Rapid Parameterized Model Checking of Snoopy Cache Coherence Protocols

E. Allen Emerson and Vineet Kahlon

Department of Computer Sciences and Computer Engineering Research Center*
The University of Texas, Austin TX78712, USA

Abstract. A new method is proposed for *parameterized* reasoning about snoopy cache coherence protocols. The method is distinctive for being exact (sound and complete), fully automatic (algorithmic), and tractably efficient. The states of most cache coherence protocols can be organized into a hierarchy reflecting how *tightly* a memory block in a given cache state is bound to the processor. A broad framework encompassing snoopy cache coherence protocols is proposed where the hierarchy implicit in the design of protocols is captured as a *pre-order*. This yields a new solution technique that hinges on the construction of an *abstract history graph* where a global concrete state is represented by an abstract state reflecting the occupied local states. The abstract graph also takes into account the history of local transitions of the protocol that were fired along the computation to get to the global state. This permits the abstract history graph to exactly capture the behaviour of systems with an arbitrary number of homogeneous processes. Although the worst case size of the abstract history graph can be exponential in the size of the transition diagram describing the protocol, the actual size of the abstract history graph is small for standard cache protocols. The method is applicable to all 8 of the most common snoopy cache protocols described in Handy's book [19] from Illinois-MESI to Dragon. The experimental results for parameterized verification of each of those 8 protocols document the efficiency of this new method in practice, with each protocol being verified in just a fraction of a second. It is emphasized that this is parameterized verification.

1 Introduction

Cache protocols provide a vital buffer between the ever growing performance of processors and lagging memory speeds making them indispensable for applications such as shared memory multi-processors. Unfortunately, cache protocols are behaviorally complex. Ensuring their correct operation, in particular that they maintain the fundamental safety property of *coherence* so that different processes agree on their view of shared data items, can be subtle. The difficulty of the problem is often magnified as the number n of coordinating caches increases. Moreover, it is highly desirable that a cache protocol be correct independent of the magnitude of n. There is thus great practical as well

* This work was supported in part by NSF grants CCR-009-8141 & CCR-020-5483, and SRC contract 2002-TJ-1026. The authors' email addresses are {emerson,kahlon}@cs.utexas.edu.

as theoretical interest in uniform parameterized reasoning about systems comprised of n homogeneous cache protocols so as to ensure correctness for systems of *all* sizes n. This general problem is known in the literature as the *Parameterized Model Checking Problem (PMCP)*. It is in general algorithmically undecidable. Prior attempts to address the PMCP for cache protocols (cf. Section 5) have had a number of limitations, ranging from incompleteness to the need for considerable human intervention and ingenuity to potentially catastrophic inefficiency.

In this paper, we present a general method for solving the PMCP over snoopy cache coherence protocols of the sort commonly used in shared memory multiprocessors. Our framework includes all of the protocols in the book of Handy [19]. Our method is specialized to dealing with *safety* properties, as is appropriate for reasoning about coherence. We give a solution for this PMCP over our cache framework for safety that is distinguished by being exact (sound and complete), fully automatic (algorithmic), and having complexity bounds that are quite tractable. The worst case complexity of our general algorithm is single exponential time in the size of the state diagram of a single cache unit; however, our experimental results show that our algorithm performs *very* efficiently in practice. We have applied our method to verify parameterized versions of the MSI, MESI, MOESI, Illinois (MESI-type), Berkeley, N+1, Dragon, and Firefly cache coherence protocols.

In our framework, we model cache coherence protocols using a specialized variant of broadcast protocols [14] that we call *pre-ordered broadcast protocols*, where processes coordinate using broadcast primitives plus boolean guards. A broadcast transmission corresponds to a cache protocol putting a message on the bus; reception of such a message corresponds to snooping the bus and taking appropriate action. Boolean guards make it possible to model protocols (e.g., Illinois, Firefly, Dragon) that need to determine the presence or absence of the required memory block in other caches. Our approach exploits a key feature common to most snoopy cache coherence protocols [8]: their states can be organized into a *hierarchy* based on how *tightly* a memory block in a given state is bound to the processor. Consider, for example, the MSI cache coherence protocol (cf. Figure 1). A memory block in the *modified* state is intended to be used by at most one processor and can be written to by that processor locally without generating any memory transactions across the bus. So it is tightly bound to the processor. However, a block in the *shared* state can potentially be shared by multiple processes and cannot be modified locally. Hence it is less tightly bound to the processor. We make precise this notion of tightness by capturing it as a *pre-order*[1] on the state set of an individual cache protocol. Intuitively, a state higher in the order is more tightly bound to the processor than a state that is comparably lower in the order. For instance, in the case of the MSI protocol, the pre-order, $\preceq$, is given by $I \prec S \prec M$.

Our technique involves the construction of an *abstract history graph* over nodes of the form $(a, A) \in S \times 2^S$, where S is the set of states of the given cache protocol. The key idea is the following: We represent global state s of a system with n caches

[1] A *pre-order* on finite set S is a reflexive and transitive binary relation $\preceq$ on S. There are several associated relations. We say x is equivalent to y, written $x \approx y$, iff $x \preceq y \wedge y \preceq x$; x strictly precedes y, written $x \prec y$, iff $x \preceq y \wedge \neg(y \preceq x)$; x is incomparable to y, written $x \not\sim y$, iff $\neg(x \preceq y) \wedge \neg(y \preceq x)$.

by a tuple of the form $(a, A) \in S \times 2^S$. Here a denotes the local state of the process executing the most recent transition in the computation leading up to s that flushes all the other processes into some unique fixed state. The set A denotes the maximal set of states of S that could potentially be filled given arbitrarily many processes by firing (a stuttering of the) the sequence of local transitions that were fired in the system with n caches to get to s. The standard abstract graph construction used in, e.g., [25] just stores the set of local states occurring in a global state. Our new construction's extra historical information permits us to reason about an arbitrary number of caches in an *exact* fashion with respect to safety properties. In the worst case, the size of the abstract graph may be exponential in the size of the state diagram of the given cache protocol. But in practice the abstract graph tends to be small as documented by our empirical results. In our experiments, protocols with k states had abstract graphs with ck abstract states, for small $c \leq 2$. We believe this may be a reflection of the tendency for broadcast transitions to drive recipients from a wider range of cache states to a narrower (lower in the pre-order) range of cache states, thereby reducing the number of degrees of freedom possible for abstract states. Finally, we discuss how our technique enables us to generate error traces once an error is detected.

