states are generated one by one. Although the core idea of this technique is simple enough

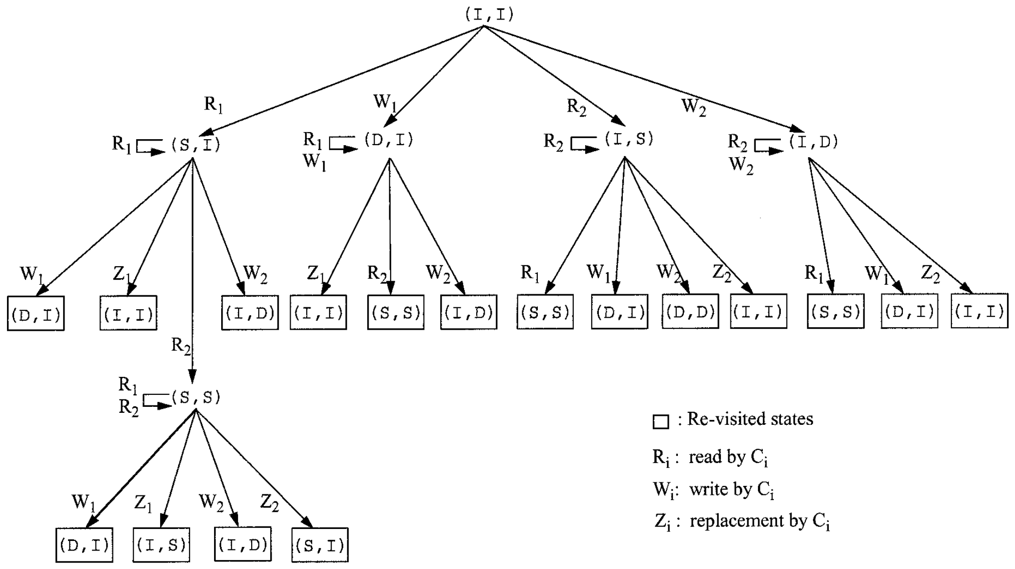


Figure 5. Reachability graph of protocol example.

to be fully automated, its applicability is limited to small scale models because of the state space explosion problem, as was shown in several studies [Pong et al. 1994; Pong 1995]. An excellent quantitative analysis of this technique is also given in Holzmann [1990].

5.2 Specification of the Protocol Model

An important feature of state enumeration methods is that they assume *full control* or *manipulation* of state variables. In the protocol model, state variables are declared, accessed, and modified as global variables in a *procedural* or *imperative* programming language such as C. When a state transition occurs, we can change the value of any state variable. This allows us to build abstract models more easily, without considering the details of a real implementation. For example, in the Illinois protocol [Archibald and Baer 1986], a read miss in a cache may return the cache block either in state *Valid-Exclusive* or *Shared*, depending on the presence of the block in other caches. These transitions can be easily modeled in a state enumeration method as shown in the pseudoprogram of Figure 6.

The model is made of n caches whose states and values of cached copies are kept in variables ($c_1, c_2, \ldots, c_n$) and $v_1, v_2, \ldots, v_n$), respectively. In the next-state transition rule shown in the example, when processor p_1 has a read miss in c_1, it makes the miss visible to other caches via the function *ExportCacheEvent*. In the function, all remote caches change their local cache states according to the finite state machine that specifies the cache behavior. When remote caches complete their operations, p_1 changes its cache state and loads the data either from a remote cache or from the memory. The function *ExportCacheEvent* represents some physical device such as a set of data and address lines and a polling bus wire that determines the sharing status of the data. In the model, all these bus wires are abstracted.

Because of the abstraction of protocol components by globally accessible state variables, the method is extremely flexible and thus very useful in the early

```
var
   $c_1, c_2, ...., c_n$ : {Invalid, Shared, Valid-Exclusive, Dirty};
   $v_1, v_2, ...., v_n$ : {0, 1}; // values of cached copies in $c_1, c_2, ...., c_n$
   $v_{mem}$: {0, 1}; // value of the memory copy
Function ExportCacheEvent(req-type, req-id): 0..n;
   if (req-type = read-access)
      for i : 1 to n
         if ((i != req-id)) // i : remote processor
            switch ($c_i$)
               case Dirty:
                        mem := $v_i$; data-provider := i; $c_i$ := Shared;
                        assert (OnlyCopy(i)) "dirty copy is not exclusive";
                        return (i);
               case Shared, Valid-Exclusive:
                        data-provider := i; $c_i$ := Shared;
                        return (i);
            endswitch;
         endif;
      endfor;
      return (0); // no cached copy in the system
   elseif (req-type = write-access)
            :::
End;

Rule "$p_1$ reads"
(1) // enable condition is always true: assume atomic access
==>
   switch ($c_1$)
      case Dirty, Shared, Valid-Exclusive: // read hits: no state change
      case Invalid:
            if ((id=ExportCacheEvent(read-access, 1)) != 0)
               $c_1$:=Shared;
               $v_1$:=$v_{id}$; // load the block from $c_{id}$
            else
               $c_1$:=Valid-Exclusive;
               $v_1$:=$v_{mem}$; // load the block from the memory
            endif;
   endswitch;
Endrule;
```

Figure 6. Pseudocode for state transition rule in a state enumeration model.

protocol design phase. The model can focus on the functionality of the protocol rather than on lower-level implementation details. By contrast, Nanda and Bhuyan [1992] specified and verified the Write-Once protocol [Archibald and Baer 1986] for a bus-based system of two caches by a state enumeration method. In their model, the behavior of the bus, cache controllers, and the memory controller are specified by finite state machines. These finite state machines are considered as *processes* in *Hoare's Communicating Sequential Processes* [Hoare 1978] language (CSP). The state space of the model is then explored by *composing* the bus, the caches, and the memory machines. Because of the composition of finite state machines, the model is somewhat complicated and must include many handshaking messages (ports in CSP) in order to interface the cache controllers and the memory to the bus. Inclusion of the bus in the model is purely for the purpose of synchronizing communications among the two caches and the memory. The bus-handshaking messages are not related to the protocol behavior or to the actual system; they are just there to serialize bus accesses in the model.

To serialize bus accesses in a state enumeration method, all we need is one state variable to model the bus; *bus_holder* in {*null*, p_1, p_2, . . . , p_n, *memory*}, given a system of n processors. When a processor wants to access the bus, all that is needed is a simple check to the variable *bus_holder* to detect whether the bus is free. Thus all handshaking messages between the bus controllers and cache controllers are abstracted. To abstract further, individual low-level operations (e.g., checking the bus status and sending a message if it is free) can be lumped and executed together if they are not critical to the protocol behavior. Due to this flexibility, protocol models can be rapidly modified as the protocol design evolves. In the early development phase of complex protocols, this is a considerable advantage of state enumeration methods.

The Murφ system implemented by Dill et al. [1992] is a well-known tool based on state enumeration. The Murφ specification language is purely *procedural* or *imperative* and has the same structure as the code shown in Figure 6. In Murφ, the collection of all state variables forms the global state. The protocol model is built on top of a set of next-state transition rules. Each transition rule is a *guarded* command, consisting of an enable condition and an action section in which values of next state variables are defined. Given a current state, the enable condition of a transition rule is checked. If it is true, a next state is generated by changing the values of state variables as specified in the rule. For error detection, Murφ checks for simple deadlock states which are states with no exit to states other than themselves. Murφ also checks violations of given inline assertions (e.g., the assertion "*assert* (*OnlyCopy*(*i*)) . . ." in the function *ExportCacheEvent* of

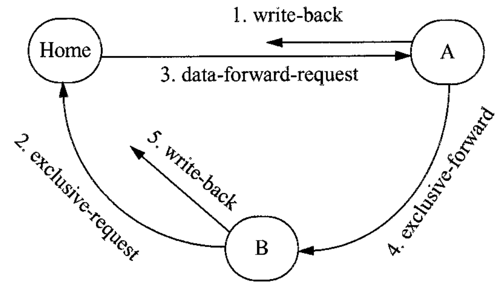


Figure 7. Stale write-back error and detection mechanism.

Figure 6) on the values of current state variables in rules and invariants for all reached states.

5.3 Detection of Protocol Errors

To verify the safety property of data consistency, a common approach is to show that all reachable global states are *permissible* [Nanda and Bhuyan 1992]. Usually, the definition of a cache state carries some semantic interpretation of the cached copy. For the protocol considered in this article as well as for many others [Archibald and Baer 1986; Sweazey and Smith 1986], a cache in the Shared state means that the cache has a clean copy consistent with the memory copy and that other caches may have a copy too. Similarly, a cache in the Dirty state indicates that it has the latest and sole cached copy. Therefore, global states (D,S), (S,D), and (D,D) are not permissible because they conflict with the definitions of the Shared and Dirty states. States that are not permissible are classified as *erroneous* states.

Unfortunately, this checking based on cache states is theoretically insufficient in general. The agreement between cache states and data copies must be guaranteed throughout the expansion process, which means that data values must be modeled along with state transitions. Consider the example of Figure 7 of a *stale write-back* error found in Pong et al. [1995]. Suppose that cache A has a dirty copy of the block, replaces it, and writes it back to the home node. In order to guarantee that the memory receives the block safely, cache A keeps a valid copy of the block until it receives an acknowledgment from memory. Meanwhile, cache B sends a request for an exclusive copy to home. Subsequently, cache A processes the *data-forward-request* from home, confuses it for an acknowledgment for the prior write-back request and sends the block to B. B then modifies its copy and replaces it, due to some other miss. As shown in the figure, a race condition exists between the two write-back requests. If the write-back from B wins the race, the stale write-back from A overwrites the values updated by B. Note that, in this example, all state transitions are permissible. A possible approach to keep track of values is to have processors randomly write one of two predetermined values such as 0 and 1; subsequently, a check verifies that processors do not read different values [Ip and Dill 1993a]. However, the stale write-back error might still go undetected unless the protocol model maintains a global variable to remember which write-back carries the latest value.

A systematic solution to such problems is suggested in Pong and Dubois [1993a]. Every cache is associated with a variable *cdata* which takes value from the set {*Nodata*, *Fresh*, *Obsolete*}; similarly, the status of the memory copy is indicated by a variable *mdata* which can be *Fresh* or *Obsolete*. When a data transfer is initiated from cache C_i to cache C_j, the value of $cdata_i$ is copied to $cdata_j$. When a processor performs a write, it defines a new fresh datum, and all other copies become obsolete. Violation of data consistency occurs when a processor can read an obsolete datum. For the scenario of Figure 7, when cache B performs its write it defines a *Fresh* value and the write-back message from cache A carries an *Obsolete* value. If the write-back by A overwrites the values updated by B, the memory value is *Obsolete*. Thus the state write-back error is easily detected.

The detection of unexpected message receptions is straightforward. Because protocols are described by FSMs, unexpected message receptions are merely unspecified inputs to the FSMs. In the rest of this article, we do not pay much attention to this type of error.

According to Definition 1, the absence of deadlock and livelock is normally proved by checking certain conditions: (1) there does not exist a state in which the protocol is trapped without exit to other states, and (2) every cache can reach the Shared and the Dirty states *infinitely* many times after a read miss and a write access, respectively. In practice, (2) is very difficult to check in a state enumeration method because there is no formal way to reason about the behavior of a protocol for an infinite number of transitions. To detect livelocks, the state graph must be kept and the transition relations (*connectivity* information [Holzmann 1990]) among global states must be maintained during the state expansion. At the end one can check whether the state graph is *strongly connected*, that is, whether a path exists between every pair of states [Archibald 1987; Pong et al. 1994]. Also, given a state in which a processor is in the Invalid state, there must exist a reachable state in which the processor is in the Shared state or the Dirty state after a read miss or a write access, respectively. When the state graph is large, the traversal of the graph is a costly operation in terms of computation time and memory consumption.

5.4 Improvements to Perturbation Methods

To overcome the inefficiency and large memory requirement of state enumeration methods, several variations have been proposed. In this section, we give an insight into these techniques. Although some of these techniques have not been applied to cache coherence protocols, they are general enough to be considered in this survey.

Before we continue the discussion, we distinguish between the *search state* *space* and the *system state space*. The search state space is the set of states generated during the process of state expansion, whereas the system state space is the set of reachable states accumulated and recorded in memory. Usually, the size of the search space is several orders of magnitude larger than the size of the system space because a global state is often produced more than once during the state expansion, as shown in Figure 5. Clearly, the verification time is dominated by the size of the search space and the memory requirement by the size of the system state space. To shorten the verification time, we need to reduce the number of search states and/or speed up the state comparison operations. On the other hand, memory can be saved by (1) exploiting *equivalence* relations among states so that a set of equivalent states is represented by a canonical state (Section 5.4.3), or (2) verifying the protocol "on-the-fly" as explained next.

5.4.1 *On-the-Fly Stack Search, State Space Caching, and Sleep Set.* On-the-fly stack search guarantees full exploration of the system state space while limiting the memory consumption by keeping track of states on the current expansion path only [Holzmann 1990; Jard and Jeron 1991]. A *depth-first* expansion only keeps states on the current expansion path whose length is bounded by the depth of the state space graph (Figure 8). The initial state is pushed onto the working list managed as a stack. At each step, one successor state *s* of the top state *q* on the stack is generated. If *s* is first-time generated, it is pushed on the stack and a recursive call to the algorithm is made, where *s* becomes the top state on the stack. If *s* has been generated before, either another expansion path originated from *q* is taken or *q* is removed from the stack and the algorithm backtracks to the predecessor of *q*.

```
W: list of working states managed as a stack.

push the initial state to W.

Stack_Search ()
begin
   q=Top(W)
   for each successor state s of q
      if (s ∉W)
       begin
         push s onto W
         Stack_Search()      /* recursion */
         delete s from W
       end
end.
```

Figure 8. Algorithm for on-the-fly stack search.

