Formal Automatic Verification of Cache Coherence in Multiprocessors with Relaxed Memory Models

Fong Pong, Michel Dubois*
Computer Systems and Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-33
February, 2000

E-mail: fpong@hpl.hp.com
dubois@paris.usc.edu

Shared-Memory Multiprocessor, relaxed memory consistency models, delayed consistency, verification, symbolic state model

State-based, formal methods have been successfully applied to the automatic verification of cache coherence in sequentially consistent systems. However, coherence in shared-memory multiprocessors under a relaxed memory model is much more complex to verify automatically. With relaxed memory models, incoming invalidations and outgoing updates can be delayed in each cache while processors are allowed to race ahead. This buffering of memory accesses considerably increases the amount of state in each cache and the complexity of protocol interactions. Moreover, because caches can hold inconsistent copies of the same data for long periods of time, coherence cannot be verified by simply checking that cached copies are identical at all times.

This paper makes two major contributions. First, we demonstrate how to model and verify cache coherence under a relaxed memory model in the context of state-based verification methods. Frameworks for modeling the hardware and for generating correct memory access sequences driving the hardware model are developed. We also show correctness properties which must be verified on the hardware model. Second, we demonstrate a successful application of a state-based verification tool called SSM for the verification of delayed protocol, an aggressive protocol for relaxed memory models. SSM is based on an abstraction technique preserving the properties to verify. We show that with classical, explicit approaches the verification of cache coherence is realistically unfeasible because of the state space explosion problem whereas SSM is able to verify protocols both at both behavioral and message-passing levels.

* Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA 90089-2562
© Copyright Hewlett-Packard Company 2000

1 Introduction

In this paper, we develop the framework to verify automatically and formally cache coherence protocols under relaxed memory models using formal, state-based approaches. We also demonstrate the feasibility of verifying such cache protocols by applying a state based verification tool called SSM (for "Symbolic State Model") [25] to an aggressive cache protocol under relaxed memory models, called the delayed protocol [10]. The delayed protocol is an aggressive implementation of release consistency [14], in which the sending of invalidations and the effect of received invalidations are *deliberately* delayed until the next *release* and the next *acquire* respectively. (Releases and acquires are synchronization primitives to order accesses of concurrent processes sharing writable data.) As compared to protocols under a strong memory model [9, 18], the performance improvements of the delayed protocol stem from the more effective use of store buffers, the more aggressive pipelineing of memory accesses, and the reduction of *false sharing* effects [11].

State-based, formal methods have been successfully applied to the automatic verification of cache coherence under a strong memory model [16, 17, 26]. Under a strong memory model, memory accesses in the verification model are limited to loads and stores. In every step of a state-based verification model, any processor can issue a load or a store unless it is blocked due to a prior pending access and the issuance of a memory access is not restricted by the states of other processors. Coherent accesses to a memory block, even to different words in the block, are serialized and the verification problem is simplified by assimilating a memory block to a single word since the coherence unit is a memory block [21].

The verification of cache coherence under a relaxed memory model is much more complex. First of all, the sequence of memory accesses driving the system cannot just be any arbitrary sequence of loads and stores. Consider the execution of figure 1 in a system with a relaxed memory model [1, 2, 9, 14]. 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 complete before the write of p_0 due to the explicit synchronization, p_0 does not need to block at the write waiting for the invalidation of p_1's copy. The only requirement for a correct execution is that the value written by p_0 becomes visible to p_1 before p_1 reads it. To enforce this requirement the hardware relies on lock accesses. The

invalidation can propagate from p_0 to p_1 when p_0 releases the lock (Unset) and must reach the cache of p_1 when p_1 executes the acquire (Test&Set). To verify cache protocols in such systems, the model must take into account synchronizations on top of regular data accesses. In the execution sequence of figure 1, p_1 is allowed to issue its read only after p_0 and p_1 have performed their Unset and Test&Set respectively. Clearly, the state expansion process of a state enumeration method must be restricted to generate only such legal sequences of reads, writes and synchronizations.

Second the hardware to model is much more complex. It includes buffers for stores and invalidations, write caches and lockup-free caches. These additional mechanisms must be modeled by simple constructs in the hardware model and considerably increase the amount of state associated with each cache. Thus they greatly contribute to the state explosion problem plaguing state-based verification approaches.

A third problem is how to model and formulate the condition of data coherence. In many protocols [3], if a cache block is in the "*shared*" state, an implication is that other processors may have cached the block and all copies are identical to the main memory copy. This semantic relation between cache states and data copies is not guaranteed in a cache protocol under a relaxed memory model. We cannot rely on checking if all individual cache states are compatible [20]. Because the enforcement of data consistency is delayed until synchronization points, caches may have data copies with incoherent values for long periods of time. Even if a block is still accessible in a cache, some of its words may be stale and, in general, the effective state of a word within a block is different from the state of the block. The verification model must track the states of the block and of its words.

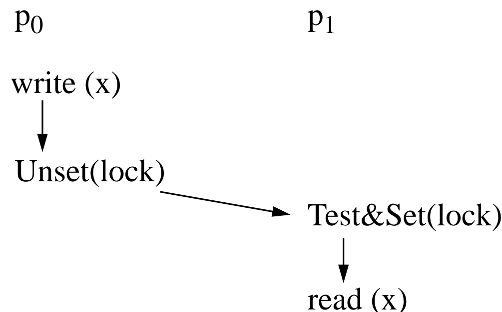

Figure 1. Explicit Synchronization in a Relaxed Memory Model.

Once these problems are solved we must adopt a verification strategy such that protocols for system of realistic sizes can be verified. We will show that classical, explicit state-based methods fail in this regard for systems with relaxed memory models. By contrast we will also show that the sate-based verification method based on *symbolic state models* (SSM) can successfully verify coherence at both behavioral and message-passing levels. SSM [21] was conceived specifically for the verification of cache protocols. It verifies cache protocols by exploring the state space of a protocol as in conventional state enumeration methods [15], but it exploits the symmetry and homogeneity of cache-based systems to reduce the size of the state space.

In traditional state-based methods, the behavior of the caches is specified by homogeneous, local finite state automata (representing caches) interacting through the coherence protocol. Each and every cache is specified individually in the model. Let M be the *global* state machine in this explicit model. M is the composition of all local state automata (caches) and its state is called the *global state* (or *system state*). To simplify the global state and reduce the complexity of the search, the SSM provides a set of abstraction constructors to represent global states in M concisely, without tracking the exact number of caches in particular states. The state space of the reduced finite state machine in SSM denoted M_r is much smaller than the state space of M. The verification of complex protocols on M_r is more feasible than on M because the verification time and the amount of memory to store global states are drastically cut. In contrast to other approaches [24], SSM enables the verification of cache protocols for any system size while avoiding the state space explosion problem [15].

This paper makes two major contributions. First, we demonstrate how to model and verify a cache protocol under a relaxed memory model in the context of state-based verification methods. Three components are needed in a verification procedure for cache protocols: a model of the hardware, a model for the memory access sequences driving the model and a set of correctness conditions to verify. We show how to model complex latency tolerance mechanisms such as invalidation and store buffers, write caches and lock-up free caches. This general modeling approach can be extended to other protocols under relaxed memory models such as protocols with delayed updates instead of invalidations. We also show how to generate correct sequences of memory accesses compatible with relaxed memory models. Such sequences must include not only loads and stores but also synchronization accesses. We show correctness properties which must be verified on the

hardware model in order to verify coherence.

Second, we demonstrate a successful application of SSM for the verification of the delayed protocol, an aggressive protocol for relaxed memory models. The verification is done at two levels: the behavioral and message passing levels. In both cases we show that SSM, which is based on abstraction technique preserving the properties to verify overcomes the state space explosion problem which is typical of current state-based approaches for protocols under relaxed memory models.

The paper is structured as follows. In the next section, we present an overview of the protocol used in this paper to demonstrate the methodology and is called the *delayed* protocol. In section 3, we establish a verification model for the protocol. An important element is an *execution model* for *data race free* (DRF) programs [1, 2]. This execution model ensures that the protocol state machine is driven by legal sequences of memory accesses (figure 1). In section 4, we develop the framework of the SSM method applicable to the protocol and we describe our verification tool in section 5. Sections 6 and 7 include the verification results and section 8 is the conclusion.

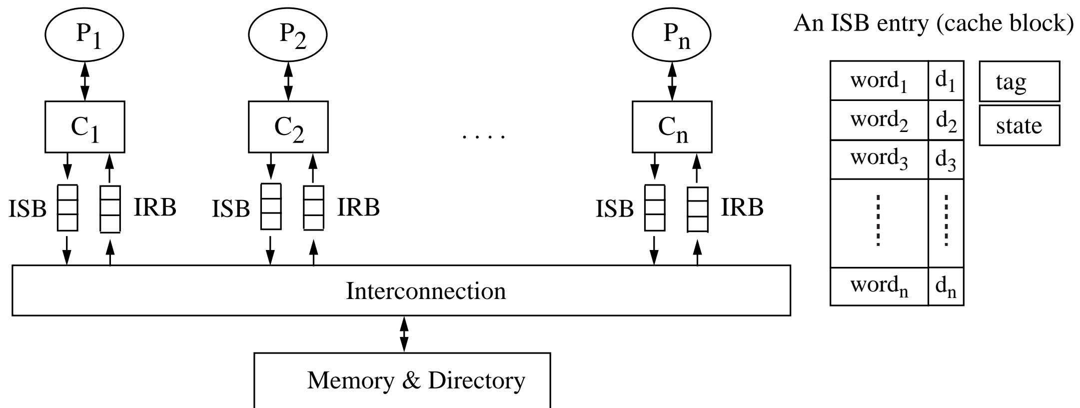

Figure 2. Illustration of System Architecture for Delayed Consistency.

2 The Delayed Protocol

The delayed protocol [10] is the target protocol to demonstrate the verification methodology proposed in this paper. The verification model includes features found in a wide spectrum of designs and is general enough to apply to other protocols under relaxed memory consistency models.