The rest of the paper is organized as follows. We begin by introducing the system model in section 2. In section 3, we present a model checking algorithm for verifying parameterized safety properties based on the construction of the abstract history graph. Applications and experimental results are discussed in section 4, while a comparison with related works and some concluding remarks are given in the final section 5.

2 Preliminaries

2.1 A Motivating Example

We use as an example the simple MSI cache coherence protocol. The state transition diagram for the MSI protocol is shown in figure 1. The symbols M, S and I stand for $modified$, $shared$ and $invalid$ states, respectively. The states are organized so that the closer the state is to the top, the more *tightly* is the memory block in that state bound to the processor. In our system model we capture this notion of tightness as a *pre-order*, $\preceq$, on the states of the cache protocol. The notation A/B means that if the controller observes the event A from the processor side of the bus then in addition to the state change it generates the bus transaction or action B. The null action is denoted by "-". Transitions due to observed bus transactions are shown as dashed arcs, while those due to local processor actions are shown in bold arcs. The $Bus\ Read$ ($BusRd$) transaction is generated by a process read ($PrRd$) request when the memory block is not in the cache. The newly loaded block is *promoted*, viz., moved up in the state diagram, from invalid to the shared state in the requesting cache. If any other cache has the block in the modified state and it observes a $BusRd$ transaction on the bus, then its copy is stale and so it *demotes* its copy to the shared state. We call such a transition a *low-push* broadcast. More generally broadcast transition $a \rightarrow b$ is a low-push transition with respect to $\preceq$ iff it forces every other process in a local that is strictly higher in the pre-order $\preceq$ than b to a state that is at most as high as b. The $Bus\ Read\ Exclusive$ ($BusRdX$) transaction is generated by a $PrWr$ to a block that is either not in the cache or is in the cache

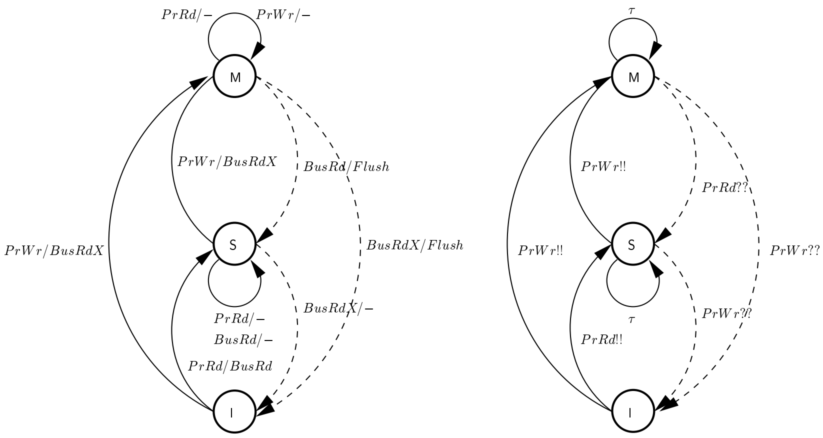

Fig. 1. The MSI Cache Coherence Protocol and its template

but not in the modified state. The cache controller puts the address on the bus and asks for an exclusive copy that it intends to modify. All other caches are invalidated. Once the cache obtains the exclusive copy, the write can be performed in the cache. This is an example of a *flush* broadcast transition, that forces every process other than the one firing the transition and in its non-initial state into a unique fixed state defined by the transition.

The template U for a protocol, such as MSI, is obtained from its state transition diagram through a simple abstraction, treating the behavior of the processors as purely nondeterministic. The transformation is straightforward, syntactic, and mechanical: Each transition generated by processor actions (represented by a bold line) and labeled by A/B, where $B \neq -$, is labeled with the broadcast send label $A!!$ while every transition generated by bus actions (represented by dashed lines) and labeled with B/C is labeled with the matching broadcast receive label $A??$. In the original diagram the relationship between a broadcast send A/B and its corresponding receive B/C was established with the common symbol B, while in the template it is established by the common symbol A in the labels $A!!$ and $A??$. Every bold transition labeled with $A/-$ represents a local action and is therefore labeled with the local transition label τ. The natural [2] pre-order $\preceq$ on U is $\mathsf{I} \prec \mathsf{S} \prec \mathsf{M}$. All transitions labeled with $PrRd$ are low-pushes with respect to $\preceq$, while those labeled with $PrWr$ are flushes.

2.2 The System Model: Pre-Ordered Broadcast Protocols

In this paper we consider families of systems of the form U^n, such that a pre-order, $\preceq$, can be imposed on the states of *template* U such that each transition of U is either a local transition or a flush broadcast or a low-push broadcast with respect to $\preceq$. Furthermore

[2] There is usually a natural and visually obvious pre-order, but there may be more than one suitable pre-order. A suitable pre-order can be constructed as shown in the section 3.4.

the transition could also be labeled with the specialized disjunctive guard $\bigvee \neg(\mathsf{i})$ or the specialized conjunctive guard $\bigwedge(\mathsf{i})$. We call such systems *pre-ordered broadcasts*.