Although on-the-fly stack search utilizes memory sparingly, the tradeoff is a potentially longer verification time due to the regeneration of *forgotten* states as shown in Figure 9. In the example, the expansion starts with the state (I,I) and takes the path depicted as the left-most subtree of (I,I). Along the path, states such as (S,I), (D,I), and (S,S) are generated. When the path is exhausted, the expansion process backtracks and states such as (S,S) on the path are forgotten. As a result, forgotten states may be regenerated many times when different expansion paths are taken.

To avoid the regeneration of forgotten states, a technique called *state space caching* has been proposed [Godefroid et al. 1992; Holzmann 1985; Jard and Jeron 1991]. In this technique, some part of memory is managed as a cache memory to keep as many visited states as possible. A state removed from the current expansion path is saved in the caching memory space. When a new state is generated, it is checked with states in the caching space. If the state is found, expansion on the current path is stopped and the algorithm backtracks. Thus the method effectively uses all available memory and at the same time cuts the verification time taken by the stack search strategy.

Although state space caching can effectively reduce the number of regener-

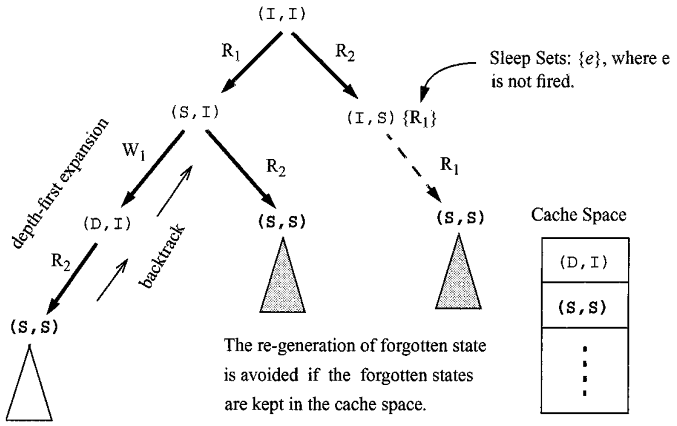

(I,I)
R1
R2
Sleep Sets: {e}, where e is not fired.
(S,I)
(I,S) {R1}
depth-first expansion
W1
R2
R1
(D,I)
(S,S)
(S,S)
Cache Space
R2
backtrack
(D,I)
(S,S)
(S,S)
The re-generation of forgotten state is avoided if the forgotten states are kept in the cache space.

Figure 9. Regeneration of forgotten states in stack search and sleep sets.

ations of forgotten states, inefficiency still exists because of redundant state comparisons due to unnecessary explorations of all permutable sequences of concurrent events that lead to the same state. As shown in Figure 9, from the starting state (I,I), the same state (S,S) is reached many times after applying transitions R_1 and R_2 in different orders. Transitions leading to the same state independent of the order in which they are applied are *independent* transitions [Godefroid et al. 1992]. In a shared-memory system, dependency can arise between accesses to shared-memory locations. For instance, two accesses to the shared-memory location are dependent if at least one of them is a write operation, whereas two concurrent read accesses are independent because they can be executed in any order without affecting the resulting state.

Based on the partial ordering of independent transitions, the "*sleep set*" was introduced in Godefroid [1990] and Godefroid et al. [1992] to explore the full state space without exploring all enabled transitions in each state. In this approach, every state is associated with a sleep set consisting of transitions that are firable but that will not be executed. For example, in Figure 9, when state (I,I) is under expansion, either the read access R_1 or the read access R_2 can be fired. Because the transition of R_1 (R_2) does not depend on the state of the cache $C_2(C_1)$ and the transition does not affect the resulting state of $C_2(C_1)$, R_1 and R_2 are independent transitions. Therefore, when the state (I,S) is generated by firing R_2 on (I,I), R_1 is kept in the sleep set of (I,S) to prevent the generation of state (S,S).

This technique has great potential for improving the efficiency of depth-first perturbation analysis because cache-based multiprocessor systems are full of independent concurrent events. For example, consider the protocol model given in Pong et al. [1994] for directory-based protocols [Censier and Feautrier 1978] (Figure 10). In this model, every processor is associated with one *message-sending* and one *message-receiving* channel, which simulate the interconnection. When processor P_i consumes message m_i emerging from its receiving channel, it may generate and send response messages to its sending channel without knowing or affecting states of other processors. In a broad sense, a processor's activity is totally isolated from the activities of other processors. Therefore transitions corresponding to propagation of messages in different channels and consumption of messages from different channels are independent. "Sleep sets" improve the verification time substantially by exploring only one interleaving of independent

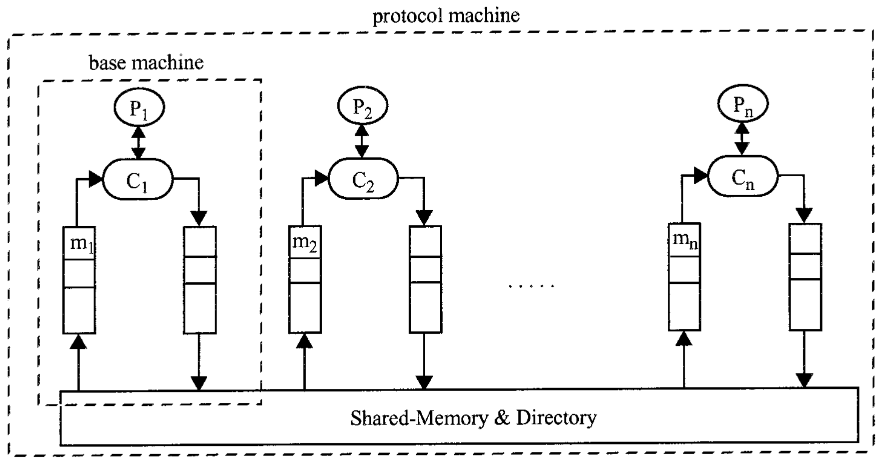


Figure 10. Protocol model for systems with directory-based cache coherence protocol.

transitions. But computing the "sleep set" adds extra complexity to the construction of the verification model. The verification tools also need to assist the protocol designer in identifying the independent concurrent events.

Finally, it is not clear how the livelock condition defined in Section 3.2 can be verified in a stack search algorithm because this on-the-fly technique only keeps track of states on the current expansion path.

5.4.2 *Supertrace Hashing.* Another possibility for improving the efficiency of state enumeration methods is to encode the state information to save memory, and to use hash tables to speed up the search and comparison operations. An extreme method along this line is Holzmann's [1990] *supertrace hashing*. The idea is based on a hash table T of binary values all initially set to 0. When a state s is generated, its hash value $h(s)$ is computed to access the corresponding bit in $T[h(s)]$. If $T[h(s)]$ is reset, the state is first-time generated and $T[h(s)]$ is set to 1. If $T[h(s)]$ is set, the state has been visited before.

The memory needed by this technique is limited to a stack to keep states for future expansion and a hash table of bits to detect visited states. Moreover, state matching operations are reduced to a calculation of hash values and a bit comparison. Unfortunately, this technique is not totally accurate because several states map to the same hash value and an ideal hash function mapping each global state to a unique hash value $h(s)$ is not practical. To reduce the probability of hash table collisions, Holzmann has advocated the use of two hash functions h_1 and h_2 such that a state s is present if both $T[h_1(s)]$ and $T[h_2(s)]$ are set. However, in this *multi-hash* scheme, T will fill up quickly and a very large hash table is required to ensure a reliable result (it is argued in Holzmann [1991] that the result remains reliable only if the table is less than 1% full.)

Recently, Wolper and Leroy [1993] revisited this problem and proposed an alternative called *hashcompact* to limit the probability of hash table collisions for even a single state to a negligible value. Holzmann's approach relies on a very large hash table T to provide a low probability of hash table collisions. Wolper and Leroy refined the method as follows. Assume that T has 2^k entries. (In practice k is large; e.g., $k = 64$.) When a state s is generated, its address in T is computed and represented by k bits. Then a much smaller standard hash table T' with a collision resolution scheme is used to store these k bits. The advantage of this method is that the probability of hash table collision can be negligibly small because a hash address is computed with respect to a very large table T, which is not physically allocated. The amount of memory required is of the order of entries actually stored in T'.

The hashcompact scheme is expected to be more efficient than the multihash scheme. Suppose that m bits are required to represent a state, where m (typically several hundreds of bits) is normally much larger than k (at most 64 bits). The hashcompact scheme computes and stores the k bits in T'. On the other hand, lk bits must be computed in the l-level multihash scheme. Moreover, at the same level of reliability, the hashcompact scheme has better storage efficiency. A detailed analysis of this hashcompact scheme is given in Wolper and Leroy [1993].

Although the hashcompact scheme is effective, the number of states that can be explored is fairly small. Given a memory size of 100 Mbytes and a tolerable collision probability of 10^{-3}, the number of states that can be explored is of the order of 10^7, which is fairly small for cache protocols even with a small model [Pong et al. 1994; Pong 1995]. More memory is still needed if the connectivity of the state graph must be maintained to verify the liveness property.

5.4.3 *State Pruning Based on System Symmetry.* Other techniques adopted in the symmetric Murφ [Ip and Dill 1993a,b] attack the state space explosion problem directly by constructing a tractable model that exploits system symmetries. In Figure 5, states (S,I) and (I,S) are identical to each other by symmetry, as are (D,I) and (I,D). In general, processors in a multiprocessor system are symmetric if the context of any two processors can be swapped without affecting the correctness of the system (Figure 10). To exploit this symmetry, the symmetric Murφ uses a function $\xi(s)$ mapping a state s into a unique state, which canonically represents the *orbit* [Clarke et al. 1993] of s, namely, the set of states that are symmetric permutations of s.

Ip and Dill have applied this method to verify the Stanford DASH protocol [Lenoski et al. 1990] and have shown that state enumerations with exploitation of system symmetries indeed reduce the size of the state space. Given a protocol model with n processors as in Figure 10, the maximum reduction is $n!$. Unfortunately, it was shown that even for fairly small models of 3 or 4 processors the reduced state space size is still above 10^7 states [Pong et al. 1994; Pong 1995] which indicates that the reduction by symmetry may not be sufficient to avoid the state space explosion problem.

5.4.4 *Summary.* State enumeration methods have been successfully applied to the verification of communication protocols for a long time [Bochmann and Sunshine 1980; Danthine 1980]. It is natural to adopt these methods to verify cache protocols. However, frequently communication protocols deal with only two agents. In contrast, in cache protocols multiple controllers participate in one transaction. Therefore many optimization techniques improving state enumeration methods for networking protocols cannot be applied [Yuang 1988]. For the verification of cache protocols, perhaps the most interesting optimization technique is to exploit the property of symmetry. However, it has been shown that the resulting reduction of the state space may not be sufficient to avoid the explosion problem.

A real advantage of state enumeration methods and tools such as Murφ is the simplicity with which protocol models are built and changed. As we mentioned in Section 3.1, most practically useful verifications are done at the system level. In the state enumeration method, specifying a protocol model is as simple as coding a program in an imperative language (Section 5.2). Therefore many useful data structures such as arrays and records, parameterized loop control, and procedures found in high-level programming languages can be easily supported in a model specification language such as Murφ. These constructs simplify the task of coding protocol models. Moreover, the protocol model is valid for any model size.

Powerful tools integrating the advantages of various optimization techniques are possible. For instance, stack search supported by caching memory implemented as the hash table in the supertrace or the hashcompact techniques could result in a very efficient, yet reliable method. Since the supertrace or the hashcompact techniques compress enormous state information into smaller hashing addresses stored in the hash table, a caching space with fixed size could contain more states as compared to storing the state information explicitly. Efficiency would be further improved if system symmetries were also exploited.

6. MODEL CHECKING

Model checking is a verification approach in which properties of the protocol are expressed as formulas in *temporal logic* and the verification is carried out formally. A temporal logic[3] is an extension of predicate logic with additional tense operators for expressing properties evolving with time. The specification is given in the form of a set *Spec* of logic formulas and the implementation is given as a semantic model *Imp*. The satisfaction relationship between the specification *Spec* and its implementation *Imp* is established by proving that *Imp* is a model for *Spec*. In general, the procedure starts with the construction of the state graph of the protocol model; then properties specified as temporal logic formulas are evaluated on the graph.

In particular, the branching time temporal[4] logic called *Computation Tree Logic* (*CTL*) can express mandatory properties of the protocols [Browne et al. 1986; Clarke et al. 1986]. An advantage of CTL is its broad expressiveness. It can handle arbitrary temporal formulas that represent safety and liveness properties. In general, it is more difficult to check liveness properties than safety properties because liveness properties are only meaningful for infinite behavior, whereas safety properties are checked by merely considering the finite behavior of a system. Representing the infinite behavior of a system is more difficult than the finite behavior. For example, to detect livelocks, a method based on state expansion must keep track of the history of state transitions and of all possible branches that may be taken during the state expansion. On the other hand, safety properties such as the consistency of multiple copies of the same memory block are easily checked by comparing the values of all data copies in the current state. By contrast, in model checking methods infinite system behavior is easily and efficiently expressed by *fixpoint* characterizations of CTL formulas [McMillan 1992].

In the following sections, we briefly review CTL and discuss the limitations of the model checking method. Next we survey the symbolic model checking approach proposed to overcome problems with model checking.