2.1 System Configuration

Figure 2 shows a simplified architecture model with a shared memory and private caches [6]. Every memory block has a directory entry which maintains the global state of the block and a full-map *presence* bit vector. The presence bit vector points to the processors with a cached copy. Every processor has a cache, an *Invalidation Send Buffer* (`ISB`) and an *Invalidation Receive Buffer* (`IRB`). The `ISB` keeps blocks that have been modified locally but are not owned locally and the modified words in each block are marked by dirty bits; it could be implemented as a write cache or as part of a lockup-free cache [12]. The `IRB` is a buffer for received invalidations[1]. Cached copies with pending invalidations in the `IRB` are considered *Stale* and are still accessible until the next lock[2] executed by the local processor and the updated blocks in the `ISB` are propagated to memory at the next unlock. In such a system, accesses to synchronization variables are treated differently that regular shared variables and the delayed protocol only applies to regular, non-synchronization data accesses.

2.2 Cache states and Algorithm

Every cache block can be in any one of four states: `Invalid`, `Keeper`, `Stale`, and `Owner` [10], with the following meanings: A `Keeper` copy is loaded into the cache on a read miss, a block copy is `Stale` when a pending invalidation is in the `IRB`, and an `Owner` copy implies that the local copy has been modified. Because invalidations are not performed on-the-fly, an `Owner` copy does not necessarily mean an *exclusive* copy in the system but the `Owner` is responsible for providing the copy in response to other processors' misses. Local modifications are buffered in the `ISB` even if the block is invalid in the cache, which means that a miss in the cache must check the `ISB` before issuing a request to memory. This leads to complex definitions for cache misses and hits in a processor node.

Definition 1 (Cache Misses and Hits) *Cache misses and hits are defined on the local cache state and the `ISB` state. For every access to address `d` in block `B`,*

---

1. Note that the IRB is not necessarily a hardware buffer, although we model it as such here. For example, it can be implemented with an additional *stale* bit in each block of the cache [10].

2. To simplify the discussion, we assume that lock and unlock are two primitive synchronization instructions recognized by the hardware. They can be used to build up more sophisticated synchronization libraries.

1. *A read misses when* `B` *is invalid in the cache and* `d` *is not valid in the local* `ISB`*,*
2. *A read hits when* `B` *is valid in the cache or* `d` *is valid in the local* `ISB`*,*
3. *A write misses when* `B` *is invalid in the cache and* `B` *is not in the local* `ISB`*, and*
4. *A write hits when* `B` *is valid in the cache or* `B` *is in the local* `ISB`*.*

Cached block *B* is valid if *B* is not in the `Invalid` state. A data word at address *d* in block *B* is valid in the `ISB` if *B resides* in the `ISB` (an `ISB` entry holds the tag of block *B*) and the dirty bit for *d* is set.

A brief description of the protocol is outlined below for completeness. Detailed explanations of coherence messages and transactions can be found in [10] and in [26]. The following description is from the perspective of local cache C_i. C_j is a generic cache other than C_i.

1. **Read hit**. No coherence action is taken.
2. **Read miss**. If C_j has a data copy in the `Owner` state, it must update memory with its data copy. The memory then sends a copy to C_i. If no `Owner` exists, memory directly supplies its copy to C_i. In both cases, C_i becomes a `Keeper`. If the block resides in C_i's local `ISB`, the valid words kept in the `ISB` entry are merged with the returned data block. The `ISB` block remains in the buffer.
3. **Write hit**. If C_i is the `Owner`, the write is performed to the cache only. (A cache line in the `Owner` state cannot be present in the `ISB`.) Otherwise, C_i must either have a valid copy (`Keeper` or `Stale`), or an `Invalid` copy with a valid `ISB` entry. An entry must be allocated in the `ISB` if an entry does not already exist. The value is always stored into the `ISB` block copy as well as in the cache copy if it is valid. The dirty bit of the modified word in the `ISB` block is set.
4. **Write miss**. C_i requests a copy with ownership from memory, which then sends invalidations to all caches with their presence bit set (these caches must be `Keepers` or `Owners`). When a cache C_j receives an invalidation, it sends its copy to memory if it is an `Owner` and an acknowledgement otherwise. The invalidation message received by C_j is buffered in the local `IRB` and C_j keeps an accessible `Stale` copy. The actual invalidation of C_j is delayed until the next lock_acquire. When all remote copies have been staled, C_i gets a copy from memory and becomes the new `Owner`.

5. **Lock (Remove IRB)**. Before a lock (lock_acquire), the invalidations buffered in C_i's local IRB must propagate to the local cache. In other words, all Stale copies become Invalid. This is also true when the IRB is flushed for other reasons.
6. **Unlock (Remove ISB)**. Before an unlock (lock_release), all entries in C_i's local ISB must be flushed. For each block in C_i's ISB, if the cache line is in the Keeper state, the memory merges its copy with the modified words in the ISB copy, stales all other copies in the Keeper state, and notifies C_i that it is the new Owner. Otherwise, if C_i's cached copy is Stale or Invalid, the memory is updated with the ISB content and the cache state does not change. This is also true when the ISB must be flushed for other reasons.
7. **Replacement**. A cache block must be replaced (victimized) when the cache needs room for data returned on a miss. If the victim is the Owner, memory is updated. If the victim is a Keeper, two cases are possible depending on the state of the ISB. If the block is not in the ISB, a request is sent to memory to clear the presence bit. Otherwise, the ISB copy is written back to memory which then merges the modified words with its copy and sends invalidations to all other Keepers. If the victim has an entry in the IRB (i.e., it is Stale), it becomes Invalid and the pending invalidation is removed from the IRB; moreover, if an ISB entry exists, memory is updated and all other Keepers are notified.

3 Verification of the Explicit Model

In this section, we describe the global state machine M used in the verification of the explicit model. We also specify the coherence conditions to verify on the model. In the next section we will show how to abstract this explicit model to form machine M_r used in the SSM.

The verification model for M is a finite state transition system, which in general can be defined as follows.

Definition 2 (Finite State Transition System) *With respect to a cache block, the behavior of a memory system with m local automata is modeled by a finite state transition system* M:(s_0, A, S, Σ, δ), where

s_0 is the initial state $\langle A \times A \times \ldots \times A \rangle_m$,

A is the set of state symbols,

S is the global state space (a subset of $[\langle A \times A \times \ldots \times A \rangle_m]$),

Σ is the set of operations causing state transitions, and

δ is the state transition function, $S \times \Sigma \rightarrow S$.

Given the general definition 2, we need to identify the elements of the model, in particular, the set of state symbols A in order to represent the system correctly. Also, the verification model must incorporate rules for '*correct*' programs.

3.1 Execution Model

Correct programs for systems with relaxed memory models must be DRF1 [2]. In DRF1, *conflicting* memory accesses must be explicitly synchronized. (Two memory accesses to the same address are deemed *conflicting* if at least one of them is a store.) In the verification model, the global state machine *M* must be steered by access sequences compatible with DRF1. To generate such sequences the execution model in the verification keeps track of each processor's execution *mode*. The mode of each processor is defined with respect to a data word (a data word is the smallest addressable unit of memory) and transitions between modes are specified by a state diagram for each processor and each data word, as shown in figure 3. There are three possible modes for each data word and each processor:

1. **Semi-Critical Section Mode** (`SCS`). In this mode, the processor may execute read accesses only. To support *concurrent readers*, processors in mode `SCS` do not prevent other processors in mode `SCS` from reading the shared data. A processor may enter the `SCS` mode provided no other processor is in the `CS` mode. A processor exits the `SCS` mode by executing an unlock.
2. **Critical Section Mode** (`CS`). A processor in mode `CS` is the only processor allowed to read or modify the shared data. When a processor is in mode `CS`, all other processors must be in the `OUT` mode. A processor may enter the `CS` mode provided no other processor is in the `SCS` or `CS` mode. A processor exits the `CS` mode by executing an unlock.
3. **Out Mode** (`OUT`). When a processor is in the `CS` mode, all other processors must be in the `OUT` mode and cannot access the shared data. Several processors may be in mode `SCS` while others in the `OUT` mode. Any read or write by a processor in the `OUT` mode must be preceded by a lock; if another processor is in mode `CS`, the lock cannot be executed before that processor exits

mode CS via an unlock.

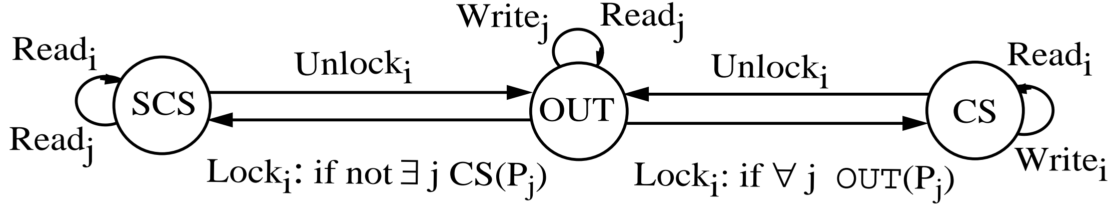

Figure 3. Processor Execution Modes from the Standpoint of Local Processor p_i

In [23], we proved that the above model correctly forces processors to issue sequences of memory accesses in conformance with DRF1 programs. The proof is quite trivial and we omit it here.

3.2 Modeling Data Words and Automaton States

Because coherence is a property bearing on each and every cache block taken separately, the verification procedure must deals with one memory block. Extending the model to multiple blocks is straightforward, but there is no compelling reasons to model multiple blocks at the protocol level[3]. Interferences between different memory blocks only occur on cache replacements. We can always model replacements by assuming that the block we track in the verification procedure may be removed from the cache at any time.

Because critical and semi-critical sections protect individual words and not cache blocks, the data block is split in two parts, wd_1 and wd_2. wd_1 is the word in the block which we track for data consistency. Concurrent accesses to wd_1 are restricted by the rules of DRF1. The rest of the block, wd_2, has an arbitrary size greater than or equal to one word. Modeling wd_2 is important because the state of wd_2 can affect the state of wd_1.

3.3 Overview of the Global State Machine *M*

To summarize, the local finite state automaton of each cache in the global state machine *M*

3. In this paper, we only consider functional errors of the cache protocol. Accesses to multiple memory blocks can cause additional implementation errors such as deadlocks by circularly holding resources, but it is out of the scope of this paper.

of the explicit state model is characterized by three components:

1. The cache state (c), where $c \in \{\mathtt{Owner}, \mathtt{Keeper}, \mathtt{Stale}, \mathtt{Invalid}\}$.
2. The Invalidation Send Buffer (isb) state[4], where isb $\in \{isb00, isb01, isb10, isb11\}$. Each of these four states specifies one of four possibilities: 1) no entry in the ISB, 2) wd_2 dirty in the ISB, 3) wd_1 dirty in the ISB, and 4) both wd_1 and wd_2 dirty in the ISB.
3. The processor mode (ps), where $ps \in \{\mathrm{OUT}, \mathrm{SCS}, \mathrm{CS}\}$. Mode CS indicates that the local processor is in the critical section and has the right to modify wd_1. Processors in SCS are allowed to read wd_1 and processors in OUT cannot access wd_1 at all.