The process template U is formally defined by the 4-tuple $(S, \Sigma, R, \mathsf{i})$, where

- S is a finite, non-empty set of *states*.
- Σ is a finite set of *labels* including the local transition label τ, broadcast labels $l!!$ and receive labels $l??$.
- The local transition relation R is such that each transition tr is either local $a \xrightarrow{g:\tau} b$, or a broadcast, $a \xrightarrow{g:l!!} b$, or a receive $a \xrightarrow{g:l??} b$.

We assume that receives are deterministic: for each label $l!!$ appearing in some broadcast send and for each state s in S, there is a unique corresponding receive transition on $l??$ out of s.

The guard g labeling each transition tr of R is either the boolean expression *true* or the *specialized conjunctive* guard $\bigwedge(\mathsf{i})$, or the *specialized disjunctive* guard $\bigvee \neg(\mathsf{i})$. We assume that the guard is *true* for receive transitions. In practice, the above mentioned guards suffice in modeling cache coherence protocols as each cache only needs to know whether another cache has the memory block it requires, expressed using the specialized disjunctive guard, or whether no other cache has it, expressed using the specialized conjunctive guard.

We further stipulate a pre-ordering, $\preceq$, on the state set S of U such that i is the minimum element, i.e., for all local states $a \neq \mathsf{i}$, we have $a \succ \mathsf{i}$, and such that each broadcast transition tr is of either of the two forms

1. *Flush*: Given state a of U, transition $b \xrightarrow{l!!} c \in R$, where $c \neq \mathsf{i}$, is called an *a-flush* transition provided that there exists the matching receive transition $\mathsf{i} \xrightarrow{l??} \mathsf{i}$ in R and for each state $d \neq \mathsf{i}$ of U, there is a matching receive transition of the form $d \xrightarrow{l??} a$ in R; a *flush* transition is an a-flush for some a. Intuitively, an a-flush transition pushes every process in its non-initial state, other than the one firing the transition, into local state a.
2. *Low-push*: Transition $a \xrightarrow{l!!} b$ is a *low-push* transition provided that, $b \neq \mathsf{i}$, $\neg(b \prec a)$, and for each state c such that $b \prec c$ there is a matching receive transition of the form $c \xrightarrow{l??} d$ such that $d \preceq b$; and, for all other states c, there is a matching self-loop receive transition $c \xrightarrow{l??} c$. Intuitively transition $a \xrightarrow{l!!} b$ is a low-push if it pushes every process in a local state strictly higher than b in the pre-order $\preceq$ into a state at most as high as b while leaving the rest of the processes untouched.

In practice, a natural pre-order $\preceq$ is normally supplied along with the diagram of U as it drawn in appropriate levels. If not, there is given in the section 3.4 an efficient algorithm (O($|U|^2$)) to compute an appropriate pre-order if one exists.

To capture block replacement behavior, we also require that templates be *initializable* [3]. This means that from each state a of a protocol, there is a local transition of the form $a \xrightarrow{\tau} \mathsf{i}$. Such initializations model block replacement behavior, where a cache is non-deterministically pushed into its invalid state, irrespective of the current state of the

[3] Initializability is not needed for the mathematical results of section 3.1; however, it is needed for the results of section 3.2.

block. For simplicity, re-initialization transitions and self-loop receptions are not drawn in state transition diagrams of cache protocols (cf. [8]).

Given U, the state transition digram for $U^n = (S^n, \Sigma, R^n, \mathrm{i}^n)$, the system with n copies of U, is based on interleaving semantics in the standard way.

A *path* $x = x_0x_1...$ of U^n is a sequence of states of S^n starting at the initial state i^n of U^n such that for every $i \geq 0$, $(x_i, a, x_{i+1}) \in R^n$ for some $a \in \Sigma$. For global state s of U^n, and $i \in [1:n]$, we use $s[i]$ to denote the local state of process U_i in s and for computation path z of U^n, we use $z[i]$ to denote the local computation path of U_i in z, viz., the sequence $z_0[i]z_1[i]...$. We write $x.s \in U^n$ to mean that finite computation path x of U^n ends in global state s. In this paper we will focus on finite paths and computations as they suffice for safety. Finally, given global state s of U^n, and local state a of U, we let $num(a, s)$ denote the number of copies of a in s, viz., the number of processes in local state a in global state s.

3 Safety Properties

Given a state a of U, we say that a is *reachable* iff there exists n such that there is a finite computation of U^n leading to a state with a process in local state a. For cache coherence protocols, we are typically interested in *pairwise reachability*, viz., given a pair (a, b) of local states a and b of template U, deciding whether for some n, there exists a reachable global state of U^n, with a process in each of the local states a and b. For instance, in the case of the MSI protocol, we are interested in showing that none of the pairs in the set $\{(M, M), (M, S)\}$ is pairwise reachable.

3.1 Systems without conjunctive guards

In this section, we assume that U is a template without conjunctive guards; guards of the form *true* or $\bigvee \neg(\mathrm{i})$ are permitted. This allows us to handle the MSI, MOESI, MESI (not the Illinois version which is handled in the next section), Berkeley and N+1 protocols.

A standard technique for reasoning about parameterized systems involves the construction of an abstract graph to capture the behaviour of a system instance of arbitrary size. Classically, the abstract graph is defined to be a transition diagram over the set 2^S with a given concrete global state s of a system instance U^n being mapped via mapping ϕ, say, onto the set $A = \{a_i | num(a_i, s) \geq 1\}$. For $B, C \in 2^S$, a transition is introduced from B to C in the abstract graph iff there exists m and concrete states t and u of U^m such that $\phi(t) = B$, $\phi(u) = C$ and u results from t by firing a concrete transition of U^m. There is a loss of information in the mapping ϕ which is reflected in the fact that it might not be possible to identify a unique successor B of A in the abstract graph that results by firing a transition $tr = a \rightarrow b$, where $a \in A$. For instance if tr is a local transition, then two different successors are possible: $B_1 = A \setminus \{a\} \cup \{b\}$ and $B_2 = A \cup \{b\}$ depending, respectively, on whether there is exactly one or at least 2 copies of a in the concrete state that maps onto A. To preserve soundness we cover for both cases and introduce both B_1 and B_2 as possible successors. However this may generate bogus paths in the abstract graph, viz., paths for which there do not exist matching concrete computations. Thus there might exist paths in the abstract graph that don't "lift" to concrete computations and hence the above technique though sound is not complete.