6.1 Model Checking and Computation Tree Logic

The model checking method is based on an infinite computation trace of the system model. As shown in Figure 11, the trace is composed of sequences of states derived from the *global* state machine of the system. A path is an infinite sequence of states (s_0, s_1, . . .) such that a transition exists from s_i to s_{i+1}, for all i. Properties evolving with computation

[3] Please see Browne et al. [1986], Clarke et al. [1986], and Pnueli [1977, 1981].

[4] A branching time temporal logic is a temporal logic that allows any state to have a finite number of immediate successor states.

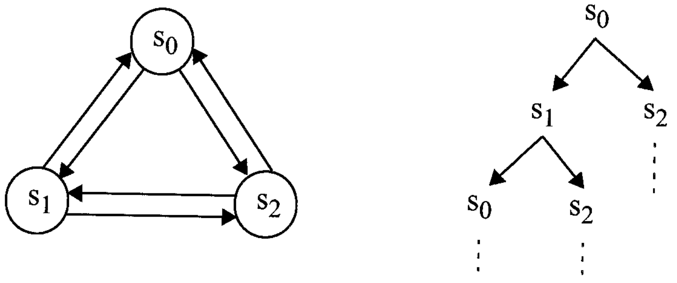


Figure 11. Global state machine and its computation trace.

on all or on selected execution paths, are specified in the Computation Tree Logic (CTL) [Browne 1986].

The CTL includes the usual logic connectives of negation ($\neg$) and conjunction ($\wedge$), with additional *tense* operators and *path quantifiers*. Given a set of atomic propositions AP, which are properties specified on the values of state variables, we have CTL formulas:

(1) Every $p \in AP$ is a CTL formula.

(2) If f_1 and f_2 are CTL formulas, so are $\neg f_1$, $f_1 \wedge f_2$, AXf_1, EXf_1, $A[f_1 U f_2]$, and $E[f_1 U f_2]$.

Tense operators X and U are the *next time* operator and the *until* operator, respectively; A and E are *for all* and *there exists* path quantifiers, respectively. Thus AXf_1 (or EXf_1) means that f_1 is true for all (or some, resp.) next immediate successor states of the present state. $A[f_1 U f_2]$ (or $E[f_1 U f_2]$) means that for every computation path (or for some computation path, resp.) there exists an initial prefix of the path such that f_2 is true in the last state of the prefix and f_1 is true in all other states along the prefix. The following additional operators are defined in Browne et al. [1986].

—$f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2)$, $f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$ and $f_1 \leftrightarrow f_2 \equiv (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$.

—$AF(f) \equiv A[True\ Uf]$ means that for every path, there exists a state in which f is true.

—$EF(f) \equiv E[True\ Uf]$ means that for some path, there exists a state in which f is true.

—$AG(f) \equiv \neg EF(\neg f)$ means that for every path, f is true in every state on the path.

—$EG(f) \equiv \neg AF(\neg f)$ means that for some path, f is true in every state on the path.

Given a formula f and a state s, the model checking method determines whether f is true in state s. The fact that a CTL formula f is true in state s is denoted by $s \mid= f$. As previously described, the basic model of the method is a global finite state machine $\mathcal{K} = (S, TR)$, where S is a nonempty finite set of global states, and $TR \subseteq S \times S$ is the set of transitions between global states. Formally, the truth or falsehood of a formula is computed as follows.

—$s_0 \mid= p$ iff $p \in AP$ is an atomic proposition and is true at s_0.

—$s_0 \mid= \neg f$ iff $s_0 \mid\neq f$.

—$s_0 \mid= f \wedge g$ iff $s_0 \mid= f \wedge s_0 \mid= g$.

—$s_0 \mid= AXf$ iff for all states s_1 such that $(s_0, s_1) \in TR$, $s_1 \mid= f$.

—$s_0 \mid= EXf$ iff for some state s_1 such that $(s_0, s_1) \in TR$, $s_1 \mid= f$.

—$s_0 \mid= A[f\ U\ g]$ iff for all paths $(s_0, s_1, \ldots)$, $\quad i\ [i \geq 0 \wedge s_i \mid= g \wedge \quad j\ [0 \leq j < i \rightarrow s_j \mid= f]]$.

—$s_0 \mid= E[f\ U\ g]$ iff for some path $(s_0, s_1, \ldots)$, $\quad i\ [i \geq 0 \wedge s_i \mid= g \wedge \quad j\ [0 \leq j < i \rightarrow s_j \mid= f]]$.

The method uses a *model checker* to evaluate the truth of a formula. To determine whether a formula f is satisfied in a given state, the model checker computes the *fixpoint*, that is, the set of states in which f is true. This is

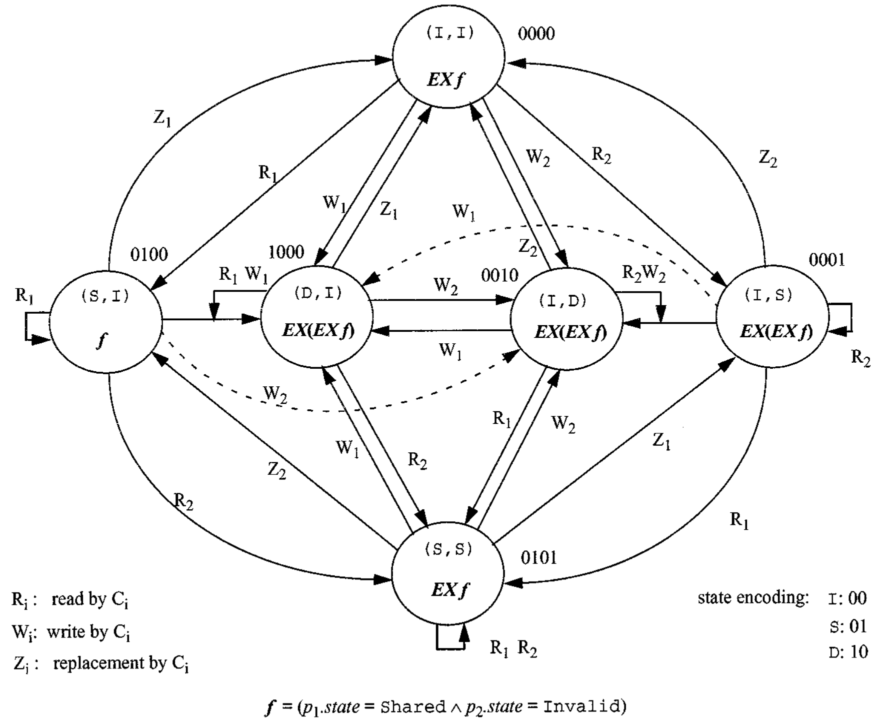


Figure 12. Fixpoint characterization of *EFf*.

achieved by a graph-traversal, iterative labeling process. Figure 12 shows the state graph of our simple protocol example.

By $EFf \equiv E[True\ U\ f]$, EFf is satisfied at state t if and only if $t \models f$ or there exists a path that leads from state t to state s and $s \models f$. Therefore, EFf can be characterized by a fixpoint $EFf \equiv f \vee EX(EFf)$ that labels all states having an access to state s in which f is true as an atomic proposition; that is, $s \models f$. This labeling process starts by first labeling all states t that are direct predecessors of s, which means $t \models EXf$. Then it labels all states w that are direct predecessors of t, that is, states w such that $w \models EX(EXf)$, and so on.

In our protocol example, the liveness property

$$EF(p_1.state = \text{Shared} \wedge p_2.state = \text{Invalid})$$

is true at state (S,I) as an atomic proposition. To prove it, the checker starts from an *empty set*, and labels state (S,I). Next states (I,I) and (S,S) are labeled because (S,I) is an immediate successor state of (I,I) and (S,S). States (D,I), (I,D), and (I,S) are labeled in the second iteration. At this point, the fixpoint is reached because no additional state can be labeled by iterating further.

Similarly, the set of states satisfying the safety property of data consistency AGf, where $f = (p_1.state = \text{Dirty} \rightarrow p_2.state = \text{Invalid})$ is computed by labeling all states initially. Then the fixpoint computation iteratively removes the label of some state x if there exists any successor state y of x such that f is not true in y. A detailed explanation of a fixpoint computation can be found in McMillan [1992].

As shown, the model checking method provides a very efficient way to reason about properties that evolve with time. Any property that can be specified as a formula can be verified. However, because the model is the global state graph which is obtained by an exhaustive exploration of the system state space, the state space explosion problem must be addressed as in state enumeration approaches [Graf et al. 1989], by representing the states and the state transitions symbolically.

6.2 Symbolic Model Checking

Symbolic model checking is a technique to perform model checking without explicitly representing the state graph [McMillan 1992]. Certain essential differences separate the model checking method from the symbolic model checking method. First, the global state graph in the symbolic method is implicitly represented by *Ordered Binary Decision Diagrams* (*OBDDs*) [Bryant 1986], which can be very compact so that the excessive memory requirement due to the state space explosion problem is avoided. Second, the symbolic method composes finite state *modules* [Nanda and Bhuyan 1992] to build the transition relations among global states. Reachable global states are not produced one by one as in state enumeration methods.

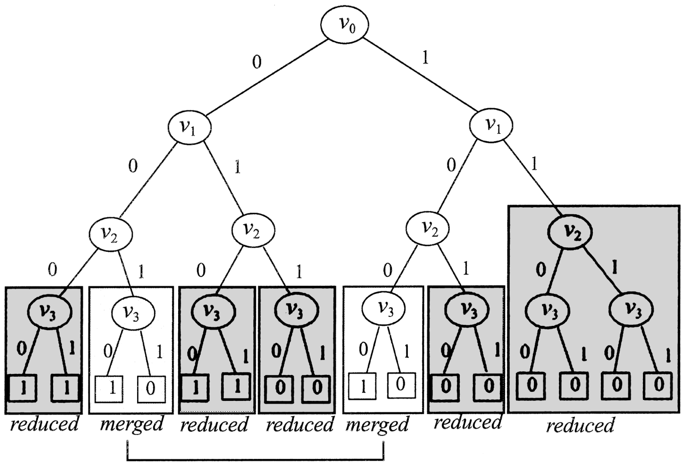
In the following sections, we first look into the structure of the OBDDs representing the global states and the transition relations. Then we show how the truth value of a formula is evaluated on the structure of the symbolic state graph, and how to build transition relations for a finite state system.

6.2.1 *OBDD Representation of the State Space and Transition Relations.* In the symbolic model checking method, a vector V of *Boolean* variables represents the states of components in the system. A system state is an assignment of either 0 or 1 to each variable in V. Consequently, the set of all system states is obtained by all possible interpretations to variables in V. A Boolean function $f_s(V)$ can then represent the global state space if $f_s(V)$ is true for all reachable states and $f_s(V)$ is false for all unreachable states. More generally, a Boolean function $g(V)$ is a characteristic function that represents the set of global states such that $g(V)$ is true. Based on the same idea, all state transitions in the state diagram are also represented by a Boolean function $TR(V, V')$ such that $TR(V, V')$ is true for all V and V' such that there is a state transition from V to V'. In the following, we use a primed variable for every next-state variable. Thus V and V' represent the current- and the next-state, respectively.

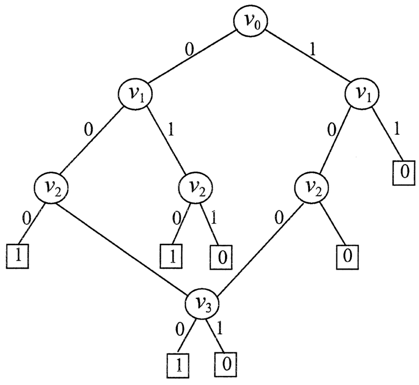
In our protocol example, the set of all reachable global states is {(I,I), (I,S), (S,I), (S,S), (I,D), (D,I)} and the state graph is shown in Figure 12. First, we need to encode the cache states. Because a cache can be in three states, two bits are required to encode a state. Arbitrarily, we encode state I as 00, state S as 01, and state D as 10. Thus every global state is represented by a vector $V(v_0, v_1, v_2, v_3)$ of 4 binary bits, and the set of global states is $\{(\mathrm{I,I})_{0000}, (\mathrm{I,S})_{0001}, (\mathrm{S,I})_{0100}, (\mathrm{S,S})_{0101}, (\mathrm{I,D})_{0010}, (\mathrm{D,I})_{1000}\}$. We can easily represent this set by a Boolean function $f_s(V)$ so that $f_s(V)$ returns true values for states in the system state space. In this example,

$$\begin{aligned} f_s(v_0, v_1, v_2, v_3) &= \bar{v}_0\bar{v}_1\bar{v}_2\bar{v}_3 + \bar{v}_0\bar{v}_1\bar{v}_2 v_3 \\ &\quad + \bar{v}_0 v_1\bar{v}_2\bar{v}_3 + \bar{v}_0 v_1\bar{v}_2 v_3 \\ &\quad + \bar{v}_0\bar{v}_1 v_2\bar{v}_3 + v_0\bar{v}_1\bar{v}_2\bar{v}_3. \end{aligned}$$

For the purpose of efficient manipulation, OBDDs [Bryant 1986] represent Boolean functions internally. Figure 13 shows the OBDD representing the state space of our example. Given an ordering of variables (v_0, v_1, v_2, v_3), the OBDD representing f_s is shown in Figure 13(a). Each nonterminal node containing a decision variable is circled and each terminal node containing a function value 0



(a) f_s and its OBDD.



(b) Reduced OBDD.

