Modeling and Verification of Cache Coherence Protocols

Lubomir Ivanov
Department of Computer Science
Iona College
715 North Avenue
New Rochelle, NY 10801
livanov@iona.edu

Ramakrishna Nunna
Department of Electrical and Computer Engineering
California State University Fresno, MS 94
Fresno, CA 93740
rnunna@csufresno.edu

Abstract

A cache coherence protocol is a set of rules, which cache controllers in a system with multiple cache memories must follow to maintain the consistency of data stored in the local cache memories as well as in main memory. MESI is a popular cache coherence protocol used to synchronize the operation of cache controllers in many Shared Memory MIMD systems. MESI is also used to maintain the consistency between the level-1 and level-2 caches of the Intel Pentium® microprocessor. I this paper we present a model of the MESI protocol based on the recently introduced series-parallel poset modeling and verification methodology. We illustrate the use of the new methodology by verifying a few properties 0of the MESI protocol.

Introduction

As the complexity of hardware systems and protocols increases, so does the likelihood of design errors, while the usefulness of the classical simulation and testing methods for uncovering these design faults decreases. A promising alternative is offered by the field of formal verification. Its goal is to create a mathematical model of the system under consideration, and, by proving facts about the system model, to ensure that all desired properties are satisfied, and unwanted properties and design faults are absent. An excellent overview of the field of formal verification can be found in [1]. Some powerful formal verification methods such as *Symbolic Model Checking* [1] and *ω-Automata Verification* [2] have become extremely popular, and have led to the development of industrial-level verification tools (SMV, FormalCheck, etc.). Unfortunately, the higher expressiveness of a method usually leads to a higher complexity of the associated verification algorithms. Thus, a number of new verification methods have emerged, which, while relatively less powerful and general, guarantee a significantly improved efficiency. Among these, several methods have been based on using partial orders to describe the dependence or independence of sets of events occurring in a hardware system [3], [4], [5], [6], [7]. The main appeal of using partial orders in modeling and verifying system behavior is in avoiding the study of all possible interleavings of events occurring during a run of the system. In addition, partial order models are usually very clear and intuitive, and the verification algorithms can be fully automated. In some cases, partial order methods have been shown to reduce the complexity of verifying properties of asynchronous circuits from exponential to polynomial.

In [8] we introduced a new formal verification method for proving timing properties of non-iterated systems. The method is based on the inductively defined notion of series-parallel posets. The associated verification algorithms are characterized by a low-order polynomial complexity. The method was further expanded [9] to allow the modeling and verification of globally-iterated / locally-non-iterated systems. In [10], we introduced a reduction methodology, which further improves the time- and space complexity of the verification algorithms. In an upcoming paper [14], we present the methodology for dealing with the much more complicated case of general iterated systems. A significant application of this methodology was presented in [12], where we outlined the formal verification of a handshaking protocol and the popular PCI local bus protocol.

In this paper we present another important application of our series-parallel poset methodology - the modeling and formal verification of the Modified/Exclusive/Shared/Invalid (MESI) cache coherence protocol for a system of n write-back cache memories in a Shared Memory MIMD multiprocessor system. We begin with a description of the system and a specification of the MESI cache protocol. We then present the formal model of the MESI protocol implemented by a set of n independent cache controllers. Next we demonstrate the verification of a few properties of the cache protocol. The main presentation is followed by a brief introduction to series-parallel posets, and an outline of the series-parallel poset verification approach for iterated systems. Finally, we briefly discuss some strengths and weaknesses of our methodology in the context of other related formal verification work.

Modeling and Verification of the MESI Protocol

A shared-memory MIMD computer involves a number of CPUs, each with its own local cache, sharing a common main memory over a shared bus or a specific interconnection network.

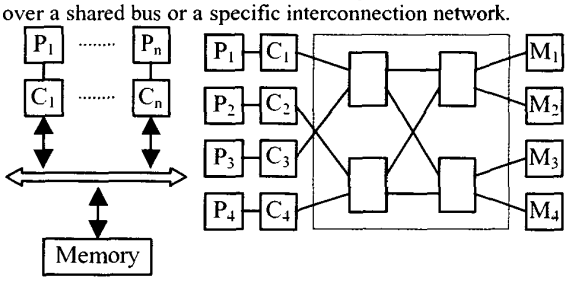

Fig. 1a - Shared Bus Fig. 1b - Omega Network

To maintain the consistency of data in the system, a protocol must be used to synchronize the operation of the cache controllers. The MESI Protocol is a popular cache coherence protocol used in the Intel Pentium microprocessor to maintain the data consistency between the Level-1 and Level-2 caches. We shall concentrate on the use of the MESI protocol in a shared-memory MIMD microprocessor system, where each processor has its own local cache. The individual caches are assumed to be write-back. The MESI protocol is an example of a snoopy cache coherence protocol, where cache controllers monitor (snoop) the activity on the bus and react whenever a particular transaction involves data stored in their local cache memories. Each cache line is assumed to be in one of four states:

0-7803-6685-9/01/$10.00©2001 IEEE

Authorized licensed use limited to: Indian Statistical Institute Kolkata. Downloaded on April 03,2024 at 10:56:03 UTC from IEEE Xplore. Restrictions apply.

- *Modified* - The cache line contains data not available in any other cache nor in main memory
- *Exclusive* - The cache line contains data not available in any other cache but available in main memory
- *Shared* - The cache line contains data available in other caches and in main memory
- *Invalid* - The cache line contains data which is no longer valid

Upon request for a data read or write from the CPU, each cache controller may take different actions [13]:

- *Read Hit*: The data requested by the CPU is found in the local cache, and read. No further action is needed.
- *Read Miss*: The data needed was not found in the local cache. The controller issues a *snoop request* to all cache controllers in the system. If other caches have copies of the required cache line (which will be in a *shared* state), they send a signal indicating the cache line is *shared*. The initiating cache then reads the data from memory and modifies its state to *shared*. If another cache has a copy of the line in the exclusive state, it signals the initiating cache that another copy of the line exists, and modifies the state of its line from *exclusive* to *shared*. The initiator, in the meantime, reads the line from main memory and changes its state to *shared*. If another cache has a *modified* copy of the line, it issues a signal to terminate the memory read transaction of the initiator. It then forwards the cache line directly to the initiator and modifies its state to *shared*. The initiator receives the line and also modifies its state to *shared*. Finally, if nobody has a copy of that cache line, the initiator reads the line from main memory and modifies its state to *exclusive*.
- *Write Hit*: The CPU wants to write a data item to a cache line in the CPU's local cache. If the line is *shared*, the cache controller broadcasts a signal to all other caches to *invalidate* their copies of the cache line, which is to be modified. The data is then written to the cache line, and its state changed to *modified*. If the cache line is in exclusive state, the cache controller does not need to broadcast an *invalidate* signal. It simply writes the data to the cache line, and switches its state to *modified*. Finally, if the cache line is already *modified*, the data is written in and the state of the line remains the same.
- *Write Miss*: The CPU attempts to write to a cache line, which is not in the local cache. The cache controller sends a *Read With Intent To Modify* (*rwitm*) signal to notify other caches, which may have a copy of the line, that it is to be modified. If another cache has the line in an *exclusive* state, or if several other caches have the line in a *shared* state, the state of the line is switched to *invalid*. The initiator then proceeds to read the cache line from main memory, write to it, and change its state to *modified*. If the cache line in question is held by another cache in a *modified* state, its cache controller issues a signal to stop the memory read transaction of the initiator, writes the line to main memory and transitions its state to *invalid*. The other controller repeats the memory read, writes to the newly acquired line, and changes its state to *modified*.

Each of the cache controllers repeatedly engages in one of the above activities, independently of the other cache controllers.

The behavior of a single cache controller is be modeled by the following series-parallel poset expression:

$$\boldsymbol{B}_{cache\_controller\_i}=(1+\boldsymbol{B}_{read\_hit\_i}+\boldsymbol{B}_{read\_miss\_i}+\boldsymbol{B}_{write\_hit\_i}+\boldsymbol{B}_{write\_miss\_i})^*$$

$$\boldsymbol{B}_{read\_hit\_i} = cr_i$$

$$\boldsymbol{B}_{read\_miss\_i} = send\_snoop\_rq_i \bullet (\otimes_{j=1..n,\, j\neq i}\, rcv\_snoop\_rq_i) \bullet ((\otimes_{k\in SH}\, send\_sh_k) \bullet rcv\_sh_i \bullet mr_i \bullet S + send\_sh_E \bullet (S_E \otimes (rcv\_sh_i \bullet mr_i) \bullet S_i) + send\_stop_M \bullet send\_cl_M \bullet (S_M \otimes (rcv\_stop_i \bullet rcv\_cl_i) \bullet S_i)) + mr_i \bullet E_i)$$