The notation used to specify the automaton of each cache is c_{ps}^{isb}.

In addition to the above elements, we augment each cache automaton with auxiliary data variables (called the *cache data status*):

1. cwd_1 keeps track of the value of wd_1 in the cache copy.
2. $isbwd_1$ keeps track of the value of wd_1 in the ISB.

The cache data status takes values from the domain $\{nodata, fresh, obsolete\}$ and emulates the data part of the protocol semantics. When a processor modifies wd_1, its own cwd_1 becomes *fresh* while copies of cwd_1 in other caches and in memory become *obsolete*. Data transfers are emulated by assigning the value of the data status of some cache to the data status of other caches. The value of wd_2 is not modeled because the role of wd_2 is limited to modeling the changes in the states of wd_1 caused by accesses to wd_2.

The global state machine also includes the memory directory state denoted by m. A single global variable called the memory data status $mwd_1 \in \{fresh, obsolete\}$ holds the value of wd_1 at the memory. To avoid laying out all the details of the memory states in this paper, we refer the reader to [26] for a complete description of the protocol. The generic notation for a global state in an explicit system with n caches is:

---

4. The state of IRB is already embedded in the cache state. Remember that a stale copy indicates a pending invalidation in the IRB. Actually we advocate implementing the IRB with a stale state in the cache [10].

$$(c1^{isb1}_{ps1}[{}_{cwd_1, isbwd_1}], \ldots, cn^{isbn}_{psn}[{}_{cwd_n, isbwd_n}], m[{}_{mwd_1}]) \quad .$$

Two local cache automata are in the same state if they agree on their cache state, their ISB state, their processor state, and their data status. The state transitions are not functions of the status of the data.

Finally, the set of operations triggering state transitions in the global state machine are:

1. read and write accesses to wd_1 (wd_2),
2. *lock* and *unlock* synchronization accesses,
3. *remisb* (removal of the block from the `ISB`),
4. *remirb* (removal of the entry in the `IRB` and invalidation the block in the cache), and
5. *repl* (block replacement).

Read, write and synchronization accesses are restricted by the DRF1 model. All other operations including *remisb*, *remirb*, and *repl* can be executed at arbitrary times. *remisb and remirb* model a realistic implementation with `ISB` and `IRB` of finite capacities. Since they trigger the same actions as *lock* and *unlock*, they can also be used to model occurrences of synchronization accesses.

3.4 Model for Data Consistency and Detection of Inconsistency

To verify data consistency, the model keeps track of the values of data copies explicitly, as described in section 3.3. Data inconsistency is reported when a processor is allowed to read data with *obsolete* values.

Definition 3 (**Detection of Data Inconsistency**) *By tagging all data copies with values in the set* {*nodata*, *fresh*, *obsolete*} *and emulating data transfers*, *data inconsistency is detected when a processor is allowed to read data with obsolete values*.

Specifically, in the verification procedure, we check the following two conditions for every global state reached by *M* and for all processor p_i which can read wd_1 (i.e., p_i is either in the CS or in the SCS mode)

1. If p_i's cache copy is valid, then the copy must have the value *fresh*.

$$\forall i \quad ((ps_i = CS) \vee (ps_i = SCS)) \wedge (c_i \neq Invalid) \rightarrow cwd_{1i} = fresh$$

2. If p_i 's cache copy is invalid but a valid `ISB` entry exists, the ISB copy must be *fresh*.

$$\forall i \quad ((ps_i = CS) \vee (ps_i = SCS)) \wedge (C_i = Invalid) \wedge$$
$$((isb_{1i} = isb10) \vee (isb_{1i} = isb11)) \rightarrow isbwd_{1i} = fresh$$

The above two conditions are safety properties that must be true in all global states. The check of data consistency on wd_1 can be generalized to all other memory locations by symmetry [16]. Data consistency is checked only *when* a data copy can be accessed and therefore obsolete data copies can exist as long as they are not accessible.

4 Verification by Symbolic State Model (SSM)

The explicit model is now fully specified. However, as we will see, its verification is not feasible for systems of practical sizes. In order to make realistic protocols verifiable we must drastically reduce the size of the state space.

One possible approach to do this is based on system symmetry. Cache-coherent, shared-memory multiprocessor systems such as the SGI Challenge [13] are *symmetric*, which implies that contexts of processors can be swapped without affecting the correctness of the system. Given a protocol model of *n* processors, the symmetry property reduces the size of the state space by a factor of *n*!. However, for complex protocols, this reduction is not sufficient in general to avoid the state space explosion problem [24, 25].

To further reduce the size of the state space, we take a different approach. We observe that, in most existing cache protocols [3, 6, 10],

1. The behavior of every cache is specified by the same finite state machine (Therefore, caches in the same state move to the same next state in response to the same inputs.), and
2. When contending writes are posted, only one can progress at a time, but multiple concurrent reads are allowed.

The implication of the above is that the *exact* number of caches in the same state is irrelevant to protocol correctness. For example, whether there are 1, 5 or 200 caches in the shared and

clean state is not relevant. All these copies respond identically to memory events triggered elsewhere in the system. Similarly, we need to single out the cache which *owns* the cache block and is responsible for providing data to other processors which access the data and miss. Taking advantage of these observations, the SSM maps system states to more abstract states which do not keep track of the exact number of copies. In the balance of this section we show how this can be done successfully.

In general, we need first to identify an abstracted state transition system with the following properties.

4.1 Correspondence of State Transition Systems

Consider a state transition system M: (s_0, A, S, Σ, δ). Our goal is to find a more *abstract* state transition system M_r: (s_{0r}, A, S_r, Σ, δ_r) such that M_r *corresponds* to M and S_r is much smaller than S. Importantly, M_r must preserve the properties to verify. Error states of M must be mapped into error states in M_r in the context of the correctness properties to verify.

Formally we define this correspondence as follows.

Definition 4 **(Correspondence)** *Given two state transition systems* M: (s_0, A, S, Σ, δ) *and* M_r: (s_{0r}, A, S_r, Σ, δ_r), M_r *corresponds to* M *iff there exists a correspondence relation* φ *such that*:

1. s_{0r} *corresponds to* s_0 as denoted by $s_{0r}\varphi s_0$,
2. *For each* $s \in S$, *at least one state* $s_r \in S_r$ *corresponds to* s, i.e., $s_r\varphi s$.
3. *If* M *in state* s *makes a transition to state* t *on an enabled operation* τ, *and state* s_r *of* M_r *corresponds to state* s, *then there exists a state* t_r *such that* M_r *can move from* s_r *to* t_r *by* τ *and* t_r *corresponds to* t.
4. *With respect to the correctness properties,* M_r*'s states corresponding to error states in* M *must be error states.*

Figure 4 illustrates this correspondence relation. The problem remains to find the relation φ such that $M_r\varphi M$. Once this correspondence is established, we can prove that no protocol errors

will go undetected in the abstract model M_r. We now proceed to define our abstraction.

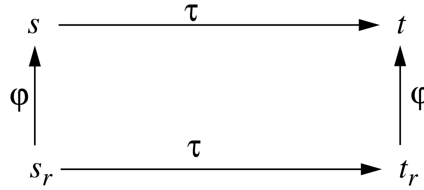

Figure 4. Correspondence Relation.

Our definition of correspondence relation between two automata is similar to the simulation relation based on possibility mapping in [19]. However, the fundamental proof tactic is different. In [19], the idea is to prove that an automaton *Imp* simulates another automaton *Spec* in the sense that every correctness conditions satisfied by the behavior of *Spec* is satisfied by the behavior of *Imp*. It is based on a concept of hierarchical refinement that the low-level description by an automaton *Imp* is a correct implementation of a high-level abstraction (specification) by an automaton *Spec*.

By contrast to hierarchical refinement of proving that one automaton is a correct implementation of another, we mean to reduce the size of the automaton on which the correctness conditions are evaluated. During the process, we convert the structure of an automaton to a corresponding automaton by notations with more powerful expressiveness.

We now present the set of repetition constructors which will enable us to group together sets of states to form superstates.

4.2 SSM Abstraction

4.2.1 Abstract State Representation

In SSM repetition constructors are used to represent global states [21-23]. In this paper we will use the following set of constructors:

Definition 5 (Repetition Constructors)

1. **Null** *(*0*) indicates zero instance.*
2. **Singleton** (1) *indicates one and only one instance, which can be omitted in the notation.*
3. **Star** (*) *indicates an unknown number of instances (i.e., greater than or equal to* 0*).*

We can use the above constructors to represent the global states concisely in a system with an unspecified number of caches. For instance, we can represent all the global states such that "one cache is in the Invalid state, and zero, one or multiple caches are in the Shared state" by (I, S^*). This representation includes a large set of explicit states.

We define the structure of global states (called composite states) in SSM as follows.

Definition 6 (Composite State) *A composite state is the composition of local automaton states in a system with an arbitrary number of caches. Suppose that Q denotes the local finite state machine. A global state has the form* $\left(q_1^{r_1}, q_2^{r_2}, \ldots, q_n^{r_n}\right)$*, where* $n = |Q|$ *is the number of all possible states of Q and* $r_i \in [0, 1, *]$ *is one of the repetition constructors of definition 5.*

For the verification of the delayed protocol, q_i in definition 6 includes all the elements of the state given in section 3.3, i.e. q_i is a particular instance of c_{ps}^{isb} or the memory directory state, including the data status as well.

Composite state containment is an important characteristic of SSM, which leads to a large reduction in the size of the state space expansion.

4.2.2 State Containment Relation

The repetition constructors in SSM are ordered according to the set of states they represent. The resulting orders are 1 < * and 0 < *. This leads to the definition of *state containment*.

Definition 7 (Containment) *Composite state* S_2 *contains composite state* S_1*, or* $S_1 \subseteq S_2$*, if*

$$\forall q^{r_1} \in S_1 \quad \exists q^{r_2} \in S_2 \quad \text{such that } q^{r_1} \le q^{r_2} \text{ i.e.} r_1 \le r_2$$

where $r_1, r_2 \in [0, 1, *]$.