Figure 13. OBDD for set of reachable state: (a) f_s and its OBDD; (b) reduced OBDD.

or 1 is represented by a square. For instance, when $v_0 = v_1 = v_2 = v_3 = 1$, $f_s(V) = 1$. Therefore the path in the tree for ($v_0 = v_1 = v_2 = v_3 = 1$) leads to a terminal node with function value 1.

Two reduction rules are repeatedly applied to the OBDD tree in order to reduce its size. First, a vertex whose two branches point to the same vertex are deleted. For example, $f_s(V)$ returns 1 for ($v_0 = v_1 = v_2 = v_3 = 0$) and ($v_0 = v_1 = v_2 = 0$, $v_3 = 1$). Thus the value of $f_s(V)$ is independent of v_3 for these two paths and node v_3 can be deleted. Second, two isomorphic subgraphs are merged. The resulting graph is called the reduced OBDD. Checking whether a state is reachable is as simple as traversing the reduced OBDD tree.

6.2.2 *Evaluation of Formulas.* Given a finite state system, the symbolic model checking method starts with the construction of the OBDD for the transition relations $TR(V, V')$. To evaluate the truth or falsehood of a formula f in a state s, the symbolic model checker performs operations similar to the model checking method in Section 6.1. It computes the set of states (as characterized by a Boolean function and represented by an OBDD) where f is true by a fixpoint computation. It then checks whether the given state s belongs to the generated set of states. For example, let us assume that we want to find out whether state s has a successor state in which formula f is true, that is, whether $s \mid= EXf$. Informally, the method starts by finding the set of states in which f is true. This set is represented by the characteristic function $f(V')$. Then we determine the set of states $g(V)$ that have a successor state in $f(V')$. Finally, the method checks whether $s \in g(V)$. Formally, we have [McMillan 1992]:

$$s \mid= EXf$$

$$\text{iff} \quad s \in (\quad V'(f(V') \wedge TR(V, V')).$$

When the conjunction (AND) operation is executed on $f(V')$ and $TR(V, V')$, the resulting OBDD $TR'(V, V')$ characterizes the transitions from states in $g(V)$ to states in $f(V')$. Subsequently, the method executes the existential quantification on the next-state variables in V'.[5] Since the existential quantifier on $v' \in V'$ means an arbitrary choice of decision paths (i.e., independent of the value of v'), it is an OR operation on the left and right subtrees of v'. After the quantification operation, all variables v' in V' have disappeared from $TR'(V, V')$. The resulting OBDD for $g(V)$ represents the set of states that can reach states in $f(V')$ in one step. Whether the state s is in $g(V)$ can be checked by traversing the OBDD tree of $g(V)$.

Similarly, to compute the set of all reachable states, we can start with a characteristic function $Init(V)$ for the set of initial states. The set of immediately reachable states $r(V')$ from any initial state can be computed as:

$$V \quad Init(V) \wedge TR(V, V').$$

[5] Existential quantification and conjunction operations can be executed simultaneously during the computation. It depends on the implementation.

The conjunction and existential quantification operations are performed as previously explained. The resulting OBDD represents the set of next-state $r(V')$ reachable from states in $Init(V)$ in one step. To compute the set of states reachable from $Init(V)$ in two steps, we use $r(V' \leftarrow V)$ to substitute $Init(V)$ in the preceding equation, where $V' \leftarrow V$ means that the next state variables are replaced by the current state variables. The computation is repeated iteratively until the fixpoint is reached, and the resulting OBDD does not vary.

6.2.3 *Complexity of Model Specification and Construction of Transition Relations.* As shown in the previous section, the transition relations TR are essential for the model checking method. The OBDD for TR must be built efficiently. Establishing the transition relations among global states one by one as in state enumeration methods is not acceptable. Typically the protocol model is built on asynchronously communicating finite state *modules*, a model similar to the one taken by Nanda and Bhuyan [1992] and previously explained in Section 5.2. In the CSP language adopted by Nanda and Bhuyan [Hoare 1978], a model is specified in terms of communicating processes and a process that executes an input (output) primitive is blocked until the process with which it tries to communicate executes a corresponding output (input) primitive. By contrast, the symbolic model checking method was developed in the context of digital circuits and the semantic of its specifications is similar to the *Verilog* hardware description language [Tomas and Moorby 1995]. Modules do not block implicitly as CSP processes do whenever a communication cannot be established. Every module executes its logic cycle by cycle. (A cycle can be thought of as a state transition step.) In each cycle, the value of every state variable must be precisely set, which results in very complex models.

For our simple protocol example, the

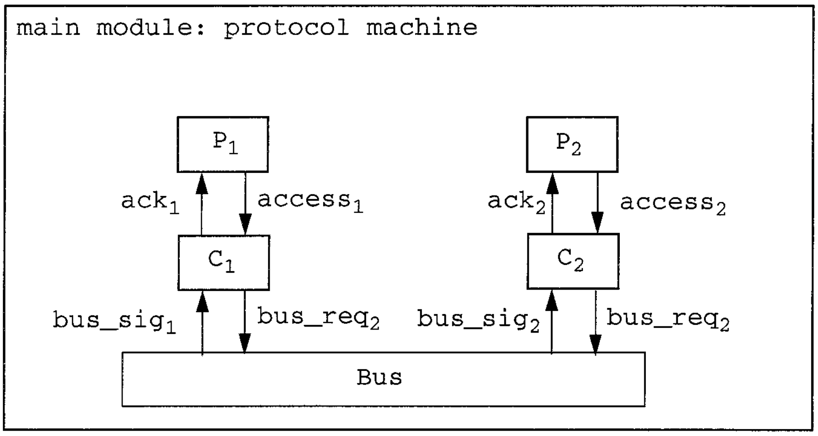


Figure 14. Structural description of bus-connected system with two caches.

protocol model consists of processor, cache, and bus modules (the bus is needed as in Nanda and Bhyuan's model). Figure 14 shows a structural description of the protocol model in which the data paths are not included for simplicity. We first need to define the behavior of each module. The behavior of each processor module is given by the pseudocode of Figure 15. The module has one output port *Req* which indicates the type of memory access and one input port *Ack* which signals the completion of a prior access. The module also maintains two internal variables *ProcSt* and *PendingAcc* which remember the status of the processor and the type of any pending access, respectively. *ProcSt* is initially set to *ready* and *PendingAcc* to *nop*. In the next cycle, *ProcSt* moves onto the *issuing* state, and then to the *waiting* state after issuing the memory access (a load or a store). When *Ack* is activated, the next state of *ProcSt* is *ready*. For all other cases, *ProcSt* remains in its current state. Syntactically, the next state value of *ProcSt* is:

$$
\begin{aligned}
&ProcSt' \\
&= [(ProcSt = ready) \wedge (issuing)] \\
&\quad \vee [(ProcSt = issuing) \wedge (waiting)] \\
&\quad \vee [(Ack) \wedge (ProcSt = waiting) \wedge (ready)] \\
&\quad \vee (\neg([(ProcSt = ready)] \\
&\qquad \vee [(ProcSt = issuing)] \\
&\qquad \vee [(Ack) \wedge (ProcSt = waiting)]) \\
&\qquad \wedge (ProcSt)).
\end{aligned}
$$

To understand the preceding equation, consider a simpler expression: $next(x) = if(c)\ x\ else\ \neg\ x$, where x *and* c are Boolean variables. We know that $x' = (c \wedge x) \vee (\neg\ c \wedge \neg\ x)$ because when $c = 1$, $x' = x$ and $c = 0$, $x' = x$. As a result, we can represent the transition relation between x and x' by an OBDD over c, x, and x'. In the same way, the transition relations for state variables *ProcSt*, *PendingAcc*, and *Req* in the processor module of Figure 15 are characterized by three OBDDs over the state variables. Since the processor executes its logic and updates the values of its state variables in each cycle, a conjunction operation on the three OBDDs yields the OBDD TR_P, which characterizes the state transitions for all variables defined in the processor module. Following the same idea, we can build TR_C and TR_B for the cache and the bus modules, respectively.

We now describe the main module for the protocol model. A brief segment of code for the main module is shown in Figure 15. *Instances* of processor, cache,

```
module Processor (Req, Ack);
  output Req; // output port
  input Ack; // input port

  ProcSt: {ready, issuing, waiting};
  PendingAcc: {nop, load, store};

  init(ProcSt) = ready;        // initial state
  init(PendingAcc) = nop;

next(ProcSt) = switch
        case (ProcSt = ready) : issuing;
        case (ProcSt = issuing) : waiting; // move one the waiting state
        case (Ack) & (ProcSt = waiting) : ready; // move back to the ready state
        else : ProcSt; // remain at the same state;
    endswitch;

next(PendingAcc) = switch
        case (ProcSt = waiting): nop; // no operation is issued while waiting
        case (ProcSt = issuing): {load, store}: // non-deterministic
        else: PendingAcc;
    endswitch;
Req = PendingAcc;
endmodule;
      :::
      :::

module Cache (Relase, MemAcc);
  output Release;
  input MemAcc;

      :::
      :::

module Main;
  access1: .....
  ack1: ......

Processor  P1  (access1, ack1); // instantiate P1
Cache      C1 (ack1, access1); // instantiate C1

      :::
```

Figure 15. Processor, cache modules, and module instantiations in main module.

and bus modules are first created. *Wire variables* connect modules as shown in Figure 14. Each wire variable connected to the port of a submodule is substituted in the code of the submodule when the submodule is instantiated so that the OBDD is built with the correct state variables. The transition relations among state variables are built for each individual instantiated module, leading to TR_{P1}, TR_{P2}, TR_{C1}, TR_{C2}, and TR_B for the two processors, the two caches, and the bus module in Figure 14. Depending on whether the execution mode is *synchronous* or *asynchronous*, the global state transition relations are formed by either taking a conjunction operation or a disjunction operation on the five individual relations; that is,

$$TR_{sync} = TR_{P1} \wedge TR_{P2} \wedge TR_{C1} \wedge TR_{C2} \wedge TR_B \text{ or}$$

$$TR_{async} = TR_{P1} \vee TR_{P2} \vee TR_{C1} \vee TR_{C2} \vee TR_B.$$

In the synchronous mode, all modules advance in a lock-step manner, whereas, in the asynchronous mode, only one module executes at a time as in state enumeration methods. Which mode is better in terms of verification time and memory consumption is unclear [McMillan 1992].

Our example shows that the specification of a cache protocol model in the symbolic model checking method is fairly complex, even though we have excluded the data paths. In our opinion, this method is not meant for protocol verification, but for the verification of digital circuits. Essentially, the system under verification must be modeled by a set of Boolean formulas. This is a natural representation for a digital circuit [Tomas and Moorby 1995] because each wire is a Boolean variable. However, it is cumbersome for cache protocol models, in which each state must first be binary encoded.

A serious drawback of the specification is that many irrelevant details must be included in the protocol model. As mentioned in Section 3.1, most useful formal verifications are done at the message-passing level during the early protocol design phase. In that stage, the implementations of cache controllers, the arbitration schemes to access the cache, and the handshaking protocol to interface the cache modules to the interconnect are neither useful nor defined. These aspects of the design can be easily abstracted in the state enumeration methods as was shown in Section 5.2, but not in the symbolic model checking method. In most cases, these extra signals to establish communication among modules are not even part of the actual implementation of the protocol. Nevertheless, they are required in the verification in order to connect the submodules logically.[6] Thus building a model centric to the behavior of the cache protocol is very difficult.

Another serious drawback of the model checking method is that state variables cannot be manipulated globally. In Section 5.3, we showed the error caused by a stale write-back [Pong et al. 1995]. To detect the error, the verification method must keep track of the value written by the latest store. A state enumeration method does this efficiently by remembering the most recent data copy as a *Fresh* copy while all other copies turn *Obsolete* at the point in time when the store is performed. To adopt the same model in the symbolic model checking method, every cache must *export* the value of its cached copy to all other caches. A cache module must also be able to modify the state variables of other caches. Unfortunately, this is very difficult to do in the symbolic model checking method.

6.2.4 *Complexity of the OBDD Representation.* Although the symbolic model checking method does not enumerate all reachable states explicitly, the size of the OBDDs representing the set of reachable states and the transition relations may increase rapidly in proportion to the scale and the complexity of the system. OBDD representations are efficient only in a heuristic sense and the worst case may still be catastrophic. For large problems, it is often the case that the OBDD representing the transition relations is too large to compute. Early studies have shown that the size of an OBDD is very sensitive to the ordering of decision variables. Several researchers have looked for the best choice of variable ordering to optimize the size of an OBDD.[7] However, recently, Liaw and Lin [1992] have shown that almost all Boolean functions (of n variables) require at least $2^n/2n$ nodes even for the optimal ordering of variables. Wegener [1994] subsequently proved that almost all Boolean functions have a *sensitivity* of almost 1. In other words, the minimal OBDD size for the optimal variable or-

[6] This problem also exists in Nanda and Bhuyan's [1992] approach of composing modules (Section 5.2).

[7] Please see Bryant [1992], Ender et al. [1991], Friedman and Supowit [1990], Fujita et al. [1988], and Malik et al. [1988].

dering differs from the minimal OBDD size for the worst variable ordering by a factor of almost $1 + \varepsilon(n)$, where $\varepsilon(n)$ converges exponentially fast to 0. These worst-case theoretic results may explain the disappointing performance of the method in some cases [Hu and Dill 1993a,b].