$$\boldsymbol{B}_{write\_hit\_i} = ((bcast\_I_i \bullet (\otimes_{k\in SH}\, (rcv\_bcast\_I_k \bullet I_k))) + 1) \bullet cw_i \bullet (M_i + 1)$$

$$\boldsymbol{B}_{write\_miss\_i} = send\_rwitm_i \bullet (\otimes_{j=1..n,\, j\neq i}\, rcv\_rwitm_j) \bullet ((I_E + \otimes_{k\in SH}\, I_k) \bullet mr \bullet cw_i \bullet M_i + send\_stop_M \bullet mw_M \bullet (l_M \otimes (rcv\_stop_i \bullet mr_i \bullet cw_i) \bullet M_i))$$

Note: The set *SH* includes all caches holding the cache line in a *shared* state. The subscripts *E* and *M* indicate that the signals are sent by a cache, holding the line in *exclusive* or *modified* state

Signal	Interpretation
cr	Cache Read
send_snoop_rq	Send snoop request
rcv_sh	Receive signal that the cache line is shared
mr	Memory read (followed by a cache line fill)
S	Change state to "Shared"
rcv_stop	Received stop signal from another cache controller
rcv_cl	Receive a cache line directly from another cache controller
E	Change state to "Exclusive"
bcast_I	Broadcast an "Invalidate" signal to all other cache controllers
cw	Cache write
M	Change state to "Modified"
send_rwitm	Issue a "Read with Intent to Modify" signal
Rcv_snoop_rq	Receive snoop request
send_sh	Send signal that the cache line is shared
send_stop	Send a stop signal to stop the main memory read of another cache controller
send_cl	Send a cache line directly to another cache controller
rcv_I	Receive an "Invalidate" signal
rcv_rwitm	Receive "Read with Intent to Modify" signal
mw	Main memory write

The behavior of a system of *n* cache controllers is given by:

$$\boldsymbol{B} = \boldsymbol{B}_{cache\_controller\_1} \otimes \boldsymbol{B}_{cache\_controller\_2} \otimes \ldots \otimes \boldsymbol{B}_{cache\_controller\_n} = \otimes_{i=1..n}\, \boldsymbol{B}_{cache\_controller\_i}$$

Let us now illustrate the verification process by verifying two properties of the MESI protocol:

"If cache i has a read miss for a cache line, and another cache, j, has that line in the exclusive state, then, after the read, the cache line will be in a shared state in both caches"

$$\boldsymbol{P}_1 = send\_snoop\_rq_i \bullet ((send\_sh_E + 1) \bullet (S_i \otimes S_E)) = send\_snoop\_rq_i \bullet (send\_sh_j \bullet (S_i \otimes S_E) + (S_i \otimes S_E))$$

$$\boldsymbol{AS}(\boldsymbol{B}_{read\_miss\_i}, \boldsymbol{P}_1):$$

$$\boldsymbol{B'}_{read\_miss\_i} = \boldsymbol{Pr}(\boldsymbol{B}_{read\_miss\_i}, set(\boldsymbol{P}_1)) = send\_snoop\_rq_i \bullet (send\_sh_j \bullet (S_E \otimes S_i) + (S_E \otimes S_i))$$

$$\boldsymbol{AS}(\boldsymbol{B'}_{read\_miss\_i}, \boldsymbol{P}_1) = \boldsymbol{AS}(\boldsymbol{B'}_{read\_miss\_i}, \boldsymbol{P}_1):$$

$\boldsymbol{AS}(send\_snoop\_rq_i, send\_snoop\_rq_i)$: TRUE

$\boldsymbol{AS}(send\_sh_E \bullet (S_E \otimes S_i) + (S_E \otimes S_i), send\_sh_E \bullet (S_E \otimes S_i) + (S_E \otimes S_i))$:

$\boldsymbol{AS}(send\_sh_E \bullet (S_E \otimes S_i) + (S_E \otimes S_i), send\_sh_j \bullet (S_E \otimes S_i))$:

$\boldsymbol{AS}(send\_sh_E \bullet (S_E \otimes S_i), send\_sh_E \bullet (S_E \otimes S_i))$:

$\boldsymbol{AS}(send\_sh_E, send\_sh_E)$: TRUE

$\boldsymbol{AS}((S_E \otimes S_i), (S_E \otimes S_i))$:

$\boldsymbol{AS}(S_E, S_E)$: TRUE

$\boldsymbol{AS}(S_i, S_i)$: TRUE

$pred_{SE}(\{S_i\}) = \varnothing \cap \{S_E\} = \varnothing$

$pred_{Si}(\{S_E\}) = \varnothing \cap \{Si\} = \varnothing$

$\Rightarrow \boldsymbol{AS}((S_E \otimes S_i), (S_E \otimes S_i))$: TRUE

$pred_{\,send\_shE \bullet (SE \otimes Si)}(\{S_i, S_E\}) = \{send\_sh_E\} \cap \{send\_sh_E\} = \{send\_sh_E\}$

$\Rightarrow \boldsymbol{AS}(send\_sh_E \bullet (S_E \otimes S_i), send\_sh_E \bullet (S_E \otimes S_i))$: TRUE

Authorized licensed use limited to: Indian Statistical Institute Kolkata. Downloaded on April 03,2024 at 10:56:03 UTC from IEEE Xplore. Restrictions apply.

$AS(\ (S_E \otimes S_i), send\_sh_E \bullet (S_E \otimes S_i))$:
$AS((S_E \otimes S_i), send\_sh_E)$: FALSE
$\Rightarrow AS(\ (S_E \otimes S_i), send\_sh_E \bullet (S_E \otimes S_i))$: FALSE
$\Rightarrow AS(send\_sh_E \bullet (S_E \otimes S_i) + (S_E \otimes S_i), send\_sh_E \bullet (S_E \otimes S_i))$: FALSE
$AS(send\_sh_E \bullet (S_E \otimes S_i) + (S_E \otimes S_i), (S_E \otimes S_i))$:
$AS(send\_sh_E \bullet (S_E \otimes S_i), (S_E \otimes S_i))$:
$AS((S_E \otimes S_i), (S_E \otimes S_i))$: TRUE (as shown above)
$\Rightarrow AS(send\_sh_E \bullet (S_E \otimes S_i), (S_E \otimes S_i))$:
$AS((S_E \otimes S_i), (S_E \otimes S_i))$: TRUE (as shown above)
$\Rightarrow AS(send\_sh_E \bullet (S_E \otimes S_i) + (S_E \otimes S_i), (S_E \otimes S_i))$: TRUE
$\Rightarrow AS(send\_sh_E \bullet (S_E \otimes S_i) + (S_E \otimes S_i), send\_sh_E \bullet (S_E \otimes S_i) + (S_E \otimes S_i))$: TRUE[1]

pred $_{B'read\_miss\_i}(\{send\_sh_E, S_E\ S_i\}) =$
$= \{send\_snoop\_rq_i\} \cap \{send\_snoop\_rq_i\} = \{send\_snoop\_rq_i\}$
$\Rightarrow$ $AS(B'_{read\_miss\_i}, P_1)$: TRUE
$\Rightarrow$ $AS(B_{read\_miss\_i}, P_1)$: TRUE

Consider now a second property:
"*On a write miss for cache i, all other copies of the cache line being written to are invalidated.*": $P_2 = send\_rwitm_i \bullet (\otimes_{k \in SH} I_k + I_E)$
$AS(B_{write\_miss\_i}, P_2)$:
$B'_{write\_miss\_I} = Pr(B_{write\_miss\_i}, set(P_2)) = send\_rwitm_i \bullet (I_E + \otimes_{k \in SH} I_k)$
It is now easy to show that $AS(B_{write\_miss\_i}, P_2)$ is TRUE.

Let us consider one more property:
"*On a cache write, before the cache line state is switched to modified, all other copies of this line are invalidated.*"
$P_3 = (\otimes_{k \in SH} I_k + I_E + I_M) \bullet M_i$, $B = B_{write\_hit\_I} + B_{write\_miss\_i}$
$AS(B, P_3)$:
$B' = Pr(B, set(P_3)) = Pr(B_{write\_hit\_i}, set(P_3)) + Pr(B_{write\_miss\_i}, set(P_3))$
$= (\otimes_{k \in SH} I_k) \bullet M_i + (I_E + \otimes_{k \in SH} I_k) \bullet M_i + I_M \bullet M_i = (I_M + I_E + \otimes_{k \in SH} I_k) \bullet M_i$
It easy to see, that the third property is satisfied as well.