In section 4.3, we will prove that the SSM yields a *monotonic containment relation* such that, if $S_1 \subseteq S_2$, then $S_1 \to S_1' \Rightarrow (S_2 \to S_2') \wedge (S_1' \subseteq S_2')$. We will also show that the abstract state machine M_r based on SSM is efficient and accurate for detecting protocol errors.

Since SSM directly expands composite states instead of explicit states, transitions triggered on composite states are now specified.

4.2.3 State Expansion Rules and Algorithm

Consider the general form of a composite state $(q_1^{r_1}, q_2^{r_2}, \ldots, q_n^{r_n})$. The set of operators applicable to composite states during the state generation process is defined as follows, where '/' signifies "or" selection, '$\rightarrow$' means a transition and Q is a generic term grouping local state machines of no direct interest.

1. **Aggregation**: $(Q, q^0, q^r) \equiv (Q, q^r)$, $(Q, q^1, q^{1/*}) \equiv (Q, q^*)$,and $(Q, q^*, q^*) \equiv (Q, q^*)$.
2. **Coincident Transition**: $q_1{}^r \rightarrow^\tau q_2{}^r$, where $r \in [1, *]$. All automata in the same state will move to the same next state in response to the same inputs.
3. **One-step Transition**:

   (a) $(Q, q_1) \rightarrow^\tau (Q', q_2)$ and

   (b) $(Q, q_1{}^*) \rightarrow^\tau (Q', q_2, q_3{}^*)$,

   where all machines not in state q_1 are denoted by Q in the tuple, $\rightarrow^\tau$ is a transition applied to one of the component machines in state q_1 such that $q_1 \rightarrow^\tau q_2$, and τ causes all other state machines in state q_1 to move to state q_3. After the transition, some state machines in Q may be affected as shown by the change from Q to Q'.
4. **N-steps Transitions**: This rule specifies the repetitive application of the same transition N times, where N is an arbitrary positive integer and Q represents other machines which are not in state q_1.

   (a) $(Q, q_1) \rightarrow^\tau (Q, q_2, q_1{}^0)$, and

   (b) $(Q, q_1{}^*) \rightarrow^\tau (Q, q_2{}^*, q_1{}^*)$.
5. **Progress Transitions**:

   (a) $(Q_1, Q_2{}^*) \rightarrow^\tau (Q_1', Q_2'^*)$ and

   (b) $(Q_1, Q_2{}^*) \rightarrow^\tau (Q_1')$.

   These two rules deal with transitions $\rightarrow^\tau$ leading to different protocol behaviors on empty and

non-empty Q_2. Because Q_2 represents an unknown number of automata in some states and the constructor * may include the null case, we generate two states corresponding to empty and non-empty Q_2 [25].

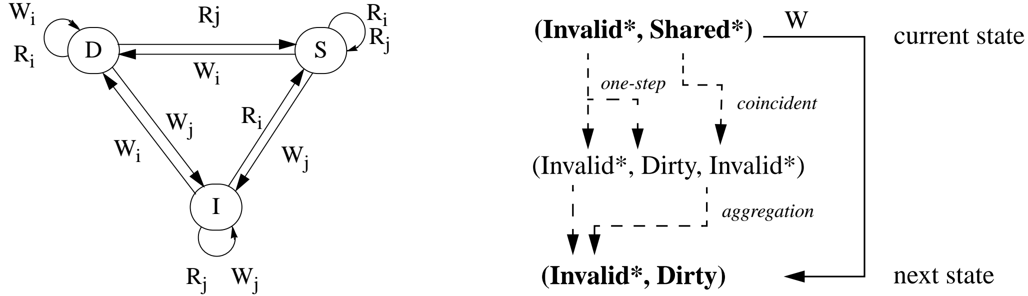

Figure 5. Illustration of State Transitions in SSM.

A state expansion step has two phases. During the first phase, a new composite state is produced by applying transition rules to the current state. In the second phase, the *aggregation* rule is applied to lump together local state machines in the same state. For illustration, consider the example of figure 5. On the left is the finite state machine for a three-state protocol [3] from the perspective of cache *i*. *j* denotes any other cache. R_i and W_i stand for read and write initiated by cache *i*. On the right is the state transition from the current state (*Invalid**, *Shared**) to the next state (*Invalid**, *Dirty*) caused by a write miss. As shown in the example, the aggregation rule is a post-transition rule to keep the representation for composite states concise.

Whereas, the SSM expansion rules manipulate the repetition constructors in the composite state, the state transitions, or more precisely, the ways in which the local finite state machines react to an input, are still dictated by the cache coherence protocol. For example, the transition from (*Invalid**, *Shared**) to (*Invalid**, *Dirty*) is labeled by a write miss. Thus the SSM does not alter the semantics of the protocol.

The example of figure 5 also demonstrates the application of the one-step rule. If we trigger another write-miss transition, the cache in the dirty state becomes invalidated and a new dirty copy is generated. Although the new state is still (*Invalid**, *Dirty*), the semantic is different. In the transition from (*Invalid**, *Shared**) to (*Invalid**, *Dirty*), the missing block is provided by memory

(a clean copy is supplied by memory [3].) By contrast in the write-miss transition from (*Invalid**, *Dirty)* the block is supplied by the current cache in the dirty state.

Generally speaking, the one-step transition rule is sufficient to carry on the expansion. Nevertheless, to speed-up convergence in a series of transitions, we have introduced the N-step rule. An *N-steps transition* consists of a chain of one-step transitions:

$$(Q, q_1^*) \rightarrow^{\tau} (Q, q_2^1, q_1^*) \rightarrow^{\tau} (Q, q_2^2, q_1^*) \rightarrow^{\tau} \ldots \rightarrow^{\tau} (Q, q_2^*, q_1^*).$$

The same transition $q_1 \rightarrow^{\tau} q_2$ can be applied an unlimited number of times as long as there are state machines in state q_1. Every application of the transition brings down the number of base machines in state q_1 by one and increases the number of base machines in state q_2. Note that the transition $\rightarrow^{\tau}$ must have no effect on other machines (included in Q in the tuple). A typical application of this rule is when processors replace their copy in shared state, or when processors post requests independently. Applications of the N-step rules do not necessarily mean that the intermediate states are correct. In each of these intermediate states the property to verify must still be checked.

The progress rules are applied in the case of write-invalidations. In write-invalidate protocols, a write may cause invalidations to propagate to all caches sharing the block. The write transaction is complete when all processors have received their invalidation and have acknowledged the home directory. Because the SSM abstracts the number of caches in a particular state, two states are generated, one in which the write transaction is in progress and one in which it is completed. Note that the progress rule is an extension to our original work [21] and is only needed when memory accesses are modeled as non-atomic operations [25].

4.3 Monotonicity

Given the framework of SSM, we can prove that the expansion rules are monotonic operators, i.e., if $S_1 \subseteq S_2$, then $S_1 \rightarrow S_1' \Rightarrow (S_2 \rightarrow S_2') \wedge (S_1' \subseteq S_2')$. Practically, a composite state in SSM represents a set of global states in an explicit state enumeration model. Therefore intuitively, if an SSM state S_1 is contained by S_2 and if an expansion step is done correctly, then the next states of all the states included in S_1 must also be contained in the next states of all the states in S_2.

In general, let's consider a system composed of finite state machines such that one machine can communicate with all other machines directly. A composite state of any SSM for that system has the form

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

Lemma 1. *The aggregation process is monotonic, that is, if* $q^{r_{11}} \le q^{r_{21}}$ *and* $q^{r_{12}} \le q^{r_{22}}$*, then we have* $Aggregation(q^{r_{11}}, q^{r_{12}}) = q^{r_1} \le Aggregation(q^{r_{21}}, q^{r_{22}}) = q^{r_2}$.

Proof: The proof follows from the ordering relation among the repetition constructors and from checking all possible combinations of r_{11}, r_{12}, r_{21} and r_{22} subject to the constraints of this lemma and to the aggregation rule. ❑

We extend this monotonic aggregation process to state transitions.

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

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

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

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

if $1 \le \bar{r}_i$, $r_j = \bar{r}_j$ *for all* $j \ne i$*, and the same expansion rule taken on the same memory event* τ *is applied to* S_1 *and* S_2.

Proof: We only need to consider the effect of applying τ to machines in state q_i in S_1 and S_2. To simplify the notation, all classes q_j ($j \ne i$) are lumped in Q. Provided $q_i \rightarrow^{\tau} q_k$, the following two states are generated when a one-step transition rule is applied to S_1 and S_2.

(1). $S_1=(Q, q_i) \rightarrow^{\tau} \overline{S}_1=(Q', q_i^0, q_k)$,

(2). $S_2=(Q,\ q_i^{\bar{r}_i}) \rightarrow^{\tau} \bar{S}_2=(Q',\ q_i^{\bar{r}_i'},\ q_k)$

Q' means that the transition may cause state changes of other machines (coincident transition). Since $q_i^{\bar{r}_i}$ includes the case of a single machine, $q_i^{\bar{r}_i'}$ must contain the case of zero machine. It is clear that $\bar{S}_1 \subseteq \bar{S}_2$. This containment relation is also true when compound rules involving multiple one-step transitions such as the N-steps rules are applied to S_1 and S_2. ❑

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

Proof: Because the aggregation process is monotonic by lemma 1, lemma 3 simply extends the result of lemma 2. ❑

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* $\bar{S}_1 \subseteq \bar{S}_2$.

Proof: This is an immediate result of lemma 3. ❑

Because of the property of monotonicity, we only need to keep track of composite states which are not contained by any other state. The final output is a set of *essential states*.

Definition 8 (Essential State) *Composite state S is essential if and only if there does not exist a distinct composite state* $\bar{S}$ *such that* $S \subseteq \bar{S}$.

It should be pointed out that the generation of essential states terminates as soon as any logical protocol error is detected, since expanding error states, which lead to unpredictable states, is practically meaningless. At the end of a successful expansion process, the (explicit) state space is partitioned into several families of states (which may be overlapping), each represented by an essential composite state.

4.4 Correspondence Between State Enumeration and SSM Model

Now that we have shown that the SSM expansion is monotonic, we need to prove that a correspondence relation exists between the abstract SSM state transition system M_r: $(s_{0r}, A, S_r, \Sigma, \delta_r)$ and the explicit state transition system M:$(s_0, A, S, \Sigma, \delta)$ with respect to the property of data

consistency. For this purpose, we first define the correspondence relation φ.