Several approaches to avoid the explosion of OBDD sizes have been proposed. Hu and Dill [1993b] reduce the OBDD size by eliminating *functionally dependent variables*, that is, variables that are always functions of other independent variables. Chiodo et al. [Chiodo et al. 1992; Shiple et al. 1992] have introduced *compositional model checking*. Their idea, applicable to synchronous systems, is to reduce the component machines with respect to the property to verify. It is hoped that the OBDD formed by the product of reduced component machines will be smaller. Other approaches [Burch et al. 1991; Hu et al. 1992] avoid building a giant OBDD for all next-state relations and use disjunctive partitioned OBDDs. This idea is illustrated by the transition relation for asynchronously interleaved transitions in Section 6.2.3; that is, $TR_{async} = TR_{P1} \vee TR_{P2} \vee TR_{C1} \vee TR_{C2} \vee TR_B$. Thus the total number of nodes is the sum of the numbers of nodes of all component OBDDs rather than their product. Further improvement can be achieved by composing even smaller OBDDs, each corresponding to individual state assignment [Hu et al. 1992]. The tradeoff is a longer verification time because one computation on a full next-state relation translates into a large number of computations, one for each component relation.

6.3 Symmetric Model Checking

As mentioned in Section 5.4.3, a complex system often exhibits a great deal of *regularity* and *symmetry*. Clarke et al. [1993] and Emerson and Sistla [1993] extended the model checking technique by exploiting symmetry. By applying (symmetric) permutation operators G on a state s, we can obtain the *orbit* set of states $\theta(s)$ of s, which can be canonically represented by one state (denoted by $\xi(s)$ in Ip and Dill [1993b]) picked from $\theta(s)$. Transition relations $TR(s_1, s_2)$ are then converted into $TR_G(\theta(s_1), \theta(s_2))$. Since states that are permutations of each other are lumped into a single canonical state, the state space and the OBDD size after transformation can be significantly reduced.

6.4 Summary

In this section, we have surveyed various aspects and variants of the model checking technique. The strength of this technique stems from the expressiveness of temporal logic, which can handle arbitrary temporal formulas, representing both safety and liveness properties. Unfortunately, since the method takes the state graph as a model, it is also plagued by the state space explosion problem.

In order to avoid the state space explosion problem, a symbolic model for the state graph is desirable. In the symbolic model checking method, a vector V of Boolean variables represents the states of all components in the system. A system state is an assignment of 0 or 1 to each variable in V, and the set of all system states derives from all possible interpretations to the variables in V. The state space is specified by a Boolean function $f(V)$, and represented by an Ordered Binary Decision Diagram or OBDD. Similarly, the next-state transition relation is represented by an OBDD, $TR(V, V')$, in which V and V' represent the current- and next-state variables.

Symbolic model checking has an additional advantage: the set of states satisfying a formula in temporal logic is found considerably faster than in a state enumeration method, because the state search operation is performed on a *set* rather than on individual states. For example, when $Init(V)$ represents the set of initial states, the operation (V $Init(V) \wedge TR(V, V')$) yields the Boolean formula for every state V' reachable from any initial state in one step.

The serious drawback of symbolic model checking is the high complexity of building a protocol model. Often unnecessary details must be included in the protocol model in order to synchronize the behavior of connected modules. Overall, the specification method is more suitable for digital circuits than for protocols. In spite of the potential performance of the method, the programming complexity to build a model is high.

Although an OBDD can be simplified by exploiting regularities in the binary decision tree, the OBDD for next-state relations is often too large to compute, which is known as the *OBDD-size explosion problem*. Several approaches to reduce the OBDD size have been proposed and implemented. Applications of the symbolic model checking method have been limited in the literature to the verification of the snooping protocols of the Gigamax [McMillan and Schwalbe 1991] and of the Futurebus+ [Clarke et al. 1993b]. Snooping protocols are relatively simple, and, currently, there is no evidence that the method is efficient enough to verify more complex protocols such as directory-based protocols.

7. STATE ENUMERATION BASED ON SYMBOLIC STATE MODEL

An approach to avoid the state space explosion problem is to exploit equivalent relations among global states so that a set of equivalent states is represented by one canonical state. In a simple perturbation method, two states are considered equivalent only if they are identical or if they are symmetrically identical in methods with symmetry extension. However, even a symmetric tool may not be very effective to avoid the state space explosion problem [Pong et al. 1994; Pong 1995].

In this section, we describe a technique that searches the state space exhaustively as in traditional state enumeration methods, but the system is represented by a *symbolic state model* (SSM). The *abstraction* in this model is much more powerful than the symmetric relations obtained from symmetry alone and an abstract state represents a large set of states so that the reduction of the state space size is more substantial. In the following, definitions have been simplified for illustration purpose. Detailed justifications and explanations can be found in Pong and Dubois [1995] and Pong [1995].

7.1 Abstraction and State Representation

The SSM method differs from other state enumeration approaches in the ways of representing a global state. The state abstraction is motivated by the following observations.

Observation 1. Cache protocol agents are *homogeneous*. In all existing protocols and in the protocol example of Figure 3, the behavior of every cache is characterized by the same finite state machine (as shown clearly in Figure 3, the two FSMs for C_1 and C_2 are exactly the same except that different processors have different labels); and

Observation 2. In all existing protocols data coherence is enforced by either broadcasting writes to all copies so that they remain coherent, or by invalidating the copies in all other caches so that modifications are done in one and only one cache at a time.

To illustrate the first observation, consider a snooping protocol for shared-bus systems. When an invalidation is sent on the bus, all caches in the Shared state invalidate their copies. Thus all the cache FSMs in the Shared state make exactly the same move from the Shared state to the Invalid state. Because of this homogeneity, all FSMs in the same state can be logically grouped into a set lumped together as a single integrated FSM. Moreover, by the second observation, the exact number of data copies in a clean shared state is irrelevant to protocol correctness. What is critical is whether there exists 0, 1, or multiple copies in a particular state (such as more than one cache in the Dirty state). Therefore we can map global states to more abstract states that do not keep track of the exact number of caches in a particular state. The following notations are used to represent these abstract states.

Definition 2. *(Repetition Constructors)*

(1) Null (0) indicates zero instance.
(2) Singleton (1) indicates one and only one instance. (This constructor can be omitted.)
(3) Plus (+) indicates one or multiple instances.
(4) Star (*) indicates zero, one, or multiple instances.

With these repetition constructors one can build concise representations of complex states. For example, we can represent the set of global states such that "one or multiple caches are in the Invalid state, and zero, one, or multiple caches are in the Shared state" as (I^+, S^*). For protocols such that the cache states must be compatible with the values of data copies, this state representation carries, to some extent, the information necessary to verify data consistency. For instance, global states such as (D^*, . . .) and (D,S^*) signal that erroneous states are reachable. An advantage of this representation is that protocols can be verified independently of the size of the verification model. Thus the verification is more reliable than with other approaches that can only deal with small protocol models.

In a system with an unspecified number of caches, the method groups cache machines in the same state into *state classes* and specifies their number in each class by one of the repetition constructors.

Definition 3. *(Composite State)*. A composite state represents the state of the protocol machine for a system with an arbitrary number of cache entities. It

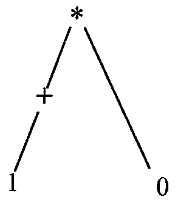


Figure 16. Ordering relations among repetition constructors.

is constructed over state classes of the form ($q_1^{r_1}$, $q_2^{r_2}$, . . . , $q_n^{r_n}$), where n is the number of states of a cache machine, and q_i is a defined cache state, $r_i \in [0, 1, +, *]$.

Repetition constructors are ordered according to the possible states they specify (Figure 16). The resulting order is $1 < + < *$; the null instance can be ordered with respect to *; that is, $0 < *$. This order leads to the definition of *state containment*.

Definition 4. *(Containment)*. A composite state S_2 contains composite state S_1, or $S_1 \subseteq S_2$, if

$$q^{r_1} \in S_1 \qquad q^{r_2} \in S_2 \quad \text{such that}$$

$$q^{r_1} \leq q^{r_2}, \quad \text{that is,} \quad r_1 \leq r_2,$$

where q is the cache state and r_1, $r_2 \in [0, 1, +, *]$.

The consequence of containment is that, if $S_1 \subseteq S_2$, then the family of states represented by S_2 is a superset of the family of states represented by S_1. Therefore S_1 can be discarded during the verification process provided we keep S_2 because every state obtained by expanding S_1 can be obtained by expanding S_2; that is, if $S_1 \subseteq S_2$, then $\tau(S_1) \subseteq \tau(S_2)$ where operator τ is a memory event applied to the current state.

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

As the expansion process progresses, new composite states are created. A new

```
Algorithm: Essential States Generation.

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

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

Figure 17. Algorithm for generating essential states.

state is discarded if it is contained in a visited state and all visited states contained in a newly expanded state are discarded (see Figure 17). At the end of the expansion process the state space is partitioned into several families of states (which may be overlapping) represented by *essential* states (Figure 18).

Definition 5. *(Essential State)*. Composite state S is essential if and only if there does not exist a reachable composite state $\bar{S}$ such that $S \subseteq \bar{S}$.

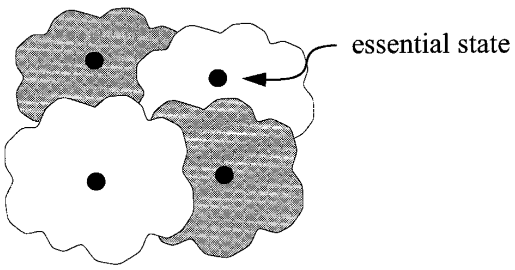


Figure 18. Representation of state space by essential states.

7.2 Generation of the State Graph and Error Detection

In the SSM, the generation of the state graph is the same as for the state enumeration methods. Enabled transition rules are first applied to the current state to derive new state information. This step is then followed by an *aggregation* step that groups caches in the same state into classes. Finally, state containment is checked. Figure 19 shows the expansion steps taken to generate the set of essential states for the protocol example in the case of an unbounded system.

The expansion starts in the initial state with no cached copy. At the end of the expansion, only three essential states (S^*,I^+), (S^+,I^*), and (D,I^*) are produced. It is critical to see that the state graph produced by the SSM is independent of the size of the verification model, as opposed to a traditional state

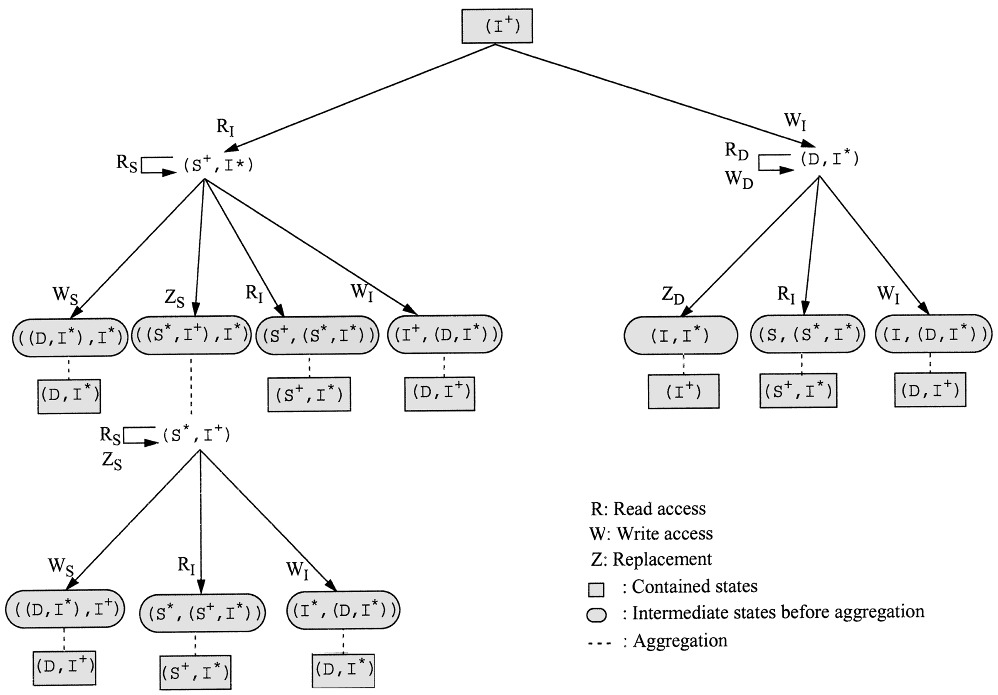


Figure 19. Generation of essential states in protocol example.

enumeration method (Figure 5), in which the number of global states produced is proportional to the number of components in the verification model. This is a very appealing feature because the question of how many processors should be included in the model in order to achieve a 100% error coverage is eliminated. The SSM yields reliable results because all possible sequences of events for models of arbitrary sizes are covered. Since the number of essential states is generally small, the traversal of the state graph is likely to be less expensive than in traditional methods. The memory needed to keep the state information is also likely to be smaller.

Since the SSM method also enumerates the state space, the mechanisms for detecting protocol errors are the same as for traditional state enumeration methods.

7.3 Summary

The SSM method employs a specialized abstraction that groups caches in the same state into a class and the number of caches in the class is symbolically represented by a repetition constructor. This symbolic representation is justified by the symmetry and homogeneity of cache-based systems. In addition, the key idea behind this abstraction is to exploit a unique property of cache coherence protocols: the fact that protocol correctness is not dependent on the exact number of cached copies. Rather, symbolic states only keep track of whether the caches have 0, 1, or multiple copies. Because an essential state represents a very large set of states that would otherwise be enumerated explicitly, the reduction of the state space is substantial. Moreover, the verification result is independent of the model size, and consequently is reliable.

The major drawback of the symbolic state method is that a more abstract protocol model is required. For instance, the exact topology of the interconnection cannot be modeled. Instead, abstract message channels with FIFO or nonFIFO scheduling are used to establish a point-to-point communication path between two components (Figure 10). This may not be a serious problem because we are mostly concerned about the order of message propagations, which can be essentially modeled by the appropriate type of message channels. Another problem of this approach is that it may not be efficient for verifying linked-list protocols such as the Scalable Coherent Interface (SCI) [James et al. 1990] or the protocol of S3.mp [Nowatzyk et al. 1994; Pong et al. 1995]. The problem is the lack of a good abstraction to model the extreme case of an infinitely long linked list, given a system with an arbitrary number of processors. Nevertheless, this problem is shared with all other methods.

8. EVALUATION AND COMPARISON

In this section, we compare the three general classes of verification methods based on state enumeration, symbolic model checking, and symbolic state model. The tools implemented for each class are Murφ [Dill et al. 1992], SMV [McMillan 1992] and Ever [Hu et al. 1992], and SSM [Pong 1995], respectively. The properties compared are their *performance*, their *generality* and *applicability*, as well as their *automaticity* and *amenity*. The performance comparison is made in terms of the computation time and of the amount of memory required to verify a benchmark directory-based protocol.

8.1 Performance

The benchmark protocol is a sequentially consistent, directory-based protocol implemented in RPM, a hardware multiprocessor testbed developed at the University of Southern California [Barroso et al. 1995]. The architectural model is given in Figure 10. Only one memory block of one word is modeled and the interconnection is modeled by two FIFO message channels. The protocol supports basic read and write accesses and two types of hardware-based prefetching operations (for shared or exclusive copies). The processor is blocked on a read or write miss but not on prefetch misses. Replacements are modeled as processor accesses that can take place at any time. The values of data copies are not explicitly represented. Instead, a cached block may be in one of three states: Nodata (the cache has no valid copy), Fresh (the cache has an up to date copy), and Obsolete (the cache has an out of date copy); the memory copy is either Fresh or Obsolete. During the course of verification, the expansion process keeps track of the status of all block copies in conformance with the protocol semantics as we explained in Section 5.3.

The model for state enumeration is shown in Figure 20. Each processor structure maintains the state of the processor, the type of pending memory accesses, the messages enqueued in the two message channels, the local cache state, and the current data value of the cached copy. The memory structure records the value and the state of the memory copy. For the SSM method, the processor context (the base machine of Figure 10) is the basic repetitive component of the abstraction.

Table II compares the performance of the SSM method and the Murφ system[8] (both nonsymmetric Murφ-ns, and symmetric Murφ-s) to the verification of the benchmark protocol at the message-passing level. The three tools were run on a SPARCstation 10 Model 30 with 128 Mbytes of memory and statistics were collected on memory usage and verification time.

For small-scale systems with less than four processors, the time complexity and the memory requirement of Murφ-s and Murφ-ns are tolerable. The sizes of both the global state space and of the search state space of Murφ-s are significantly less than those of Murφ-ns.

[8] The Murφ system we used is a preliminary version 1.52s provided by Ip and Dill. This version has not been publicly released.