Overview of Series-Parallel Posets

A *partially ordered set* (*poset*) is a set with a reflexive, antisymmetric, and transitive relation defined on the set elements. A Σ^**-labeled poset* $P=(P, \leq, l)$ consists of a poset $(P, \leq)$, and an assignment of a nonempty word (a label) $l(v) \in \Sigma^*$ to each vertex v in P. Given posets P and Q $(P \cap Q = \varnothing)$, we define two operations:

Concatenation (•): $P \bullet Q := (P \cup Q, \leq_{P \bullet Q})$
Shuffle (⊗): $P \otimes Q := (P \cup Q, \leq_{P \otimes Q})$,
where: $v \leq_{P \bullet Q} v' \Leftrightarrow v \leq_P v' \vee v \leq_Q v' \vee (v \in P \wedge v' \in Q)$
$v \leq_{P \otimes Q} v' \Leftrightarrow v \leq_P v' \vee v \leq_Q v'$

A *Series-Parallel Poset* over an alphabet Σ is defined inductively:

- The empty poset, 1, is a SPP
- For each $\sigma \in \Sigma$, the singleton poset labeled σ is a SPP
- If P and Q are SPPs, so are $P \bullet Q$ and $P \otimes Q$

Thus, the set of all series parallel posets formed from 1 and the singletons and closed under concatenation (•) and shuffle (⊗) forms a bimonoid denoted $SP(\Sigma^*)$ [11]. For our purposes, the alphabet, Σ, will consist of all distinct events occurring during a run of the system under consideration. Let each event, e_i, occurring in a system be represented by a singleton poset, e_i. The fact that event e_i precedes event e_j can be represented by $e_i \bullet e_j$. The independence of events e_i and e_j is represented by the series-parallel poset $e_i \otimes e_j$. This extends naturally to sets of events.

[1]There is at least one sub-property, which is always satisfied

What does it mean for two sets of events two be (in)dependent?
Two sets of events, P and Q, are *independent* if no event in P triggers a chain of events leading to the occurrence of an event in Q and vice versa. In other words P and Q are independent if the predecessor events of P are not in Q and vice versa.

A set of events P *always precedes* a set of events Q if all events in P occur before any event in Q does. This is so when each event in Q has all events in P as predecessors.

A set of events P *partially precedes* a set of events Q if P sometimes occurs before Q, but not always. In other words, P *partially precedes* Q if the occurrence of every event in Q is preceded by the occurrence of at least one event from P, i.e. each event in Q has at least one predecessor from P.

Let us illustrate the definitions with an example. Consider the following series-parallel poset:

$B = ((e_1 \otimes e_2) \bullet e_3) \otimes e_4$

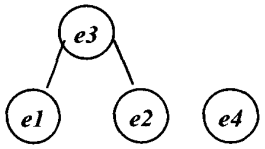

Fig.2 - A Series Parallel Poset Example

Consider now the sets of events $P_1 = \{e_1, e_4\}$, and $P_2 = \{e_3\}$. Clearly, P_1 and P_2 are not independent since e_1 must occur before e_3 does. P_1 does not always precede P_2 since one possible sequence of events is e_1, e_2, e_3, and then e_4. But P_1 may sometimes precede P_2 since another possible sequence of events is, for example, e_1, e_2, e_4, and then e_3.

Modeling and Verification of Iterated Systems

Interpreting series-parallel posets as descriptions of the dependence or independence of sets of events allows us to model system *behavior* in terms of event sequences.

A *non-iterated system* is one, in which the events are distinct and not repeated. *Non-iterated behavior* can be modeled by a single series-parallel poset over the alphabet of system events [8], [10].

An *iterated system* is one in which some or all events are repeated one or more times. Thus, an iterated system consists of a number of components, which function in series or independently such that each component is either an iterated- or a non-iterated system. To describe *iterated behavior* we need a new structure - the star shuffle semiring $\mathcal{S} = (S, +, \bullet, \otimes, *, 0, 1)$ of series-parallel posets:

- S - set of finite subsets of $SP(\Sigma^*)$, closed under the operations
- If $K \in S$ and $L \in S$, $K+L = \{P \mid P \in K \vee P \in L\} \in S$
- If $K \in S$ and $L \in S$, $K \bullet L = \{P \bullet Q \mid P \in K \wedge Q \in L\} \in S$
- If $K \in S$ and $L \in S$, $K \otimes L = \{P \otimes Q \mid P \in K \wedge Q \in L\} \in S$
- If $K \in S$, then $K^* = 1 + K + K \bullet K + \ldots = \Sigma_{i=0,\infty} K^i \in S$, where $K^i = K \bullet K \bullet \ldots \bullet K$, i times.
- 0 is the empty set of posets & 1 is the empty poset