Definition 9 (Correspondence Relation) *State* $s_r:(q_1^{r1}, q_2^{r2}, \ldots, q_n^{rn}) \in S_r$ *corresponds to state* $s:(a_1, a_2, \ldots, a_m) \in S$, *i.e.* $s_r \varphi s$, *if s is one of the states abstractly represented by* s_r, *where* a_i *is the state of the local automaton i and* $a_{i:1..m}, q_{j:1..n} \in A$. *The number of local automata of s in state* q_j *(i.e.* $|a_{i:1..m} = q_j|$*) must be a case covered by the repetition constructor* r_j, *namely,* $\forall j \ |a_{i:1..m} = q_j| \le r_j$.

For example, (S^*, I) corresponds to (S,S,I) according to definition 9.

Theorem 2. *Given* s_0, *we can easily construct* s_{0r} *such that* $s_{0r} \varphi s_0$.

For instance, it is normal to start the verification with an initial state in which no cached copy exist. In this case, all caches are invalid and $(Invalid^*)\varphi(Invalid, Invalid, \ldots, Invalid)$.

Finally, we prove that the correspondence relation illustrated in figure 4 exists for all possible transitions. Given $s_r \varphi s$ and $\delta(s, op) = t$, we show that $\delta_r(s_r, op) = t_r$ and $t_r \varphi t$, where *op* is an operation in Σ.

Theorem 3. *Consider the state transition system* $M:(s_0, A, S, \Sigma, \delta)$ *of an explicit model with (unrestricted, arbitrary) m local automata and the abstract state transition model* $M_r: (s_{0r}, A, S_r, \Sigma, \delta_r)$ *in the SSM. Consider two states* $s:(a_1, a_2, \ldots, a_m) \in S$ *and* $s_r:(q_1^{r1}, q_2^{r2}, \ldots, q_n^{rn}) \in S_r$, *where* a_i *is the state of the local automaton i and* $a_{i:1..m}, q_{j:1..n} \in A$. *Given* $s_r \varphi s$ and $\delta(s, op) = t$, *we can derive* $\delta_r(s_r, op)$ *such that* $\delta_r(s_r, op) = t_r$ *and* $t_r \varphi t$.

Proof: Because *s* is one of the states represented by s_r ($s_r \varphi s$), the monotonic operation of SSM guarantees that *t* is a state characterized by t_r. ❑

Theorem 4. *Suppose that the abstract state transition model* $M_r: (s_{0r}, A, S_r, \Sigma, \delta_r)$ *in the SSM corresponds to the explicit state transition system* $M:(s_0, A, S, \Sigma, \delta)$. *For all states, if* $s_r \varphi s$ *and if s is a state which violates the data consistency property,* s_r *is also a state which violates the same data consistency property.*

Proof: According to the definition of data consistency (section 3.4), it is a safety property for all global states. If a processor in s can read an obsolete value of wd_1, some processors in s_r must read obsolete values of wd_1 because s is one of the states covered by s_r.

4.5 Discussion

In this section we want to address a few issues that have been raised over time on the accuracy of the SSM abstraction for detecting data inconsistencies. The SSM method belongs to the class of methods using abstraction [5, 7, 8]; it maps a concrete finite state automaton (M) to a more abstract and small state automaton (M_r). It was shown that abstraction mappings are normally *conservative* [7]. They preserve invariant properties, but might report false errors because the abstract models could cover behavior not possible in the original machine. In particular, properties such as liveness formulated with the existential path quantifier may not be preserved.

Given these known shortcomings, a particular abstraction mapping must be powerful and accurate enough to verify important, global properties such as data consistency. Theorem 4 shows that the SSM abstraction does preserve the global invariant property of consistency. Verification based on M_r is at least as accurate as verification runs based on M for the detection of data inconsistencies. Importantly, since M_r is independent of the system size, the verification of M_r could discover protocol errors that may go undetected in M with a small number of processors [22].

The question remains whether the SSM abstraction developed in this paper may report false errors, or whether it is an *exact approximation* of the concrete model [7]. In [17], Ip and Dill argue that our SSM abstraction may report false errors. For example, an explicit state (I, S, S) with one invalid and two shared copies is covered by (I, S^*) in SSM; however, (I, S^*) also covers states with more than two shared copies. As a result, false errors may be reported because an SSM metastate covers many explicit states, including states which may not exist in a (finite) explicit model.

The answer to this argument is that the SSM abstraction cannot check properties depending on the exact number of processors. SSM demonstrates the correctness of a cache protocol for any number of caches in the system. It is conceivable that a flaw in a protocol never manifests itself in systems with up to N processors. Thus, if one wants to build machines with less than N processors, the verification based on SSM may over-constrain the design of the protocol. On the other hand,

depending on the value of N, it may be impossible to verify the protocol with the explicit model because of the state space explosion. Furthermore, it may be unwise to design protocols that only work for small numbers of processors.

In the most generic form, one can imagine an explicit model with a starting state such as $(q_1^{0\ldots\infty})$. The number of exact instances of machines in state q_1 is in the range of 0 to infinity. Without loss of generality, the case of null instance reflects a system without any cache. Allowing a cache protocol to operate in a system without any cache is practically meaningless, but harmless. Accordingly, the corresponding SSM meta-state is (q_1^*). Consider a transition $q_1 \to q_2$. The resulting states are $(q_1^{0\ldots\infty}, q_2^{0/1})$ and $(q_1^*, q_2^{0/1})$ respectively. ("/" means "or" selection.) There is still an infinite number of machines in state q_1 even if one cache change state to q_2 and, if the system had no cache, then there are no machine in state q_2 after the transition. The SSM meta-states cover all the explicit states. Note that $(q_1^{0\ldots\infty}, q_2^0)$ is essentially the initial state and (q_1^*, q_2^0) is the same as (q_1^*) in SSM. Suppose that a chain of the same transition (N-step transition rule) can be applied, according to the semantics of the protocol. Two corresponding sequences of states are:

$$(q_1^{0\ldots\infty}, q_2^{0/1}) \to (q_1^{0\ldots\infty}, q_2^{0/2}) \to \ldots \to (q_1^{0\ldots\infty}, q_2^{0\ldots\infty}), \text{ and}$$

$$(q_1^*, q_2^{0/1}) \to (q_1^*, q_2^{0/2}) \to \ldots \to (q_1^*, q_2^*).$$

It is clear that the final SSM meta-state catches all the possible explicit states along the path. Readers should nevertheless notice that grouping of several caches together should conform to the condition defined in [17], namely that transitions from the new state are independent of the exact number of machines grouped in a particular state. In the example above, if the protocol behavior is different when one or multiple instances of q_2 exist, the above chain of transitions must be disallowed. Logically, the * constructor characterizing the number of copies of q_2 is explicitly broken into several less abstract constructors, covering 0, 1, or multiple.

The MOSEI protocol [27] is a classic example. The number of data copies in the "shared"

state determines whether a data block returned on a load miss is loaded in the “exclusive” state or in the “shared” state. Therefore, the SSM model must distinguish the case of null copy in the “shared” state from the case of at least one or multiple. Other examples include the Illinois protocol, the Firefly protocol [3, 21] and the S3.mp protocol [22] which all have transitions sensitive to the number of caches in a particular state.

Applying the progress rule requires special attention as well. Essentially, the progress rule captures the protocol behavior for the propagation of invalidations (To simplify this discussion we limit ourselves to write-invalidate protocols, but the same argument would apply to updates in a write-update protocol.) Before a write can successfully complete, all other cached copies must be invalidated. The progress status of a write depends on whether there are cached copies.

Without loss of generality, consider a meta-state (q_1^*, q_2^*). Suppose that transitions $q_1 \rightarrow q_3$ or $q_1 \rightarrow q_4$ may be applied depending on whether the number of instance of q_2 is null and that q_2 always branches to q_5 (This would be the case for a write miss in a cache in state q_1 issuing a write miss request. In this case of a write miss, machines in state q_2 would be the ones subject to invalidations.) The two resulting states are (q_1^*, q_3) and (q_1^*, q_4, q_5^*). At first glance, it would seem that the state (q_1^*, q_4, q_5^*) should be denoted as (q_1^*, q_4, q_5^{*-0}), which excludes the null case for q_5, because the path being taken assumes a non-empty class of q_2. However, for protocols such as the delayed protocol and many other existing protocols, all the caches to be invalidated behave the same way by acknowledging their invalidation. As long as a group of caches must be invalidated, we really do not care about their exact number. As a result, we avoid complicating the model by excluding the null case.

To emphasize, the SSM abstraction does not alter the state transition functions. State transitions are always defined by the cache protocol semantics rather than by the SSM abstraction. *The SSM abstraction is just a concise representation* for the state space of a system composed of many identical finite state processes and it never collapses states in a way that could eliminate or modify protocol transitions.

Finally, in our previous work [21, 25] we introduced an additional constructor called Plus (+), indicating one or multiple instances. The starting state is (q_1^+) corresponding to explicit states $(q_1^{1 \ldots \infty})$. Operations are not performed on the null case. Since + is covered by * and operating on null is harmless, the accuracy of SSM in detecting data inconsistency errors is not affected by dropping +. In the following, we will refer to SSM schemes with or without the plus constructor as SSM-+ and SSM-*. Also in sections 6.2 and 7.1, we will show that a significant reduction of state space is achieved by the elimination of the + constructor.

5 The SSM Verification System

We have developed an automated verification system based on the SSM [25]. The system consists of a high-level description language and an automated verifier. In the SSM system, users describe their protocols at the level of finite state machines. Global states are composed of *state variables* defined by the users. State transitions among global states are specified by a set of transition rules. Each rule consists of guarded statements: it is associated with an enabling condition and it is applicable only when its enabling condition is true. The language is block structured and supports a rich subset of statements found in common programming languages, including basic assignments, if-then-else conditionals, switch-case selections, for-loops and procedure calls.