In this paper to check pairwise reachability, we use the *abstract history graph* of U, denoted by $\mathcal{A}_U$, where we bypass the above problem by mapping each concrete state s onto a tuple of the form (a, A), that denotes a formal state with at least one copy of state a and finite but arbitrarily many copies of each state in A. As we later show this permits us to reason about safety properties in a sound and complete fashion.

Definition (representative) Given template $U = (S, \Sigma, R, \mathsf{i})$, and a finite computation $x.s$ of U^n, we define *rep*$(x.s)$ to be the tuple $(a, A) \in S \times 2^S$, where, if no flush transition was fired along x, then $a = \mathsf{i}$ and $A = \{s[j] | j \in [1:n]\}$; and if U_i is the process to last fire a flush transition along x, then $s[i] = a$ and $A = \{s[j] | j \in [1:n] \wedge j \neq i\}$.

Given template U, the *abstract history graph*, $\mathcal{A}_U = (\mathcal{S}_U, \mathcal{R}_U, (\mathsf{i}, \{\mathsf{i}\}))$, is a transition diagram defined over tuples of the form $(a, A) \in S \times 2^S$. For $x.s \in U^n$, for some n, we will show how to map $x.s$ onto a tuple of the form (a, A). This mapping depends not only on the global state s but also on x, viz., the history of the computation leading to s and thus the term *abstract history graph*. Essentially in tuple (a, A), state a records the local state in s of the process executing the last flush along x, whereas A is a superset of the set of the local states of the remaining processes. This dichotomy is justified on the basis of the fact that we can pump up the multiplicity of each local state in s to any desired value except possibly of the current local state in s of the process to last execute a flush along x which could have multiplicity exactly one as we later show.

We now define the transition relation $\mathcal{R}_U$. Towards that end, given a tuple (a, A) and a local or a broadcast send transition $tr = c \rightarrow d$, we define the successor of (a, A) via tr as either the *state-successor*, denoted by *state-succ*$((a, A), tr)$ or the *set-successor* of (a, A), denoted by *set-succ*$((a, A), tr)$. As mentioned above, we think of (a, A) as a state with finite but arbitrarily many copies of each state in A plus one copy of a. The case of the *state-successor* captures the scenario when a process in local state a that possibly has multiplicity only one fires tr while the case of the *set-successor* captures the scenario when a process in local state $c \in A$ with arbitrarily large multiplicity fires enabled transition tr.

Definition (state-successor) Let $(a, A) \in S \times 2^S$ and let transition $tr = a \rightarrow b \in R$ labeled by guard g, be enabled in (a, A), viz., if $g = \bigvee \neg(\mathsf{i})$, then $\exists a' \in A : a' \neq \mathsf{i}$. Then *state-succ*$((a, A), tr) = (b, B)$, where if tr is a local transition then $B = A$ and if tr is a broadcast send transition then $B = \{b' | \exists a' \in A : \exists a' \longrightarrow b' \in R$ *that is a matching receive for tr* $\}$.

As an example, since firing the transition $tr = \mathsf{I} \stackrel{PrRd!!}{\longrightarrow} \mathsf{S}$ of the MSI protocol affects only processes in state M by causing them to transit to state S, therefore *state-succ*$((\mathsf{I}, \{\mathsf{I}, \mathsf{S}\}), tr) = (\mathsf{S}, \{\mathsf{I}, \mathsf{S}\})$.

Definition (set-successor) Let $(a, A) \in S \times 2^S$ and let transition $tr = b \rightarrow c \in R$, where $b \in A$, be such that if tr is labeled by guard g then it is enabled in (a, A), viz., if $g = \bigvee \neg(\mathsf{i})$, then for some $a' \in \{a\} \cup A$: $a' \neq \mathsf{i}$. Then, *set-succ*$((a, A), tr)$, is defined as the tuple

– $(c, \{c', \mathsf{i}\})$ if tr is a c'-flush transition

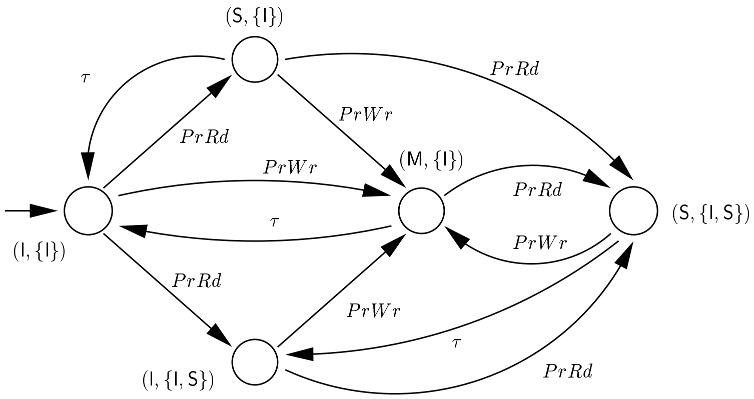

Fig. 2. The abstract history graph for the MSI Cache Coherence Protocol

- $(a, A \cup \{c\})$ if tr is a local transition. Note that since we had arbitrarily many copies of b to start with so even after firing local transition tr we are guaranteed arbitrarily many processes in local state b which is therefore not excluded from the second component of the resulting tuple.
- (d, B) if tr is a low-push broadcast transition, where $a \longrightarrow d$ is the (unique) matching receive for tr from a and $B = \{c\} \cup \{b' | \exists a' \in A : \exists a' \longrightarrow b' \in R$ that is a matching receive for tr $\}$. As in the previous case since we have arbitrarily many copies of b so in B we include the local state that results from firing the matching receive for tr from b which by definition of a low push transition (and the fact that $b \preceq b$) is b itself.