```
Var
    P:  Array [0..NumProcessor] of
     Record
      ProcSt: {blocked, un-blocked}; -- processor state
      PendingAccess: {NoPending, PendingRead, PendingWrite};-- pending access
    Rec_CH: Array [0..ChCapacity] of-- receiving channel
      Record
      MsgType: {miss-reply, miss-reply-own,...};-- memory-to-cache messages
      val: {nodata, fresh, obsolete};-- value of carried data
    End;
    Send_CH: Array [0..ChCapacity] of-- sending channel
      Record
      MsgType: {rmiss-req, wmiss-req,...};-- cache-to-memory messages
      val: {nodata, fresh, obsolete};-- value of carried data
      End;
      CacheSt: {INV, RO, RW, .....};-- local cache state
      CacheData: {nodata, fresh, obsolete};-- value of the cached block
     End;
    M: Record
      MemSt: {UNCACHED, SHARED, DIRTY, .....};-- memory state
      MemData: {nodata, fresh, obsolete};-- value of the memory block
      PBit: Array [0..NumProcessor] of {0, 1};-- presence bit vector
      DirtyBit: {0, 1};-- dirty bit
      ReqId: 0..NumProcessor;-- processor id of the pending request
     End;
```

Figure 20. State information maintained by each component in the model.

However, when more processors are added, this advantage dwindles and both the verification time and the memory usage increase drastically. For the case of four processors, the expansion process in Murφ-s does not converge at this time. There are still approximately 860,000 states to explore. Comparatively, the SSM method is more efficient. The method takes only 15,840 seconds to generate the essential states. To build the connectivity information between essential states and check the simple deadlock and livelock conditions, we simply rerun the process with the set of essential states as the set of starting states, which takes about the same amount of time as does generating the set of essential states. Memory requirement is about 18 Mbytes (state representations are not encoded).

The set of essential states reported at the end of the SSM run contains all possible distinguishable states (from the point of view of verifying coherence) in which the system could ever be. The most complex essential states consisted of 97 base machines in different states. Therefore, in a state enumeration method, a system model with at least 97 processors is required to obtain a 100% error coverage for this protocol. The SSM method, which provides reliable results by verifying the protocol for any system size, is therefore needed to obtain total error coverage. Because the number of global states reported is relatively very small (16,933 states in this case), checking the connectivity of the global state transition diagram for livelock detection is feasible with SSM. For more performance comparison results, interested readers could refer to Pong [1995].

Unfortunately, we do not have information regarding the SMV and the Ever systems, and we are unable to evaluate the symbolic model checking methods. However, it was reported in Hu et al. [1992] that the OBDDs (in the form of a fully evaluated next-state relation or disjunctive partitioned relations) required for verifying a directory-based protocol cannot be built with 60 Mbytes

Table II. Performance Comparison Between Murφ and SSM

Method	Number of processors	Size of global state space	Size of search state space	Verification time (seconds)	Memory usage (Mbytes)
Murφ-ns	2	7,132	34,870	37.2	0.2
	3	1,001,243	7,446,069	7,574.2	34.5
	4	excessive memory usage			
Murφ-s	2	3,819	18,740	26.1	0.1
	3	246,865	1,843,763	3,531.6	8.5
	4[a]	3,565,676	424,315,444	246,905	160
SSM	any n > 1	16,933	10,430,301	15,840	18

a This run is not fully completed because the expansion process does not converge. Also, the result is obtained from a Solbourne Series5E/900 system due to the memory requirement.

of memory. We think there is a reasonable explanation for this. Directory-based protocols are far more complex than snooping protocols. There are more cache states and message types in a directory-based protocol than in a snooping protocol. In Clarke et al. [1993b] the results of applying the SMV to the verification of the Futurebus+ protocols are given. For the most complex configuration of 3 buses and 8 processors, the model requires up to 250 Boolean state variables. However, for the protocol emulated on RPM, the representation of a global state in the most abstracted model with 4 processors takes up to 300 bits. When the same protocol is coded in the SMV, we should expect that many more bits will be needed since signals connecting modules must be added. For example, consider the model in Figure 10. Suppose that the memory receives a request for an exclusive copy when processors p_0 and p_1 have a copy of the block. The memory inserts an invalidation message to the last available slot of the receiving channels of p_0 and p_1. In a state enumeration method, this is done by modifying the state variables of the receiving channels of p_0 and p_1 (Figure 20). But, in the SMV, the memory module needs to export the invalidation messages to the outgoing wires connected to the input ports of the receiving channel modules. The receiving channel module must then acknowledge to the memory module that the message has been received.

In addition, in the symbolic model checking method, the possible values of writable data are normally limited to 0 and 1 so that one Boolean variable is sufficient to keep track of them. To be able to tag copies as *Fresh* or *Obsolete* as is easily done in an enumeration method, we can anticipate a model of unmanageable complexity.

Of course, the sizes of the OBDDs can be reduced by building and composing smaller OBDDs, one for each individual statement of state assignment. However, the tradeoff is longer verification time [Hu et al. 1992]. When the model size grows, it is not clear whether the time complexity is tractable and the OBDDs are reasonably small.

8.2 Generality and Applicability

Murφ, SMV, and Ever are general methods for verifying finite-state systems, whereas the abstraction in the SSM method is applicable to systems made of identical processes only. However, general methodologies for wide application domains are often tailored to specific problems in order to improve the efficiency of the tool. Therefore the fact that SSM employs a specialized abstraction for the verification of cache protocols is really not a serious problem. Although its applicability is limited to protocols with identical processes, SSM provides a better solution than current methods for the limited domain of cache protocols, which are a crucial component of shared-memory machines.

Murφ and Ever have been used to verify a restricted model of the Stanford DASH directory-based protocol [Ip and Dill 1993a,b; Lenoski et al. 1990]. SMV has been applied to two snooping protocols (the protocols of the Encore Gigamax [McMillan and Schwalbe 1991] and of the Futurebus+ [Clarke et al. 1993b]). SSM has been applied to snooping protocols [Pong and Dubois 1995], to two directory-based protocols [Pong et al. 1994; Pong 1995], to the S3.mp linked-list protocol [Pong et al. 1995], and to the delayed consistency protocol [Dubois et al. 1991; Pong and Dubois 1996] under a weak memory consistency model [Adve and Hill 1990a, 1993; Dubois et al. 1986]. From these examples it appears that every method can be applied successfully to all types of cache protocols. The question remains whether they can be applied to verify protocols at a reasonable cost.

As shown in previous sections, traditional state enumeration approaches do not succeed for large models, even with the symmetry extensions. The reason is clear: a symmetric method can reduce the size of the state space by a factor of $n!$ at most, given a protocol model with n processors [Ip and Dill 1993a]. The state space is often very large even for very small values of n. Consequently, the reduction afforded by symmetry is limited. It was argued in Dill et al. [1992] that the state space explosion problem only occurs for scaled-up models and that all errors can in fact be found with small models. A recent study of the S3.mp protocol demonstrates that this may not be true [Pong et al. 1995]. If the protocol is faithfully modeled, the state space explosion problem also exists for small models. It was also shown that errors may go undetected for small models. Intuitively, the size of a model must be scaled up proportionally with the increased complexity of the protocol in order to reach a high level of confidence in the verification. When relaxed consistency models [Adve and Hill 1990a; Gharachorloo et al. 1991, 1990] and latency tolerance techniques [Dahlgren et al. 1994; Dubois et al. 1991] come into play, we can expect that the overwhelming complexity of protocols will hamper this type of technique. Nevertheless, a state enumeration approach applied to a small model is capable of detecting a large number of errors and is very effective as a debugging tool.

So far, symbolic model checking techniques have only been applied to snooping protocols. It is still not clear whether they can overcome the problem of the enormous size of the OBDD typically observed [Hu and Dill 1993a,b]. Although many improvements such as exploiting disjunctive partitioned next-state relations [Burch et al. 1991] or employing implicitly conjoined invariants [Hu and Dill 1993a] can work well, whether the model size can be scaled up without causing an OBDD-size explosion is still unclear. On the other hand, these approaches can verify any property specified in CTL, which is very expressive and flexible. They also provide a formal way to reason about the infinite behavior of the system, and hence, properties such as absence of livelocks can be easily checked.

The reduction of the state space in SSM is so drastic that we can contemplate the efficient verification of more complex protocols. Unfortunately, the abstraction in SSM essentially removes the processor identifiers. As a result, for linked-list protocols such as the SCI (Scalable Coherent Interface) [James et al. 1990] or the protocol of S3.mp [Nowatzyk 1994], SSM may not be efficient [Pong et al. 1995].

8.3 Automaticity and Amenity

Murφ has the clear advantage with respect to these two properties. Its specification language provides common programming constructs such as scalar-valued variables, arrays, and records. State transitions are naturally described by assignment statements. The language also supports flow control statements such as *for* loops, *if-else*, and *switch*. Protocol models in Murφ are easily programmed and the tool is fully automated to the point that the full state space is generated automatically, given a specification program. Murφ also provides a very nice feature, called *trace generation*: when an error is detected, the tool outputs a trace, which leads the user to the erroneous state and helps fix the error.

SMV executes the verification task automatically as well. Unfortunately, as we have detailed in Section 6.2.3, the system under verification must be modeled in terms of Boolean formulas. Complex data structures and convenient flow controls are not supported. Also protocol models may include many state variables that are not related to the behavior of the protocol, but are needed to synchronize communicating modules. As a result, the specification of the protocol can be lengthy and difficult to build. Trivial changes, such as increasing the number of processors, can result in major modifications of the description. In addition to efficiency and accuracy, the ability to build and modify protocol models rapidly is highly desirable in any formal verification method. The high complexity of describing a model does not favor the symbolic model checking method in the context of cache protocol verification.

The SSM system [Pong 1995] is a fully automated tool that has a description language similar to Murφ's. The system accepts a description of the state variables needed to form a global state as shown in Figure 20. Users must specify the next-state transition rules and the initial state. With the protocol description, procedures (in the C programming language) to manipulate the state information and to check the containment relation between two states are generated automatically. An executable image of the verifier is then produced with a C compiler such as *gcc*. The SSM system also provides a trace generation facility to trace back protocol error. Unfortunately, due to the abstract state representation in SSM, the traces are lengthy and hard to read.

9. RELATED METHODS

Alternatives to methods that enumerate states have also been applied to the verification of cache protocols. Whereas these alternatives are of limited applicability as practical, general-purpose verification methods, we mention them here for completeness.

9.1 Error Prevention

Kubiatowicz [1993] described a scheme in which the number of possible state transitions are bounded and certain types of coherence messages cannot exist in the system at the same time in order to achieve a correct cache protocol. For instance, protocol errors often occur in the write-back transition of dirty cache copies [Archibald 1987; Pong et al. 1995]. To write-back these blocks safely, a complex acknowledgment scheme can eliminate many errors by preventing requests to a block in a write-back transition. We classify this approach as an error prevention scheme.

A drawback of this scheme is that verification interferes with protocol designs, which are forced to be simple in order to facilitate manual verification. As a result, it limits possible protocol optimizations.

9.2 Theorem Proving

Another alternative to methods based on state models is *theorem-proving* [Gjessing et al. 1991; Loewenstein and Dill 1990], which essentially reduces a formal verification to the derivation of a mathematical proof. Typically, the specification is written in the form of a set *Spec* of logic formulas and the implementation is given as a second set *Imp* of logic formulas which are considered as *axioms* or valid *theorems* in the logic so that a proof can be derived. The correspondence relation between *Spec* and *Imp* is a theorem, which is proved by a *theorem-prover*. The main strength of this approach is that it naturally leads to a hierarchical verification method.