Figure 6 shows the state declarations for the delayed protocol under atomic memory accesses. The state model was developed in section 3.2. The processor environment represented by the structure *Proc* is the basis of abstraction. Figure 6 also shows that transition rules applicable to all *Procs* are collected in a "*RuleSet*". We show the example of the rule for reading word *wd*1 when a processor is either in the *SCS* or in the *CS* mode. When the cache copy is valid or when the data is present in the *ISB* buffer, the copy must be "*fresh*". Otherwise, a violation of the assertion will cause the SSM system to report an error and start backtracking. The actions taken on the read miss are modeled by changing the values of state variables. Note that in its current form, the tool requires the users to provide information about which expansion rule (e.g., one-step or Nsteps)

should be taken.

```
Type
    ValStatus: enum {nodata, fresh, obsolete};
    ProcAccMode: {OUT, SCS, CS};
    GlobalAccMode: {GLB_OUT, GLB_SCS, GLB_CS};              -- for modeling convenience
    CacheSt: enum {INV, KEEPER, OWNER, STALE};              -- IRB state is embedded in STALE
    ISBSt: enum {ISB00, ISB01, ISB10, ISB11};               -- refer to section 3.2

    Proc:   Record
            cs: CacheSt;                                    -- cache state
            wd1:ValStatus;                                  -- status of the cached copy
            acc_mode:ProcAccMode;                           -- processor access mode
            isb_st:ISBSt;                                   -- state of the ISB buffer
            isb_wd1:ValStatus;                              -- status of wd1 in ISB buffer
    End;

    Memory: Record
            wd1: ValStatus;                                 -- status of the memory copy
            global_access_mode: GlobalAccMode;              -- global access mode to wd1
            :::
    End;

    ProtocolMachine: Record                                 -- define the protocol machine
            bm: (repetition) Proc;                          -- define a repetition base
            mem: Memory;                                    -- home memory/directory
    End;

    GlobState gs: ProtocolMachine;                          -- actually define the global state variable
RuleSet p: (repetition) Proc Do

    Rule "read access to WD1"
    (p.acc_mode==SCS || p.acc_mode==CS)                     -- p cannot read WD1 in OUT mode
    Begin
        ExpansionClass p NSteps;                            -- which rule?
        Switch (p.cs)
            Case KEEPER, OWNER, STALE:                      -- in-line assertion
                        assert (p.wd1==fresh) "read obsolete value";
            Case INV: Switch (p.isb_st)
                    Case ISB00: // also read miss in ISB
                        p.cs=KEEPER; // take actions to load the data block
                        ::::
                    Case ::::
        End
    End -- Rule
    :::::
    Rule ::::::
End -- RuleSet
```

Figure 6. Example Code in the SSM System.

The protocol description in the SSM language is not directly executable. The description is first translated into a verifier written in C, which is then compiled to generate an executable image.

The verifier uses the symbolic state expansion algorithm to explore the state space of the protocol exhaustively.

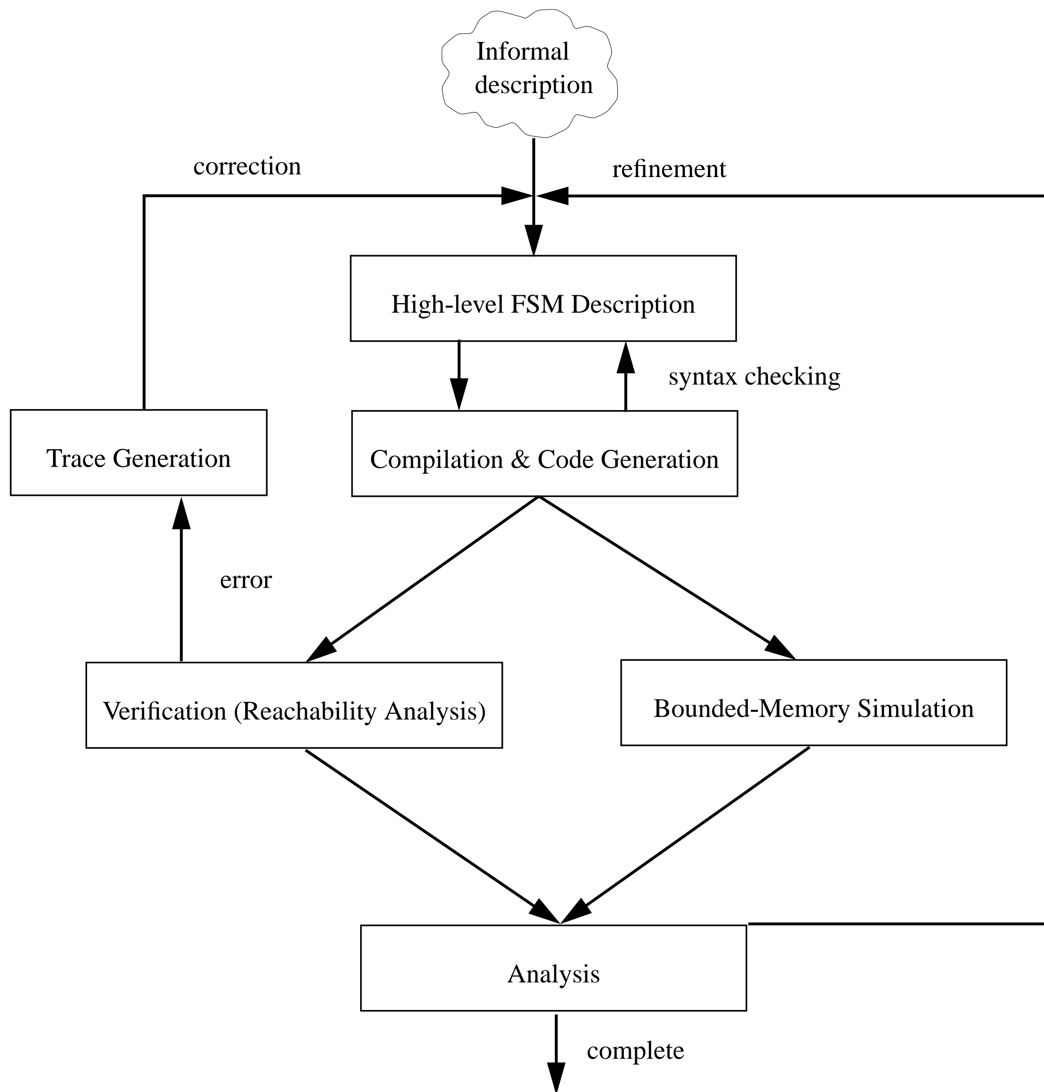

Figure 7. SSM Environment.

In its current status, the SSM verifier checks *safety* properties specified by the user. These properties must be respected in every state. The user normally inserts in-line *assertion* statements in the description program as shown in figure 6. When an assertion is not satisfied, the current state is reported as an *erroneous* state. In addition to in-line assertions, the users have all the flexibility to add sophisticated procedures for error detection. For instance, one can examine every newly generated state by calling procedures for checking errors at the end of state transitions. One such procedure for the delayed protocol model is to check if a processor can access obsolete data (sec-

tion 3.4).

```
Procedure InvariantCheck;
Var q:(pointer) ProcState;
Begin
 For q:(repetition) ProcState Do
 If (q.p_blocked == NoPending) Then
   Switch (q.c_st)
    Case INV:
        If (q.acc_mode==SCS || q.acc_mode==CS) Then -- if the processor can access wd1
        Switch (q.isb_st)
          Case ISB10,
               ISB11:
              Assert (q.isb_wd1 == FRESH) "Can read stale ISB-WD1";
        End;

        End;
    Case STALE:
        If (q.acc_mode==SCS || q.acc_mode==CS) Then -- if the process can access wd1
          Assert (q.wd1_val == FRESH) "Can read stale WD1";
        End;

    Case KEEPER,OWNER:
        If (q.acc_mode==SCS || q.acc_mode==CS) Then -- if the process can access wd1
          Assert (q.wd1_val == FRESH) "Read stale WD1";
        End;
   End; -- Switch
 End; -- If
 End;
End;
```

Figure 8. An Example Procedure for Checking Data Consistency.

Figure 8 shows the procedure for checking data consistency. After a new state is generated in the generic form $(q_1^{r1}, q_2^{r2}, \ldots, q_n^{rn})$, the procedure is called. The variable q is a control variable which walks through all q_i in the next state. The code checks that if a processor has the access right to an obsolete data copy, the protocol fails. The users can define any procedure to check the values of state variables, including the constructor carried by q_i.

At the occurrence of an error, the verifier terminates and reports a trace leading from the initial state to the error state. The verification procedure with the SSM system is illustrated by the flow diagram in figure 7.

Because the SSM abstraction does not maintain the identity of processes, it cannot check properties referred to specific processes. For instance, queries of finite behavior such as "*if* p_1 *is in state A, there exist a global state in which* p_2 *is in state B in the future*" can not generally be

answered. However, we feel this type of query is usually not interesting except for checking for the absence of deadlocks and livelocks. In [25], we adopt a simple but useful definition for deadlocks and livelocks in the context of cache protocols. In brief, a memory access should eventually be satisfied. In existing cache protocols, it means that a "shared" copy is always procured after a read miss, and a "dirty" copy after a write miss. The checking procedure work as follows. Assume that the global state graph is built on top of the final set of essential states and the starting state is (Invalid*). We logically *tag* one of the caches in the Invalid class of automata, and traverse through the state graph following the transitions. In it current status, the tool does not check the livelock case as described, but it checks simple deadlock states, which are states with no successors other than themselves.

The SSM system verifies protocols completely by exploring all reachable states and always terminates. However, the SSM system also allows users to run the system in *bounded-memory simulation* mode. The major difference between verification and simulation is that the verifier never terminates in the simulation mode: The state expansion paths are randomly selected, and hence, there is no guarantee that all reachable states are generated. The simulation mode can be very useful when the available memory is small and the verification cannot be completed due to limited memory size. The idea behind bounded-memory simulation is to approximate the quality of a complete verification by maximizing the utilization of available memory. When the SSM system is executed in this mode, users must specify the maximum memory size which can be used to store state information. The SSM verifier initially runs as usual as a verification tool. When the SSM system consumes memory in excess of the available memory, some states stored in the history list or the list of unexpanded states are removed in order to meet the memory constraint.

6 Verification at the Behavioral Level

In this section, we present the results of applying the SSM system to verify the delayed protocol. We first verify the delayed protocol at the behavior level by assuming *atomic* memory accesses, namely, protocol transitions happen instantaneously in zero time. After showing that the delayed protocol has correct behavior, we verify a design of the delayed protocol at the more detailed, message-passing level for systems with non-ordered interconnections. In this case, memory accesses are *non-atomic* and cache coherence is achieved by exchanging messages among pro-

cessors.

To evaluate the efficiency of the SSM method, we have also applied the Stanford Murφ system [16] to the delayed protocol. The Murφ is an efficient verification tool based on state enumeration, incorporating state encoding to reduce memory usage, and hash tables to speed up the search and comparison steps. There are two Murφ systems: one exploits the system symmetry (Murφ-s) and one does not (Murφ-ns). The comparison between SSM and Murφ can assess the performance advantages afforded by the drastic reduction of the global state space in the symbolic state mode.