As an example, since firing the transition $tr = \mathsf{S} \xrightarrow{PrWr!!} \mathsf{M}$ of the MSI protocol flushes every other process into state I, therefore *set-succ*$((\mathsf{I}, \{\mathsf{I}, \mathsf{S}\}), tr) = (\mathsf{M}, \{\mathsf{I}\})$. We now formally define the abstract history graph of a template U.

Definition (Abstract History Graph) Given template $U = (S, \Sigma, R, \mathsf{i})$, the *abstract history graph* of U, is defined to be the tuple $\mathcal{A}_U = (\mathcal{S}_U, \mathcal{R}_U, (\mathsf{i}, \{\mathsf{i}\}))$, where $\mathcal{S}_U = S \times 2^S$ and $\mathcal{R}_U = \{((a, A), (b, B)) | (b, B) =$ *state-succ*$((a, A), tr))$ *or* $(b, B) =$ *set-succ*$((a, A), tr))$ *for some local or broadcast send transition tr of* $U\}$.

As an example, the abstract history graph for the MSI protocol is shown in figure 3. Self loops are omitted for the sake of simplicity. For convenience, we have labeled each transition of the graph by the label of the transition responsible for “firing” it.

Note that as opposed to the classical construction, given a tuple (a, A) and transition tr both the set-successor and state-successor of (a, A) via tr are uniquely defined. This is because as will be shown in proposition 3.3, we can have arbitrarily many copies of each state in A thereby alleviating the problem of considering the different successors that may arise from concrete states with different counts of local states as was the case with the classical abstract graph construction. This permits us to give exact path correspondences between the parameterized family of concrete systems and the abstract history graph as we now show. Since we are dealing with systems of a "disjunctive" nature having (arbitrarily many) extra copies does not disable any transitions.

Given $x.s \in U^n$, the precise mapping of $x.s$ onto a tuple of $\mathcal{A}_U$ is given by the ω-representative of $x.s$, denoted by ω-*rep*$(x.s)$.

Definition (ω-representative) Let $x = x_0...x_l$ be a finite computation path of U^n. Then we define the *ω-representative* of $x.x_l$, denoted by ω-*rep*$(x.x_l)$, as the tuple $(a, A) \in S \times 2^S$, defined as follows: If $l = 0$, then $(a, A) = (\mathsf{i}, \{\mathsf{i}\})$, else suppose that transition $x_{l-1} \rightarrow x_l$ is initiated by transition tr of U, fired locally by process U_j and let U_k be the process to last execute a flush transition in $x_0...x_{l-1}$. Then

$$(a, A) = \begin{cases} state\text{–}succ(\omega{-}rep(x_0...x_{l-1}.x_{l-1}), tr) & if\ j = k \\ set\text{–}succ(\omega{-}rep(x_0...x_{l-1}.x_{l-1}), tr) & otherwise \end{cases}$$

The tuple *rep*$(x.s)$ specifies the actual set of states present in the global state s, having followed path x through U^m. In contrast, the ω-representative ω-*rep*$(x.s)$ incorporates not only the local states present in s but also the states that could potentially be present, given sufficiently many processes n, in a global state of U^n that results from firing (a stuttering of) the same local transitions as were fired along x to get to s. Thus, ω-*rep*$(x.s)$ drags along some "history" of the computation x leading to s, and thereby stores more information than *rep*$(x.s)$. This is formalized as follows.

Proposition 3.1 (Containment Property) Given $x.s \in U^n$, such that *rep*$(x.s) = (a, A)$ and ω-*rep*$(x.s) = (b, B)$, we have $a = b$ and $A \subseteq B$.

We now establish a "path correspondence" between finite computations of U^n and between finite paths of $\mathcal{A}_U$ starting at $(\mathsf{i}, \{\mathsf{i}\})$.

Proposition 3.2 (Projection) For any finite path $x.s$ in U^n, there exists a finite path $y.t$ in $\mathcal{A}_U$ starting at $(\mathsf{i}, \{\mathsf{i}\})$ such that $t = \omega$-*rep*$(x.s)$.

For the other direction, we have

Proposition 3.3 (Lifting) Let x be a path of $\mathcal{A}_U$ starting at $(\mathsf{i}, \{\mathsf{i}\})$ and leading to tuple (a, A) of $\mathcal{A}_U$. Then, given $p \geq 1$, there exists $y.t \in U^n$, for some n, such that *rep*$(y.t) = (a, A)$ and t has at least p copies of each state in A plus a copy of a.

Combining the previous three results, we have

Theorem 3.4 (Decidability Result) Pair $(a, b) \in S \times S$ is pairwise reachable iff there exists a path in $\mathcal{A}_U$ starting at $(\mathsf{i}, \{\mathsf{i}\})$ to a tuple of the form (c, C) where either $a = c$ and $b \in C$; or $b = c$ and $a \in C$; or $a \in C$ and $b \in C$.

Thus we have reduced the problem of pairwise reachability for a pair of local states of a given template U to the problem of reachability in $\mathcal{A}_U$, the abstract history graph constructed from U. Since the size of the abstract graph is O($|U|2^{|U|}$), we have .

Corollary 3.5 The pairwise reachability problem for a pair of local states of a given template U can be solved in time O($|U|2^{|U|}$), where $|U|$ is the size of template U as measured by the number of states and transitions in U.

Note that in the construction of $\mathcal{A}_U$, it suffices to consider only the set of tuples reachable from the initial tuple $(\mathsf{i}, \{\mathsf{i}\}))$. In practice, the number of states of this graph may be much smaller than the worst case scenario where it could be $|S| \times 2^{|S|}$. This is illustrated clearly by our experimental results in section 4.2.