Loewenstein and Dill [1990] used *Higher-Order Logic* (*HOL*) to verify a directory-based protocol. However, their verification is based on the assumption of atomic memory accesses. Specifically, they show that data consistency can be achieved by allowing one and only one write to propagate at any time. The memory directory entry behaves as a serialization device for all coherence events to the same memory block. More work is needed to show that this method can be extended to model protocols at the message-passing level and can perform reasonably well.

Gjessing et al. [1991] have attempted to specify the SCI protocol [James et al. 1990] formally in a topdown manner. An abstract memory system without caches is first defined as the intended behavior for the system implementing SCI. Gradually the model is refined with additional intermediate cache states and fine-grain atomic actions. Finally, full concurrency is exploited at the message-passing level. However, the proof of correctness of SCI is not carried out in the paper.

9.3 Abstract Interpretation

As opposed to state enumeration methods, abstract interpretation [Cousot and Cousot 1992] is a technique that reduces a complex system to a more abstract one. Properties are then verified on the abstracted system with reasonable complexity. Consider a concrete state transition system S of variable sets X and an abstracted system S^A of variable set X^A. In general, S^A is called a *ρ-abstraction* of S if:

(1) For each transition $T(X, X')$ of S, there exists a corresponding transition $T^A(X^A, X^{A\prime})$ in S^A.

(2) Initial state Init(X^A) of S^A corresponds to the initial state Init(X) of S.

In fact, the concept of abstraction relation is similar to the *simulation* (or *refinement*) relation between two state transition systems [Loewenstein and Dill 1990]. A critical step, which unfortunately cannot be fully automated, is to find the abstraction relation. Abstract interpretation has been applied to the verification of the lazy caching algorithm [Afek et al. 1989] at the behavioral level [Cousot and Cousot 1992]. This theoretical result is very nice and important, because it shows that the conditions of Section 2.2 are not absolutely necessary for supporting the sequential consistency model. However, lazy caching is not really a cache coherence protocol and is of limited practical importance because every store triggers a broadcast operation. We feel that this method may be promising for verifying memory models. However, it is not clear how the abstraction relation can be found in general for complex systems and more research must be done before the approach can be truly useful for the verification problem of cache protocols.

9.4 Reasoning about Models with a Large Number of Identical Processes

The applicability of current state enumeration methods is limited to small-scale models. For simple protocols, this is not a serious problem; however, as the complexity of protocols keeps increasing, the reliability of verification methods must be carefully re-evaluated. Since the verification of cache protocols falls into a special class of verification problems in which the system is composed of identical processes, research on reasoning about the correctness of systems with identical component processes is relevant to this survey.

Browne et al. [1989] verified systems with many identical finite state processes by finding a *correspondence* between two (global state) structures M and M' representing systems of manageable and overwhelming complexities, respectively. A given relation E determines whether there is a correspondence relation between two structures. If two structures correspond, formulas in temporal logic are evaluated on the simpler *base* state machine. Not surprisingly, this scheme is similar to the abstract interpretation, and in practice, may be difficult to apply to complex protocols. First, considerable human ingenuity is needed to discover the correspondence relation between the two structures. Second, constructing the global state transition diagram of the base machine may be a difficult procedure itself, because the base machine capturing the essential properties of the system may still be very complex.

Kurshan and McMillan [1989] suggested another method using partial-order relations among processes to form an *invariant* process. A process is invariant if the composition of the invariant process with a new process is less than or equal to the invariant. This method has great potential, but the construction of the invariant process in Kurshan and McMillan [1989] is not automated and requires considerable ingenuity for complex protocols. A similar approach was independently pursued by Wolper and Lovinfosse [1989]. However, all the examples presented in Kurshan and McMillan [1989] and Wolper and Lovinfosse [1989] are relatively simple and the verifications mostly prove that the system meets the "behavioral" requirement of the specification. It is not clear whether these methods will be applicable to complex cache protocols beyond the behavior level.

10. CONCLUDING REMARKS AND FUTURE RESEARCH

In this article, we have presented a comprehensive survey of ongoing research efforts for the verification of cache coherence protocols. Three methods based on state enumeration, (symbolic) model checking, and symbolic state model have been thoroughly discussed. In each case we have focused on their underlying approach for dealing with the state space explosion and their algorithmic procedures for verification.

In summary, state enumeration is conceptually the simplest method but its application is limited by the state space explosion problem. Nevertheless, several studies have demonstrated that most design errors can be found quickly in small-scale models, which suggests that this method is a useful debugging tool in the early design phase.

Model checking can check any property formulated in temporal logic, but the state space explosion problem also limits its applicability. The symbolic model checking method does not rely on an explicit data structure to represent the states; rather OBDDs internally represent the next-state relations and the state space. Unfortunately, the OBDDs for next-state relations are often too large to compute, which is known as the OBDD size explosion problem. Several approaches have been proposed to overcome this problem. Some look for optimal variable orderings; others use disjunctive partitioned next-state relations instead of the giant OBDD obtained for the entire set of transition relations. More work is needed to prove that this method can scale up with the complexity and size of the models. Improvements must also be made in the ease of programming protocol models.

The method based on symbolic state models has been successful in some studies. Based on a specific property of cache protocols that all cache components are homogeneous, the method uses a specialized abstraction that maps global states to more abstract states. Thus the method can reduce the state space substantially, representing a large set of states with a single abstract state. It has the added advantage that protocols are verified independently of the system size and consequently, the verification result is reliable. However, building an abstracted model of a protocol is not always straightforward.

The approaches surveyed in this article verify the property of consistency, for which the state of a single block must be tracked. Moreover, the illustrated protocol model does not include the latency tolerance hardware found in weakly ordered systems. Verification of cache protocols is somewhat more complex in the presence of latency tolerance hardware. Consider the execution of Figure 21 in a weakly ordered system. In this particular execution, the write by p_0 and the read by p_1 are ordered by paired Test&Set and Unset synchronization accesses. Since the read of p_1 cannot be completed before the write of p_0 due to the explicit synchronization separating them, p_0 does not necessarily block at the write operation while waiting for the invalidation of p_1's copy. The only requirement for a correct execution is that the value written by p_0 must be visible to p_1 before p_1 reads it.

To enforce this requirement the hardware usually relies on the lock accesses. The value could propagate from p_0 to p_1 when p_0 releases the lock. Thus cache coherence is not enforced on-the-fly and is delayed. To verify cache protocols in such systems, the state expansion process which searches for all reachable system states must take into account

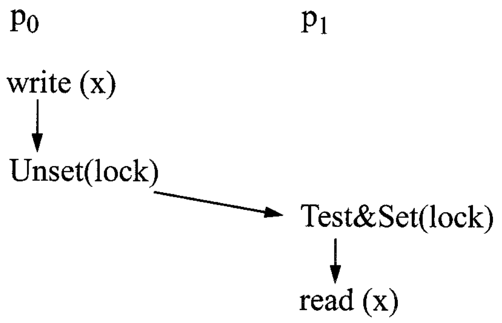


Figure 21. Explicit synchronization in weakly oriented memory model.

synchronization accesses as well as regular data accesses. In the execution sequence of Figure 21, p_1 is allowed to issue its read *only* after p_0 and p_1 have performed their Unset and Test&Set operations, respectively. This is only a problem of modeling protocols. The methods surveyed in this article can be adapted as we showed in Pong [1995] and Pong and Dubois [1996].

Several problems still need to be solved. One is the lack of a good methodology to deal with linked-list directory-based protocols such as SCI. The major difficulty is to abstract the linked-list efficiently, while correctly preserving the properties to check. A possible solution is to separate the problem of verifying the correct maintenance of the sharing lists from the problem of verifying coherence. Alternatively, we may employ some inductive process to infer the correctness of a large model from the verification results of smaller models.

A more difficult problem is the verification of the correct ordering of *all* memory accesses [Lamport 1979; Scheurich and Dubois 1987] according to the memory consistency model. Several frameworks have been introduced to specify memory consistency models formally [Collier 1992; Gibbons et al. 1991; Sindhu et al. 1992]. Formal specifications of memory consistency models are very important, for several reasons. First, some properties of systems can be proven. For example, systems can be proven to have distinguishable or indistinguishable behavior [Collier 1992; Graf 1994]. Second, formal specifications provide a clear logical model of access orderings offered by the memory system to the programmer or to the compiler. It is easier for the programmers and for the compiler to generate correct code. For example, it has been proposed that the programmer write or that the compiler generate *properly labeled* programs so that the execution results are indistinguishable from those executed on the sequential consistency model [Gharachorloo et al. 1990]. Third, formalism is a first step towards formal verification of memory access orders.

Memory access ordering is often seen as an orthogonal issue to the problem of cache coherence since it is an issue whether or not there are caches. In fact, the current research on memory consistency models barely considers the behavior of caches or cache protocols. Cache coherence protocols of practical interest are still designed based on the rules presented in Section 2.2. In short, with respect to a single memory location, one and only one write operation is allowed to propagate at a time. These rules are essentially sufficient conditions for supporting the general coherence property [Scheurich 1989] which is a requirement for useful memory models such as the sequential consistency [Lamport 1979], weak ordering [Scheurich 1989], and release consistency [Gharachorloo et al. 1991] in a cache-coherent shared-memory system. Even for protocols developed in weaker memory models [Dubois et al. 1991; Lenoski et al. 1990], the only relaxation of the preceding rules is to guarantee that a store operation finishes before the next (release) synchronization executed by the local processor, that is, enforcing general coherence at synchronization points.

From the standpoint of the techniques surveyed in this article, the memory consistency model is only meaningful in the presence of caches. The memory model tells us when the stores must propagate and the cache consistency protocol must propagate them in time for a subsequent read by a remote processor. The methods based on state expansion covered in this survey are effective at proving that the cache consistency hardware correctly fulfills this function. Unfortunately, no framework has been proposed so far to deal with the memory consistency model in the context of formal verification based on state expansion. Finding such a framework would certainly be a breakthrough in the field of formal architecture verification.

REFERENCES

Adve, S. V. and Hill, M. D. 1990a. Weak ordering—a new definition. In *Proceedings of the 17th International Symposium on Computer Architecture* (May), 2–14.

Adve, S. V. and Hill, M. D. 1990b. Implementing sequential consistency in cache-based systems. In *Proceedings of the 1990 International Conference on Parallel Processing,* I47–I50.

Adve, S. V. and Hill, M. D. 1993. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.* (Aug.), 613–624. (also Tech. Rep. #1051, University of Wisconsin).

Afek, Y., Brown, G., and Merritt, M. 1989. A lazy cache algorithm. In *Proceedings of the Symposium on Parallel Algorithm and Architecture* (June), 209–223.

Archibald, J. 1987. The cache coherence problem in shared-memory multiprocessors. Ph.D. Dissertation, University of Washington, Feb.

Archibald, J. and Baer, J.-L. 1986. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst. 4,* 4 (Nov.), 273–298.

Baer, J.-L. and Girault, C. 1985. A Petri net model for a solution to the cache coherence problem. In *Proceedings of the First Conference on Supercomputing Systems,* 680–689.

Barroso, L., Jeong, J., Öner, K., Ramamurthy, K., and Dubois, M. 1995. RPM: A rapid prototyping engine for multiprocessors. *IEEE Computer* (Feb.).