For all verification runs, we start the expansion process in an initial state in which no cache has a block copy and all processors are in the `OUT` mode, prevented from accessing wd_1. Thus, the initial state is $(Invalid_{out}^{isb00+})$ in SSM-+ and $(Invalid_{out}^{isb00\,*})$ in SSM-*.

6.1 Behavioral Correctness of the Delayed Protocol

Before reaching the conclusion that the delayed protocol presented in [10] is correct, several errors were found. These errors arise from ambiguities in the informal protocol description [10]. One major oversight leading to confusion is described in the following scenario.

1. Initially, cache C_1 is in the `Stale` state and cache C_2 is the `Owner`. Moreover, C_1 is in the critical section for *wd1* and an updated copy of *wd1* is in $\mathtt{ISB}_1$.
2. When C_1 exits its critical section, the value of *wd1* in $\mathtt{ISB}_1$ is sent to memory, which is updated. Subsequently, an invalidation is sent to C_2.
3. When C_2 receives the invalidation, it writes its copy back to memory.
4. Upon receiving the write-back message, memory is updated. Thus, the most recent value of *wd1* from C_1 is lost.

This error occurred because the literal description of the delayed protocol in [10] does not give a clear indication of the correct sequence that should be taken to update the memory by write-backs from C_1 and C_2. An incorrect protocol model was then built and verified to reveal the problem. This experience has convinced us of the importance of formal verification methods which uses formal specification rather than linguistic form to specify protocols.

After fixing such errors, the delayed protocol was proven to be correct by exploring the state space exhaustively. With respect to wd_1 and processor P_i *in* state $c_{i_{ps_i}}^{isb_i}$, the delayed protocol has the following properties:

1. $((ps_i = cs) \vee (ps_i = scs)) \wedge (c_i \neq Invalid) \rightarrow (cwd_{1i} = fresh)$. This property partially proves that the delayed protocol preserves data consistency. If processor P_i can access its local copy of wd_1, the copy must have a fresh value.
2. When there is a modified copy stored in the local ISB, all other copies are obsolete. We have

$$(isb_i = isb10) \vee (isb_i = isb11) \rightarrow$$
$$(isbwd_{1i} = fresh) \wedge (mwd_1 = obsolete) \wedge$$
$$(\neg \exists j \neq i, (cwd_{1j} = fresh) \vee (isbwd_{1j} = fresh))$$

   This property confirms that the delayed protocol ensures data consistency. Since a read access may miss in the local cache and hit in the ISB, the word present in the ISB copy must have the most recent value. The new value stored in the ISB is not yet visible to other processors and therefore, multiple copies of the same memory location in different ISBs are not allowed.
3. $(c_i = Owner) \rightarrow (isb_i = isb00)$. This invariant states that if the local cache is the Owner, there must be no corresponding entry in the ISB. This property shows an optimization of the protocol to avoid coherence overhead on private data. Updated private data is written back to memory only when the block is replaced.
4. $(c_i = Owner) \rightarrow \neg \exists j \neq i, (c_j = Owner) \vee (c_j = Keeper)$. This means that, if there is an Owner, no other caches may be an Owner or a Keeper. This property must be respected because, when a cache procures the ownership of the protocol, the protocol stales all other cached copies.
5. The final property we proved shows that a correct path leading to a fresh data copy always exists:

$$((ps_i = scs) \vee (ps_i = cs)) \wedge (c_i = Invalid) \wedge$$
$$\neg((isb_i = isb10) \vee (isb_i = isb11)) \rightarrow$$
$$(\exists j \neq i\ c_j = Owner \rightarrow cwd_{1j} = fresh) \vee ((\neg \exists j \neq i\ c_j = Owner) \rightarrow mwd_{1j} = fresh)$$

The above expression means that, if wd_1 is accessible by processor P_i, *and if neither* C_i nor the local `ISB` have a copy, P_i can get the correct data from the memory or the remote `Owner`. Since the memory controller requests the `Owner` (if any) to write back its copy on a cache miss, the `Owner` must have a fresh copy to avoid updating the memory with obsolete data.

6.2 Performance Results

Table 1 shows the performance of the four tools for the verification of the delayed protocol on a SPARCstation 10 Model 30 with 128 MBytes of memory. The state space of the delayed protocol quickly increases with the model sizes, even under the assumption of atomic memory accesses. Using the same assumption of atomic memory accesses, the delayed consistency has a much higher level of complexity as compared with the verification of other protocols [21]. The number of essential states reported by SSM-+ is 58, which is much larger than the number of essential states (usually under 8, cf. [21]) in traditional protocol designs, when atomic memory accesses are assumed.

TABLE 1. Performance Results of Different Methods for Verifying the Delayed Protocol (atomic transactions).

Method	Number of processors	Size of global state space	Size of search state space	Verification time (seconds)	Memory (Mbytes)
Murφ-ns	2	484	4,184	0.9	0.007
	3	6,228	76,170	14.7	0.1
	4	75,088	1,202,544	268.2	1.15
	5	905,312	18,132,640	3,303.1	17.35
Murφ-s	2	248	2,148	0.8	0.004
	3	1,206	14,844	14.9	0.02
	4	4,500	72,575	330.1	0.07
	5	14,366	288,932	7,949.3	0.28
SSM-+	any n > 1	58	6,864	0.67	0.01
SSM-*	any n > 1	36	1,342	0.25	0.01

The size of the state space quickly increases with the model size for state enumeration methods. The symmetric Murφ-s shows significant reductions in both the global state space and

the search state space over the non-symmetric Murφ-ns. However, Murφ-s takes twice as much time as Murφ-ns to verify the model with five processors. The computation of canonical states for symmetrically equivalent states may be the cause for this abnormality.

Table 1 also shows that the simplified SSM-* reduced the number of essential states by 38% as compared to SSM-+. Additionally, the search state space is smaller and verification time is less. By abstracting and grouping global states further, SSM-* is superior to other methods in terms of performance as we show in next section. This performance improvement is obtained with no loss in accuracy.

The most complex essential state characterizing the delayed protocol and reported by the tool consists of 14 processors in different states (i.e., we have meta-states in the form $(q_1^{r_1}, q_2^{r_2}, \ldots, q_{14}^{r_{14}})$). Therefore, SSM effectively verifies models with more than 14 processors. For conventional protocols, a model of 3 or 4 processors is able to cover all possible cases (under atomic memory accesses). We can assume that the size of the state space will be excessive for the delayed protocol if non-atomic memory accesses are modeled

7 Verification at the Non-Atomic Transactions Level

In this section, we present the verification results for an implementation of the delayed protocol at the non-atomic transactions level for systems with non-FIFO networks. In a non-FIFO interconnection, messages between two nodes are not guaranteed to be received in the same order as they were issued. The protocol is an extension of the behavioral protocol in [10]. In this case, protocol transactions are non-atomic and are accomplished by exchanging coherence messages between processors.

Figure 9 shows the abstract verification model of the system. A detailed description of the protocol is given in [26]. In addition to caches, `ISBs`, and `IRBs`, each processor is associated with one *message sending channel* (*CH!*) and one *message receiving channel* (*CH?*) to model the message flow between caches and main memory [25]. We assume that messages are never lost. Of course, the verification at the message-passing level is much more complex than the verification at the behavior level because the number of possible system states is much larger.

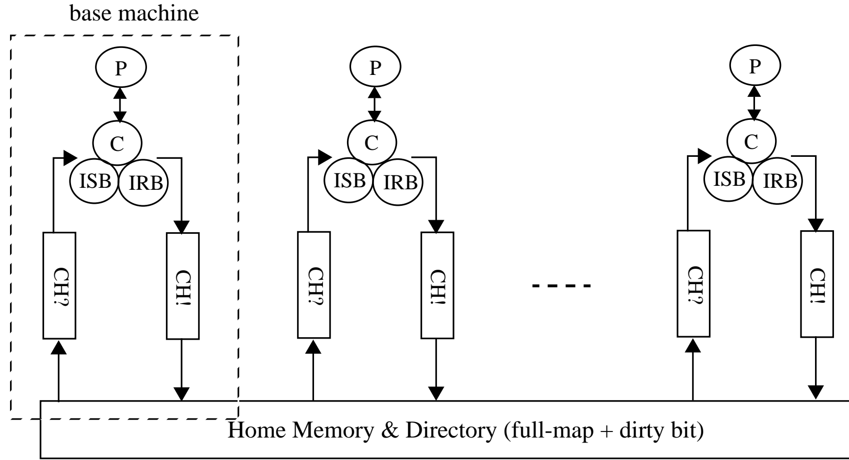

Figure 9. Abstract Verification Model at the Message-passing Level

7.1 Performance Results

Table 2 shows the results of applying the Murφ and SSM methods to verify the delayed protocol at the message-passing level. The state space explosion problem appears clearly in the case of Murφ, which uses the state enumeration method. Along with the exponential rise of the state space, the search state space, the verification time, and the memory requirement soar. For the model of four processors, Murφ-ns quickly takes up 500M bytes of memory. (Because of the enormous memory consumed, these runs are performed on a UltraSparc II workstation equipped with 512M bytes of memory.)

Even when the system symmetry is exploited in Murφ-s, the size of the state space is only reduced by the factorial of the number of processors. Given limited resources, the largest model we could verify includes four processors. This verification run uses more than 300M bytes of memory and takes more than 134 hours.

As compared to Murφ, SSM-+ is more efficient. The verification time is about 69 hours and the memory consumption is 210 M bytes. The most complex essential state consists of 111 bases machines in different states. This means that SSM-+ effectively verifies the delayed protocol model with 111 processors. Clearly, no existing method based on state enumeration is capable of handling such a complex model.