3.2 Adding the Specialized Conjunctive Guard

To reason about systems wherein the templates are augmented with the specialized conjunctive guard along with the assumption of initializability, we use a modification of the abstract history graph. Broadly speaking, the intuition behind the modification is that we can make the specialized conjunctive guard of a process evaluate to true starting at any global state by driving all the other processes into their respective initial states by making use of the local initializing transition mentioned above. Thus for every tuple (a, A) in the abstract history graph, we add a transition of the form $(a, A) \rightarrow (a', \{\mathsf{i}\})$ where either $a' = a$ or $a' \in A$ to $\mathcal{A}_U$.

Definition (Modified Abstract History Graph) Given template $U = (S, \Sigma, R, \mathsf{i})$ and its abstract graph $\mathcal{A}_U = (\mathcal{S}_U, \mathcal{R}_U, (\mathsf{i}, \{\mathsf{i}\}))$, define the modified abstract graph $\mathcal{A}_U^{\tau}$ to be the tuple $(\mathcal{S}_U, \mathcal{R}_U^{\tau}, (\mathsf{i}, \{\mathsf{i}\})))$, where $\mathcal{R}_U^{\tau}$ is the set of all transitions $((a, A), (b, B))$, where

- $B = \{\mathsf{i}\}$ and either $b = a$ or $b \in A$. This transition corresponds to the successive firing of the local initializing transition that leaves one process in state $b \in \{a\} \cup A$ and the rest of the processes in their initial states, thereby enabling guard $\bigwedge(\mathsf{i})$ labeling its transitions.
- $A = \{\mathsf{i}\} = B$ and $\exists tr = a \rightarrow b \in R$ labeled by $\bigwedge(\mathsf{i})$. This corresponds to the firing of a transition labeled with $\bigwedge(\mathsf{i})$.
- $\exists tr \in R$ labeled either by $\bigvee \neg(\mathsf{i})$ or by true such that either $(b, B) =$ *state-succ*$((a, A), tr)$ or $(b, B) =$ *set-succ*$((a, A), tr)))\}$ This correspond to the firing of transitions labeled with $\bigvee \neg(\mathsf{i})$ or true.

Then, as in section 3.1, we can show a "path correspondence" between concrete finite computations of U^n and finite paths in $\mathcal{A}_U^{\tau}$ starting at $(\mathsf{i}, \{\mathsf{i}\})$. The proofs are similar and are therefore omitted. Thus as in section 3.1, we have the following decidability result from which it follows, as before, that for this model of computation, pairwise reachability can be decided in time O($|U|2^{|U|}$), where $|U|$ is the size of the template U.

Theorem 3.6 (Decidability Result) Pair $(a, b) \in S \times S$ is pairwise reachable iff there exists a path in $\mathcal{A}_U^{\tau}$ starting at $(\mathsf{i}, \{\mathsf{i}\})$ to a tuple of the form (c, C) where either $a = c$ and $b \in C$; or $b = c$ and $a \in C$; or $a \in C$ and $b \in C$.

3.3 Generating Error Traces

A critical part of the verification process, once an error is detected, is the generation of a concrete computation of the system at hand leading to an erroneous global state.

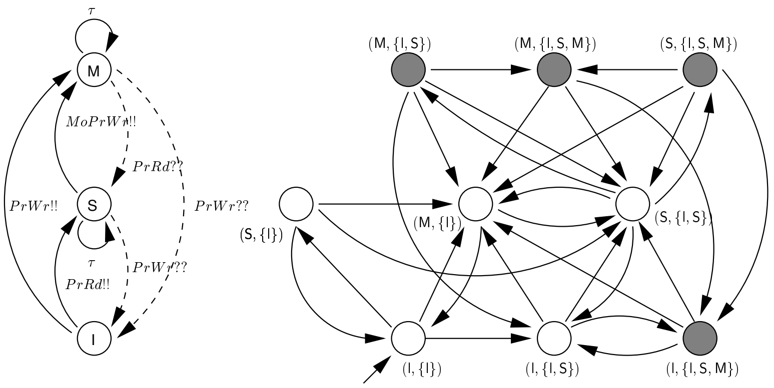

Fig. 3. The template for the Broken MSI Protocol and its abstract history graph

Till now, we have shown how to reduce the verification process for safety properties of the parameterized version of a given cache protocol to reachability analysis over the corresponding abstract history graph. This only allows us to detect an erroneous state in the abstract history graph and thereby construct a path in the abstract graph to an erroneous state. To get back a concrete computation of an instance of an original system leading to a concrete erroneous state, we make use of the construction used in proving proposition 3.3. Given a path x starting at the initial tuple $(\mathsf{i}, \{\mathsf{i}\})$ leading to an erroneous tuple (a, A) of the abstract history graph, this construction can be used to give a fully automated procedure to construct a finite computation y of a concrete system U^n, for some n, ending in a state t such that *rep*$(y.t) = (a, A)$. In general, n is of size linear in the length of x, viz., O$(|S|2^{|S|})$ in the worst case. But, as mentioned above, in practice, the number of states of the abstract history graph reachable from its initial state tend to be small and consequently so does the length of y. The ability to automatically generate error traces distinguishes our work from [9], where no effective way to generate error traces was given.