Bochmann, G. V. and Sunshine, C. A. 1980. Formal methods in communication protocol design. *IEEE Trans. Commun. COM-28,* 4 (April), 624–631.

Brown, G. M. 1990. Asynchronous multicaches. *Distrib. Comput.* 4, 31–36.

Browne, M. C., Clarke, E. M., Dill, D., and

MISHRA, B. 1986. Automatic verification of sequential circuits using temporal logic. *IEEE Trans. Comput.* (Dec.), 1035–1044.

BROWNE, M. C., CLARKE, E. M., AND GRUMBERG, O. 1989. Reasoning about networks with many identical finite state processes. *Inf. Comput. 81,* 13–31.

BRYANT, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput. C-35,* 8 (Aug.), 677–691.

BRYANT, R. E. 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv. 24,* 3, 293–318.

BURCH, J. R., CLARKE, E. M., AND LONG, D. E. 1991. Symbolic model checking with partitioned transition relations. In *Proceedings of the International Conference on VLSI.*

CENSIER, L. M. AND FEAUTRIER, P. 1978. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput. C-27,* 12 (Dec.), 1112–1118.

CHIODO, M., SHIPLE, T. R., SANGIOVANNI-VINCENTELLI, A. L., AND BRAYTON, R. K. 1992. Automatic compositional minimization in CTL model checking. In *Proceedings of the International Symposium on Computer-Aided Design,* 172–178.

CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8,* 2 (April), 244–263.

CLARKE, E. M., FILKORN, T., AND JHA, S. 1993a. Exploiting symmetry in temporal logic model checking. In *Proceedings of the Fifth International Conference on Computer-Aided Verification* (June), 450–462.

CLARKE, E. M., GRUMBERG, O., HIRAISHI, H., JHA, S., LONG, D. E., MCMILLAN, K. L., AND NESS, L. A. 1993b. Verification of the Futerbus+ cache coherence protocol. In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and Their Applications* (April).

COLLIER, W. W. 1992. *Reasoning About Parallel Architectures.* Prentice-Hall, Englewood Cliffs, NJ.

COUDERT, O., BERTHET, C., AND MADRE, J. C. 1989. Verification of synchronous sequential machines based on symbolic execution. In *LNCS: Automatic Verification Methods for Finite State Systems,* J. Sifakis, Ed., Vol. 407, Springer-Verlag, New York (June), 365–373.

COUDERT, O., MADRE, J. C., AND BERTHET, C. 1990. Verifying temporal properties of sequential machines without building their state diagrams. *Proceedings of the Second Workshop on Computer-Aided Verification,* Springer-Verlag, New York.

COUSOT, P. AND COUSOT, R. 1992. Abstract interpretation frameworks. *J. Logic Comput. 2,* 4 (Aug.), 511–547.

DAHLGREN, F., DUBOIS, M., AND STENSTRÖM, P. 1994. Combined performance gains of simple cache protocol extensions. In *Proceedings of the 21st International Symposium on Computer Architecture,* 187–197.

DANTHINE, A. S. 1980. Protocol representation with finite-state models. *IEEE Trans. Commun. COM-28,* 4 (April), 632–642.

DILL, D. L., DREXLER, A. J., HU, A. J., AND YANG, C. H. 1992. Protocol verification as a hardware design aid. *International Conference on Computer Design: VLSI in Computers and Processors* (Oct.), 522–525.

DUBOIS, M. AND SCHEURICH, C. 1990. Memory access dependencies in shared-memory multiprocessors. *IEEE Trans. Softw. Eng. 16,* 6 (June), 660–673.

DUBOIS, M., SCHEURICH, C., AND BRIGGS, F. A. 1986. Memory access buffering in multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture* (June), 434–442.

DUBOIS, M., WANG, J. C., BARROSO, L., LEE, K., AND CHEN, Y. S. 1991. Delayed consistency and its effects on the miss rate of parallel programs. *Supercomputing* (Nov.), 197–206.

EMERSON, E. A. AND SISTLA, A. P. 1993. Symmetry and model checking. In *Proceedings of the Fifth International Workshop on Computer-Aided Verification* (June), 463–478.

ENDER, R., FILKORN, T., AND TAUBNER, D. 1991. Generating BDDs for symbolic model checking in CCS. In *Proceedings of the Third International Workshop on Computer Aided Verification* (July), 203–213.

FRIEDMAN, S. J. AND SUPOWIT, K. J. 1990. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. Comput. 39,* 5 (May), 710–713.

FUJITA, M., FUJISAWA, H. AND KAWATO, N. 1988. Evaluation and improvements of Boolean comparison method based on binary decision diagram. In *Proceedings of the ICCAD,* 2–5.

GALLES, M. AND WILLIAMS, E. 1994. Performance optimizations and verification methodology of the SGI challenge multiprocessor. In *Hawaii International Conference on System Sciences,* Vol. 1 (Jan.), 134–143.

GJESSING, S., KROGDAHL, S., AND MUNTHE-KAAS, E. 1991. A top down approach to the formal specification of SCI cache coherence. In *Proceedings of the Third International Workshop on Computer Aided Verification* (July), 83–91.

GODEFROID, P. 1990. Using partial orders to improve automatic verification methods. In *Proceedings of the Second International Work-*

shop on Computer-Aided Verification (June), 176–185.

Godefroid, P., Holzmann, G. J. and Pirottin, D. 1992. State space caching revisited. In *Proceedings of the Fifth International Workshop on Computer-Aided Verification* (June/July), 178–191.

Gharachorloo, K., Gupta, A., and Hennessy, J. L. 1991. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proceedings of the Fourth ASPLOS* (April), 245–257.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. 1990. Memory consistency and event ordering in shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture* (May), 15–26.

Gibbons, P. B., Merritt, M., and Gharachorloo, K. 1991. Proving sequential consistency of high performance shared memories. In *Proceedings of the Third ACM Symposium on Parallel Algorithm and Architectures,* 292–303.

Graf, S. 1994. Verification of a distributed cache memory by using abstractions. In *Proceedings of the Sixth International Conference on Computer-Aided Verification,* 207–219.

Graf, S. and Loiseaux, C. 1993. A tool for symbolic program verification and abstraction. In *Proceedings of the Fifth International Conference on Computer-Aided Verification.*

Graf, S., Richier, J.-L., Rodriguez, C., and Voiron, J. 1989. What are the limits of model checking methods for the verification of real life protocols? In *Automatic Verification Methods for Finite State Systems* (June), 275–285.

Gupta, A., Hennessy, J., Gharachorloo, K., Mowry, T., and Weber, W. D. 1991. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th International Symposium on Computer Architecture* (May).

Haridi, S. and Hagersten, E. 1989. The cache coherence protocol of the data diffusion machine. In *Proceedings PARLE 89,* Vol. 1, Springer-Verlag, 1–18.

Holzmann, G. J. 1985. Tracing protocols. *AT&T Tech. J. 64,* 12, 2413–2434.

Holzmann, G. J. 1990. Algorithms for automated protocol verification. *AT&T Tech. J.* (Jan./Feb.).

Holzmann, G. J. 1991. *Design and Validation of Computer Protocols.* Prentice-Hall International Editions.

Hoare, C. A. R. 1978. Communicating sequential processes. *Commun. ACM 21,* 8 (Aug.), 666–677.

Hu, A. J. and Dill, D. L. 1993a. Efficient verification with BDDs using implicitly conjoined invariants. In *Proceedings of the Fifth International Workshop on Computer Aided Verification,* 3–14.

Hu, A. J. and Dill, D. L. 1993b. Reducing BDD size by exploiting functional dependencies. In *30th ACM/IEEE Design Automation Conference.*

Hu, A. J., Dill, D. L., Drexler, A. J., and Yang, C. H. 1992. Higher-level specification and verification with BDDs. In *Proceedings of the Fourth International Workshop on Computer Aided Verification* (July), 82–95.

Ip, C. N. and Dill, D. L. 1993a. Efficient verification of symmetric concurrent systems. In *International Conference on Computer Design: VLSI in Computers and Processors* (Oct.).

Ip, C. N. and Dill, D. L. 1993b. Better verification through symmetry. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and Their Applications* (April), 87–100.

Jard, C. and Jeron, T. 1991. Bounded-memory algorithms for verification on-the-fly. In *Proceedings of the Third International Workshop on Computer Aided Verification* (July), 192–202.

James, D. V. et al. 1990. Distributed-directory scheme: Scalable coherence interface. *IEEE Computer 23,* 6 (June), 74–77.

Kubiatowicz, J. D. 1993. Closing the window of vulnerability in multiphase memory transactions: The Alewife transaction store. M.S. Thesis, Dept. of Electrical Engineering and Computer Science, MIT, Feb.

Kurshan, R. P. and McMillan, K. 1989. A structural induction theorem for processes. *ACM Symposium on Principles of Distributed Computing,* 239–247.

Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28,* 9 (Sept.), 690–691.

Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. 1990. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture* (June), 148–159.

Liaw, H.-T. and Lin, C.-S. 1992. On the OBDD-representation of general Boolean functions. *IEEE Trans. Comput. 41,* 6 (June), 661–664.

Loewenstein, P. and Dill, D. L. 1990. Verification of a multiprocessor cache protocol using simulation relations and higher-order logic. In *Proceedings of the Second International Workshop on Computer Aided Verification* (June), 302–311.

Madre, J.-C. and Billon, J.-P. 1988. Proving circuit correctness using formal comparison between expected and extracted behavior. In *Proceedings of the 25th ACM/IEEE Design Automation Conference,* 205–210.

MALIK, S., WANG, A. R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1988. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proceedings of the ICCAD,* 6–9.

MCMILLAN, K. L. 1992. Symbolic model checking: An approach to the state explosion problem. Ph.D. Dissertation, Carnegie Mellon University, May.

MCMILLAN, K. L. AND SCHWALBE, J. 1991. Formal verification of the Gigamax cache consistency protocol. In *Proceedings of the ISSM International Conference on Parallel and Distributed Computing* (Oct.).

NANDA, A. K. AND BHUYAN, L. N. 1992. A formal specification and verification technique for cache coherence protocols. In *Proceedings of the 1992 International Conference on Parallel Processing,* I22–I26.

NOWATZYK, A., AYBAY, G., BROWNE, M., KELLY, E., PARKIN, M., RADKE, B., AND VISHIN, S. 1994. The S3.mp scalable shared memory multiprocessor. *HICCS.*

PNUELI, A. 1977. The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science,* 46–57.

PNUELI, A. 1981. The temporal logic of concurrent programs. *Theor. Comput. Sci.,* 45–60.

PONG, F. AND DUBOIS, M. 1993a. The verification of cache coherence protocols. In *Proceedings of the Fifth Annual Symposium on Parallel Algorithm and Architecture* (June), 11–20.

PONG, F. AND DUBOIS, M. 1993b. Correctness of a directory-based cache coherence protocol: Early experience. In *Proceedings of the Fifth Annual Symposium on Parallel and Distributed Processing* (Dec.), 37–44.

PONG, F., STENSTRÖM, P., AND DUBOIS, M. 1994. An integrated methodology for the verification of directory-based cache protocols. In *Proceedings of the 1994 International Conference on Parallel Processing,* I158–I165.

PONG, F. AND DUBOIS, M. 1995. A new approach for the verification of cache coherence protocols. *IEEE Trans. Parallel Distrib. Syst. 6,* 8 (Aug.), 773–787.

PONG, F., NOWATZYK, A., AYBAY, G., AND DUBOIS, M. 1995. Verifying distributed directory-based cache coherence protocols: S3.mp, a case study. In *Proceedings of the First International EURO-PAR Conference* (Aug.), 287–300.

PONG, F. 1995. Symbolic state model; A new approach for the verification of cache coherence protocols. Ph.D. Dissertation, Dept. of Electrical Engineering-Systems, University of Southern California, Aug.

PONG, F. AND DUBOIS, M. 1996. Formal verification of delayed consistency protocols. In *Proceedings of the Tenth International Parallel Processing Symposium* (April), 124–131.

ROTHNIE, J. 1992. Overview of the KSR1 computer system. Kendall Square Res. Rep. TR 9202001, March.

RUDOLF, L. AND SEGALL, Z. 1984. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the Eleventh International Symposium on Computer Architecture* (June), 340–347.

SCHEURICH, C. 1989. Access ordering and coherence in shared memory multiprocessors. Ph.D. Thesis, University of Southern California.

SCHEURICH, C. AND DUBOIS, M. 1987. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture* (June), 234–243.

SCHEURICH, C. AND DUBOIS, M. 1991. Lockup-free caches in high-performance multiprocessors. *J. Parallel Distrib. Comput. 11,* 25–36.

SHASHA, D. AND SNIR, M. 1988. Efficient and correct execution of parallel programs that shared memory. *ACM Trans. Program. Lang. Syst. 10,* 2 (April), 282–312.

SHIPLE, T. R., CHIODO, M., SANGIOVANNI-VINCENTELLI, A. L., AND BRAYTON, R. K. 1992. Automatic reduction in CTL compositional model checking. In *Proceedings of the International Workshop on Computer-Aided Verification* (June), 234–247.

SINDHU, P. S., FRAILONG, J-M., AND CEKLEOV, M. 1992. Formal specification of memory models. In *Scalable Shared Memory Multiprocessors,* M. Dubois and S. Thakkar, Eds. Kluwer, Norwell, MA.

STENSTRÖM, P. 1990. A survey of cache coherence schemes for multiprocessors. *IEEE Computer 23,* 6 (June), 12–24.

SWEAZEY, P. AND SMITH, A. J. 1986. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *Proceedings of the 13th International Symposium on Computer Architecture,* 414–423.

TOMAS, D. E. AND MOORBY, P. 1995. *The Verilog Hardware Description Language.* Kluwer Academic.

WEGENER, I. 1994. The size of reduced OBDDs and optimal read-once branching programs for almost all Boolean functions. *IEEE Trans. Comput. 43,* 11 (Nov.), 1262–1269.

WILSON, A. W., JR. 1987. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture,* 244–252.

WOLPER, P. AND LEROY, D. 1993. Reliable hashing without collision detection. In *Proceedings of the Fifth Workshop on Computer Aided Verification* (June).

WOLPER, P. AND LOVINFOSSE, V. 1989. Verifying properties of large sets of processes with network invariants. In *International Workshop on Automatic Verification Methods for Finite State Systems* (June), 68–80.

YEN, W. C., YEN, W. L., AND FU, K.-S. 1985. Data coherence problem in a multicache system. *IEEE Trans. Comput. C-34,* 1 (Jan.).

YUANG, M. C. 1988. Survey of protocol verification techniques based on finite state machine models. In *Proceedings of the Computer Networking Symposium* (April), 164–172.

Received December 1994; revised May 1996; accepted December 1996