Note that the search state space of SSM-+ is much larger than that of Murφ. This is reasonable because the SSM model practically includes more processors. However, the verification time of SSM-+ is shorter. This is because our implementation of the SSM system uses the monotonicity property to remove non-essential states as soon as possible. Consider a current state S and its successors S_1, S_2,.....,S_n, which are generated in sequence. Suppose that $S \subseteq S_k$, where 1<k<n. Because of the property of monotonicity, we know that S, and $S_{1..k-1}$ can be removed immediately. Moreover, $S_{k+1..n}$ do not need to be generated. We simply keep on expanding S_k, which will lead to states covering $S_{1..k-1}$ and $S_{k+1..n}$. As a result, we avoid very costly comparisons between a newly generated state and all previous states. Although hashing is used in Murφ to speed up the comparison operations, the efficiency depends on the rate of hash conflict. When the hash function is not ideal or the utilization of hash table is high, the operations of searching for the state and resolving conflict become expensive. Another reason is that Murφ encodes state information into bit vectors in order to save memory. Therefore, the modification of state information is more expensive. On the other hand, the state information in SSM is not encoded. For instance, a state variable of boolean type is translated into an enumeration type of two elements {false, true} in the generated C program. In this case, the SSM uses 4 bytes to represent information which could be coded into a single bit. When the state space is small, this is normally not a problem [25].

TABLE 2. Performance Results of Different Methods for Verifying the Delayed Protocol (non-atomic transactions).

Method	Number of processors	Size of global state space	Size of search state space	Verification time (seconds)	Memory (Mbytes)
Murφ-ns	2	29,585	109,224	138.6	0.68
	3	2,617,224	13,922,454	19,448.6	70.14
	4	(excessive memory requirement > 500M bytes)			
Murφ-s	2	14,821	54,738	369.3	0.34
	3	442,496	2,358,487	6,190.7	11.86
	4	10,083,443	72,659,595	484,456.8	308.63
SSM-+	any n > 1	331,004	285,435,076	248,638.1	210.63
SSM-*	any n > 1	8,048	5,204,909	4,190.9	2.96

To improve the verification time of SSM, we must find optimal expansion paths. Figure 10

shows the number of essential states and the number of unexpanded states kept during the verification. In brief, the state expansion process maintains two lists of global states: a *history* list and a *queue* which respectively keeps expanded and unexpanded states. At each expansion step, a state S is popped from the queue and all of its successor states are generated. When a new state is generated, it is compared with all previous states which are kept in the history list and in the queue. If the new state is found (contained in SSM), it is discarded; otherwise it is pushed into the queue for further expansion. The current state S is saved in the history list if it is not contained by any newly generated state.

As shown in figure 10, the number of states kept in the history list increases rapidly at the beginning, but drops close to the end. Meanwhile, the queue shrinks slowly and steadily. Ideally, the number of expanded non-essential states should be kept to a minimum because they represent wasted work. In the run of figure 10, about 30% of the expanded states are non-essential. Although it does not show in figure 10, one can imagine that expanding non-essential states causes larger queue sizes. In the worst case, unexpanded essential states can be buried and wrapped by non-essential states, which prevents essential states from being popped out of the queue and expanded. Searching for optimal solutions is a common problem for many applications; unfortunately, we do not have an answer.

Finally, we observe that SSM-* cuts the state space of SSM-+ by 98%, which leads to dras-

tic reductions of the verification time and memory requirement, at equal verification accuracy.

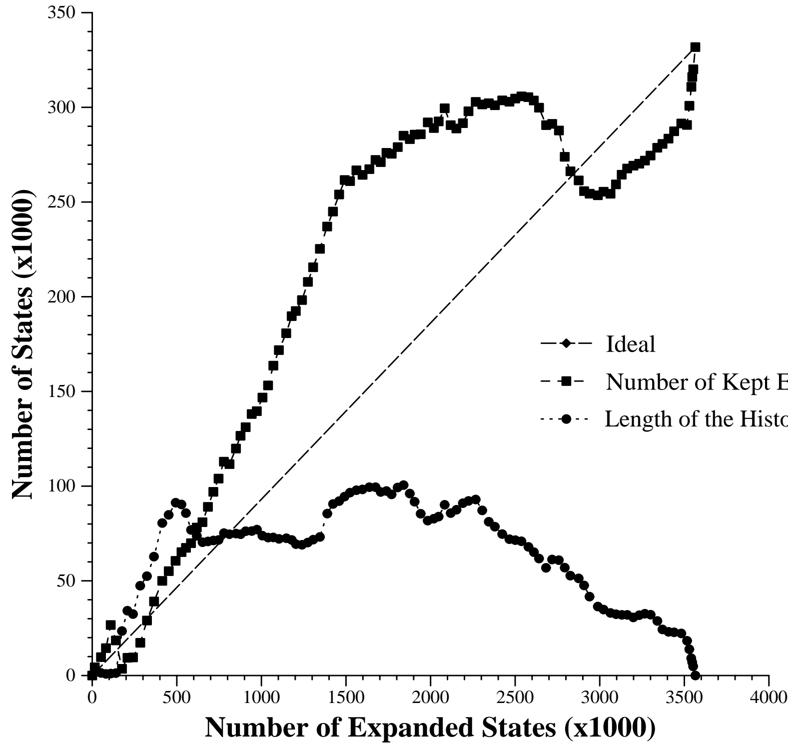

Figure 10. Efficiency of the SSM Method.

8 Conclusion

In this paper, we have verified the delayed protocol, which has high complexity even at the behavior level. This protocol delays the sending and receiving of invalidations until synchronization points in a weakly-ordered system. Because the design of delayed protocol covers a wide range of techniques that can be exploited in relaxed memory models for good performance, the verification approach can be applied to similar protocols. This includes protocols designed for systems using *write caches* [12] which can be verified in the same way because the functionality of write caches is essentially the same as the `ISB` employed in the delayed protocol.

Although the delayed protocol is complicated, the symbolic verification method effectively reduces the complexity of the verification process by only keeping track of 0, 1, or unknown number of caches in particular states. The set of repetition constructors used for abstract state represen-

tation is not surprising because cache coherence is similar to mutual exclusion or single-writer-multiple-readers protocols in distributed computing. There are only special cases (such as a single writer) where we need to explicitly keep track of its state. Other state machines can be simply lumped because they exhibit the same behavior.

We have also shown that the set of constructors in SSM can be simplified as compared to previous publication, resulting in much better performance. Interestingly, Ip and Dill have implemented a simplified variation of the SSM method and have reported a successful application of their tool in [17]. The difference is that our method works directly on the abstract state space whereas their tool expands explicit states and then constructs abstract states based on generated explicit states. Therefore, the tool may require multiple runs (adding one more processor to the model in each consecutive run) to reach the complete verification results obtained with our method. In any event, both experiences demonstrate that the SSM method is very effective in verifying complex cache protocols. For further exploration of verification techniques to cache protocols, one can refer to [24].

References

[1] Adve, S.V. and Hill, M.D., “Weak Ordering--A New Definition”, *Proc. of the 17th Int’l Symposium on Computer Architecture*, May 1990, pp. 2-14.

[2] Adve, S.V. and Hill, M.D., “A Unified Formalization of Four Shared-Memory Models”, *IEEE Trans. on Parallel and Distributed Systems*, August 1993, pp. 613-624. (Also, Technical Report #1051, University of Wisconsin.)

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

[4] Amza, C., Cox, A. L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W. and Zwaenepoel, W., “TreadMarks: Shared Memory Computing on Networks of Workstations”, *IEEE Computer*, pp. 18-28, Feb. 1996.

[5] Browne, M.C., Clarke, E.M. and Grumberg, O., “Reasoning about Networks with Many Identical Finite State Processes”, *Information and Computation* 81, 1989, pp. 13-31.

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

[7] Clarke, E. M., Grumberg, O. and Long, D. E., “Model Checking and Abstraction”, *ACM Transaction on Programming Languages and Systems*, Vol. 16, No. 5, Sep. 1994, pp. 1512-1542.

[8] Cousot P. and Cousot, R., “Abstract Interpretation Frameworks”, *Journal of Logic and Computation*, Vol. 2, No. 4, Aug. 1992, pp. 511-547.

[9] Dubois, M., Scheurich, C. and Briggs, F.A. “Memory Access Buffering in Multiprocessors”, *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp. 434-442.

[10] Dubois, M., et al. “Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs”, *Supercomputing,* Nov. 1991, pp. 197-206.

[11] Dubois, M., et al., “The Detection and Elimination of Useless Misses in Multiprocessors,” *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993, pp. 88-97.

[12] Dahlgren, F., and Stenström, P., “Using Write Caches to Improve Performance of Cache Coherence Protocols in Shared-Memory Multiprocessors”, Technical Report, Department of Computer Engineering, Lund University, April 1993.

[13] Galles, M. and Williams, E.,"Performance Optimizations and Verification Methodology of the SGI Challenge Multiprocessor", *Hawaii International Conference on System Sciences*, Jan 1994.

[14] Gharachorloo, K., et al. "Memory Consistency and Event Ordering in Shared-Memory Multiprocessors", *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990, pp. 15-26.

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

[16] Ip, C.N. and Dill, D.L., "Better Verification Through Symmetry", *Proc. 11th Int'l Symp. on Computer Hardwae Description Languages and Their Applications*, pp. 87-100, Apr. 1993.

[17] Ip, C.N. and Dill, D.L., "Verifying Systems with Replicated Components in Murφ", *Int'l Conf. on Computer-Aided Verification*, 1996.

[18] Lamport, L., "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", *IEEE Trans. on Computers*, Vol. C-28, No.9, Sept. 1979, pp. 690-691.

[19] Lynch, Nancy A. and Tuttle, Mark R., "Hierarchical Correctness Proofs for Distributed Algorithms", *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (PODC'87), pp. 137-151, August 1987.

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

[21] Pong, F. and Dubois, M., "A New Approach for the Verification of Cache Coherence Protocols", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6, No. 8, Aug. 1995, pp. 773-787.

[22] Pong, F., Browne, M., Aybay, G. Nowatzyk, A. and Dubois, M., "Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study", *Proc. of the First Int'l EURO-PAR Conf.,* Aug. 1995, pp. 287-300.

[23] Pong, F. and Dubois, M., "Formal Verification of Delayed Consistency Protocols", *Proc. of the 10th Int'l Parallel Processing Sym*., Apr. 1996, pp. 124 - 131.

[24] Pong, F. and Dubois, M., "A Survey of Techniques for Verifying Cache Coherence Protocols", *ACM Computing Surveys, Vol. 29,* Mar. 1997, pp. 83-126.

[25] Pong, F., “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. 1995.

[26] Pong, F., and Dubois, M., “The Verification of Relaxed Consistency Protocols with the Symbolic State Model,” USC Computer Engineering Report 97-12, Department of EE-Systems, University of Southern California, July 1997.

[27] Sweazey, P. and Smith, A.J., “A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus”, *Proc. of the 13th Int’l Symp. on Computer Architecture*, pp. 414-423, 1986.