We now illustrate the construction with a broken version of the MSI protocol (figure 3). The MSI protocol is clobbered by replacing the flush transition labeled with $PrWr!!$ from the shared state to the modified state by a low push transition labeled with $MoPrWr!!$. In the abstract history graph, self loops are omitted for simplicity reasons and erroneous tuples are shaded. Note that the erroneous pair $(\mathsf{I}, \{\mathsf{I}, \mathsf{S}, \mathsf{M}\})$ can be reached via the path $(\mathsf{I}, \{\mathsf{I}\}) \rightarrow (\mathsf{I}, \{\mathsf{I}, \mathsf{S}\}) \rightarrow (\mathsf{I}, \{\mathsf{I}, \mathsf{S}, \mathsf{M}\})$ by firing a transition labeled with $PrRd$ followed by a transition labeled with $MoPrWr$. From this path we can get back a concrete computation of a system with 3 caches by firing transitions labeled with $PrRd$, $PrRd$ and $MoPrWr$ in the order listed, a stuttering of the sequence $PrRd$, $MoPrWr$. The resulting concrete computation is: $(\mathsf{I}, \mathsf{I}, \mathsf{I}) \stackrel{PrRd_1!!}{\longrightarrow} (\mathsf{S}, \mathsf{I}, \mathsf{I}) \stackrel{PrRd_2!!}{\longrightarrow} (\mathsf{S}, \mathsf{S}, \mathsf{I}) \stackrel{MoPrWr_1!!}{\longrightarrow} (\mathsf{M}, \mathsf{S}, \mathsf{I})$. Here symbol a_i labeling a transition indicates that process U_i fires a transition of template U labeled with a.

3.4 Automatic Construction of Pre-order

In practice, one can usually obtain the natural pre-order by drawing the diagram in levels, reflecting how tightly a memory block in a given cache state is bound to the processor. Such levels are used in the textbook by Culler [8] et al. If not, we can efficiently exhibit a feasible pre-order, $\preceq$, that can be imposed, or determine that none exists.

We proceed by constructing the labeled, directed graph $Q_U = (S, \{\preceq, \prec, \neg \prec\}, E)$, where $E \subseteq S \times \{\preceq, \prec, \neg \prec\} \times S$ is its edge set. For $a, b \in S$, an edge of the form $(a, \preceq, b)$ represents $a \preceq b$, $(a, \prec, b)$ indicates $a \prec b$ and $(a, \neg \prec, b)$ means $\neg(a \prec b)$. We construct Q_U as follows.

1. Initially, $E = \{(\mathsf{i}, \prec, a) | a \neq \mathsf{i}, a \in S\}$. This is because of the assumption we made in the system model that for each $a \neq \mathsf{i}$, we have $\mathsf{i} \prec a$.

2. For each non-local transition or non-flush broadcast send transition[4], $tr = (a, l!!, b)$, we have $\neg(b \prec a)$. Thus we augment E by adding the edge $(b, \neg \prec, a)$. Furthermore if $(c, l??, d)$ is a matching receive for tr such that $c \neq d$, then we have that $d \preceq b \prec c$ and so we add the edges $(d, \preceq, b)$ and $(b, \prec, c)$ to E. On the other hand if $(d, l??, d)$ is a matching receive for tr, then we have that $\neg(b \prec d)$ and so we add the edge $(b, \neg \prec, d)$ to E. If E already contains an edge of the form $(e, \preceq, f)$, then in case we add the edge $(e, \prec, f)$ to E in the above step, we remove $(e, \preceq, f)$ to ensure that there is only one edge from e to f labeled with $\preceq$ or $\prec$.

Let Q'_U be the subgraph of Q_U that we get by deleting all edges labeled with $\neg \prec$. Then we can impose a pre-order $\preceq$ on the states of U compatible with its transitions iff

(1) there does not exist a cycle in Q'_U containing an edge labeled with $\prec$; and

(2) for each edge $(a, \neg \prec, b)$ of Q_U, there do not exist two distinct maximal strongly connected components of Q'_U, one containing state a and the other one containing state b such that there is path from a to b in Q'_U.

Since the maximal strongly connected components of Q'_U can be constructed in time linear in the size of Q_U, viz., linear in $|U|$, therefore the above mentioned conditions 1 and 2 can be checked in time quadratic in the size of U. Thus we can decide in O($|U|^2$) time whether a desired pre-order can be imposed on S or not.

4 Applications

As applications, we consider model checking parameterized versions of all of the snoop based cache protocols presented in [19]. The translation from the state transition diagram of a given protocol to its template is straightforward and syntactic and can be performed in the same mechanical fashion as was done for the MSI protocol in section 2.1: Firing a bold transition labeled with $A/-$ and/or one that requires that no other cache currently possesses the desired memory block does not affect the status of the memory block in any other cache. Such a transition is therefore labeled with the local transition label τ and in the second case also guarded with the $\bigwedge(\mathsf{i})$. Otherwise, a transition labeled by A/B, where $B \neq -$, is labeled with the broadcast send label $A!!$

[4] Flush broadcast send transitions can be identified syntactically as all their matching receives from every non-initial state transit to a unique state with the matching receive from i self-looping on itself. Local transitions can be identified by the absence of matching receives.

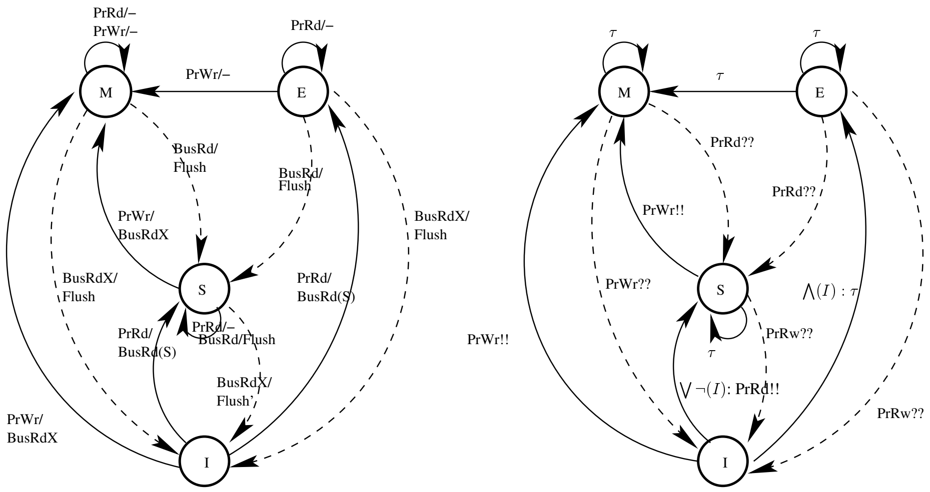

Fig. 4. The Illinois MESI Cache Coherence Protocol and its template

while every transition generated by bus actions (represented by dashed lines) and labeled with B/C is labeled with the matching broadcast receive label $A??$. If to fire the transition additionally requires some other cache to possess the desired memory block then it is also guarded by $\bigvee \neg(\mathsf{i})$. Below we consider only the *Illinois MESI* protocol in detail, with some others being handled in the full report [10].