We define the *behavior*, B, of an iterated system to be an element of the star-shuffle semiring S, i.e. $B \in S$. Thus, the behavior of an iterated system is a set of series-parallel posets. We can represent the *verification properties* as sets of series-parallel posets over a subset of the alphabet Σ, i.e. $P \in S$ as well. The *verification questions* are specified as the following *predicates*:

Authorized licensed use limited to: Indian Statistical Institute Kolkata. Downloaded on April 03,2024 at 10:56:03 UTC from IEEE Xplore. Restrictions apply.

- **SS(B, P)** is a binary predicate, interpreted as "The property P is sometimes satisfied within the behavior, B". The predicate takes a behavior and a property and verifies that the property can sometimes be traced within the behavior of the system.
- **AS(B, P)** is a binary predicate, interpreted as "The property P is always satisfied within the behavior, B". The predicate takes a behavior and a property and verifies that the property can always be traced within the behavior of the system.

There are four normal forms of behavior and property expressions:

- Concatenation: $B = B_1 \bullet B_2 \bullet \ldots \bullet B_n$ & $P = P_1 \bullet P_2 \bullet \ldots \bullet P_m$
- Shuffle: $B = B_1 \otimes B_2 \otimes \ldots \otimes B_n$ & $P = P_1 \otimes P_2 \otimes \ldots \otimes P_m$
- Plus: $B = B_1 + B_2 + \ldots + B_n$ & $P = P_1 + P_2 + \ldots + P_m$
- Star: $B = B_1^*$ & $P = P_1^*$

To simplify the reasoning about sets of events and the complexity of the verification algorithms we introduce the notion of a reduction of the system behavior. It is prompted by the fact that, while the system behavior may involve hundreds of thousands of events, in most cases the verification property involves only a few events. The reduction is carried out by a recursively defined *projection function Pr(B, set(P))*, which takes a behavior, B, and a set of events, and returns a reduced behavior, B', with respect to the events in the specified set. The effect of the projection function is to substitute I in place of all events *not in set(P)* without modifying the ordering of events in the behavior.

Based on a number of theorems, corollaries, and lemmas, which examine the satisfaction of all forms of properties with respect to all forms of behaviors, we derive the formal definition of the two verification predicates **SS(B, P)** and **AS(B, P)** for iterated systems. In this outline, we present only **AS(B, P)**:

AS(B, P) iff

- $P = e \wedge B = e$
- $P = P_1^* \wedge B = B_1 \otimes B_2 \otimes \ldots \otimes B_n \wedge \mathbf{AS}(B, P_1) \wedge \forall i \in [n] B_i = B_{i1}^*$
- $P = P_1^* \wedge B = B_1^* \wedge \mathbf{AS}(B_1, P_1)$
- $P = P_1 \otimes P_2 \otimes \ldots \otimes P_m \wedge B = B_1^* \wedge \mathbf{AS}(B_1, P)$
- $P = P_1 \otimes P_2 \otimes \ldots \otimes P_m \wedge B = B_1 \otimes B_2 \otimes \ldots \otimes B_n \wedge P \notin \mathrm{SP}(\Sigma^*) \wedge \forall i \in [m]\ \mathbf{AS}(B, P_i)) \wedge \forall i \in [m\text{-}1]\ \mathbf{Independent}_B(P_i, P_{i+1})) \wedge \forall i \in [m]\ (P_i = (P_{i1})^* \rightarrow \forall e \in P_i, L(set(lisc(B, e))) \subseteq L(set(P_i)))$
- $P = P_1 \bullet P_2 \bullet \ldots \bullet P_m \wedge (B = B_1 \bullet B_2 \bullet \ldots \bullet B_n \vee B = B_1^*) \wedge P \notin \mathrm{SP}(\Sigma^*) \wedge \forall i \in [m]\ \mathbf{SS}(B, P_i) \wedge \forall i \in [m]\ (P_i = (P_{i1})^* \rightarrow \forall e \in P_i, L(set(lisc(B, e))) \subseteq L(set(P_i))) \wedge \forall i \in [m\text{-}1]\ (\forall e \in P_{i+1}\ L(pred_B(\{e\})) \cap L(set(P_i)) = L(set(P_i)))$
- $B = B_1 + B_2 + \ldots + B_n \wedge \forall i \in [n]\ \mathbf{AS}(B_i, P)$
- $P = P_1 + P_2 + \ldots + P_n \wedge \exists\, i \in [n]\ \mathbf{AS}(B, P_i)$

In the above definitions we made use of number of functions – the labeling functions $l(s)$ and $L(\{s_1, \ldots, s_n\})$, the predecessor function, *pred*(P), and the functions *set*(P), "Non-Iterated", NI(B), and "Least Iterated Sub-Component", *lisc*(B, e). The exact definition of these functions is presented in [14] and omitted here for lack of space. We also used the auxiliary predicate **Independent$_B$(P, Q)**.

The predicates serve as the basis of a verification algorithm. The analysis of its requirements shows that the worst-case time complexity is $O(n+m^3)$, and the average case time complexity is $O(n+m^2)$, where n is the number of events in the behavior (before the reduction), and m is the number of events in the property. The space complexity is O(m).

Conclusions

In this paper, we presented the modeling and formal verification of the MESI cache coherence protocol. The approach is based on the developed series-parallel poset methodology. Relative to other formal modeling and verification methods, we are interested mainly in verifying the proper *sequencing* of events as given by the system specification. The advantage is the very low space- and time complexity of the verification algorithms.

Closest to our work is that of V.Pratt [4]. There are however important differences. The main stress in [4] is on modeling system behavior with the help of an extensive collection of operations. Our technique uses a far smaller collection of operations (•, ⊗, *), but models not only system behaviors but properties as well. The emphasis of our work is on verification. In that respect, the reduced collection of operations simplifies the analysis, and contributes to the efficiency of the algorithms.

One important shortcoming of our technique is the inability to model "N"-type dependencies among the events occurring in a system. These are encountered quite often in real systems and significantly limit the general applicability of our algorithms. We are currently working on extending our verification methodology to deal with this type of event dependencies as well. As we have demonstrated, though, the current technique is powerful enough to model many real-world systems and protocols.

References

[1] K. McMillan, "*Symbolic Model Checking*", Kluwer Academic Publishing, 1993

[2] R.Kurshan, "*Computer Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*", Princeton Series in CS, Princeton, 1994

[3] M.Nielsen, G.Plotkin, and G.Winskel, "*Petri nets, event structures, and domains*", part I, TCS, 13:85-108, 1981

[4] V.Pratt, "*Modeling Concurrency with Partial Orders*", International Journal of Parallel Programming, 1986

[5] P. Godefroid, "*Partial Order Methods for the Verification of Concurrent Systems: an Approach to the State Explosion Problem*", Doctoral Dissertation, University of Liege, 1995

[6] R. Nalumasu, G. Gopalakrishnan, "*A New Partial Order Reduction Algorithm for Concurrent System Verification*", Proceedings of IFIP, 1996

[7] D. Peled, "*Combining Partial Order Reductions with On-the-Fly Model Checking*", Journal of Formal Methods in Systems Design, 8 (1), 1996

[8] L.Ivanov, R.Nunna, S.Bloom, "*Modeling and Analysis of Non-Iterated Systems: An Approach Based on Series-Parallel Posets*", Proceedings of ISCAS'99, 1999

[9] L.Ivanov, R.Nunna, "*Formal Verification with Series-Parallel Posets of Globally-Iterated Locally-Non-Iterated Systems*", Proc. of MWSCAS'99, 1999

[10] L.Ivanov, R.Nunna, "*Formal Verification: A New Partial Order Approach*", Proc. of ASIC/SOC'99, 1999

[11] S. Bloom, Z. Esik, "*Free Shuffle Algebras in Language Varieties*", TCS 163 (1996) 55-98, Elsevier

[12] L. Ivanov, R. Nunna, "*Modeling and Verification of an Interconnect Bus Protocol*", Proc.of MWSCAS'00, 2000

[13] W. Stallings, "*Computer Organization and Architecture*", 5th Edition, Prentice Hall, 1996

[14] L. Ivanov, R. Nunna, "*Modeling and Verification of Iterated Systems and Protocols*", submitted to MWSCAS'01, 2001

Authorized licensed use limited to: Indian Statistical Institute Kolkata. Downloaded on April 03,2024 at 10:56:03 UTC from IEEE Xplore. Restrictions apply.