4.1 The Illinois MESI Cache Coherence Protocol

The transition digram and the template for the Illinois MESI cache coherence protocol is shown in figure 4. Formally the template is defined as $U = (S, \Sigma, R, \{\mathsf{i}\})$ where, $S = \{\mathsf{I}, \mathsf{S}, \mathsf{E}, \mathsf{M}\}$ with the pre-order being given by $\mathsf{I} \prec \mathsf{S} \prec \mathsf{E} \approx \mathsf{M}$. The set $\Sigma = \{\tau, \mathsf{PrRd!!}, \mathsf{PrRd??}, \mathsf{PrWr!!}, \mathsf{PrWr??}\}$. The transitions are as defined below.

Empty Broadcasts (Local Transitions): $(\mathsf{M}, \tau, \mathsf{I})$, $(\mathsf{E}, \tau, \mathsf{I})$, $(\mathsf{S}, \tau, \mathsf{I})$, $(\mathsf{I}, \tau, \mathsf{E})$, $(\mathsf{S}, \tau, \mathsf{S})$, $(\mathsf{E}, \tau, \mathsf{E})$, $(\mathsf{M}, \tau, \mathsf{M})$. Note that the first three transitions are included because of the assumption of initializability and are for simplicity reasons not shown in figure 4 nor are broadcast receive transitions that are self loops.

Low-push sends: $(\mathsf{I}, \mathsf{PrRd!!}, \mathsf{S})$.

Low-push receives: $(\mathsf{M}, \mathsf{PrRd??}, \mathsf{S})$, $(\mathsf{E}, \mathsf{PrRd??}, \mathsf{S})$.

Flush sends: $(\mathsf{I}, \mathsf{PrWr!!}, \mathsf{M})$, $(\mathsf{S}, \mathsf{PrWr!!}, \mathsf{M})$.

Flush receives: $(\mathsf{M}, \mathsf{PrWr??}, \mathsf{I})$, $(\mathsf{E}, \mathsf{PrWr??}, \mathsf{I})$, $(\mathsf{S}, \mathsf{PrWr??}, \mathsf{I})$.

The transitions $(\mathsf{I}, \mathsf{PrRd!!}, \mathsf{S})$ and $(\mathsf{I}, \tau, \mathsf{E})$ are labeled with $\bigvee \neg(\mathsf{i})$ and $\bigwedge(\mathsf{i})$ respectively, with the rest of the transitions being labeled with the *true* guard.

We need to decide whether the following pairs are pairwise reachable: (M, M), (M, E), (M, S), (E, E), (E, S).

4.2 Experimental Results

Here we summarize the results for a wide range of examples of cache coherence protocols. For detailed descriptions of these protocols refer to [19]. The column under *# of Abstract States* refers to the number of reachable states in the abstract history graph for protocols that don't use conjunctive guards, viz., MSI, MESI, MOESI, Berkeley and N+1; and in the modified abstract history graph for ones that use conjunctive guards, viz., Illinois-MESI, Firefly and Dragon. It is worth noting that although in the worst case the number of reachable abstract states in the modified abstract history graph corresponding to the template $U = (S, R, \Sigma, \mathrm{i})$ could be as large as $|S|2^{|S|}$, in practice it typically turns out to be much smaller. For instance in the MESI protocol, the number of reachable abstract states were 6, against a worst case possibility of $4 \times 2^4 = 64$ states. A similar scenario holds for the other protocols. Thus, in conclusion, the abstract history graph construction seems to work well in practice. The experiments were carried out on a machine with a 797MHz Intel Pentium III processor and 256 Mb RAM. Below, we tabulate the results for a variety of cache coherence protocols. The user time for verifying each of the cache coherence protocols was less than 0.01 seconds.

Protocol	*Pre-Order*	*# of Abstract States*
MSI	*Invalid* $\prec$ *Shared* $\prec$ *Modified*	5
MESI	*Invalid* $\prec$ *Shared* $\prec$ *Exclusive* $\approx$ *Modified*	6
Illinois	*Invalid* $\prec$ *Shared* $\prec$ *Exclusive* $\approx$ *Modified*	6
MOESI	*Invalid* $\prec$ *Owned* $\approx$ *Shared* $\prec$ *Exclusive* $\approx$ *Modified*	7
N+1	*Invalid* $\prec$ *Valid* $\prec$ *Dirty*	5
Berkeley	*Invalid* $\prec$ *Owned Non-exclusively* $\approx$ *Unowned*; *Unowned* $\prec$ *Owned Exclusively*	5
Firefly	*Invalid* $\prec$ *Shared* $\prec$ *Dirty* $\approx$ *Valid Exclusive*	6
Dragon	*Invalid* $\prec$ *Shared Clean* $\approx$ *Shared Modified* $\prec$ *Exclusive*; *Exclusive* $\approx$ *Modified*	8

5 Concluding Remarks

The generally undecidable PMCP has received a good deal of attention in the literature. A number of interesting proposals have been put forth, and successfully applied to certain examples ([7, 6, 26, 20, 2, 3, 27, 21]). Most of these works, however, suffer from the drawbacks of being either only partially automated or being sound but not guaranteed complete. Much human ingenuity may be required to develop, e.g., network invariants; the method may not terminate; the complexity may be intractably high; and the underlying abstraction may only be conservative, rather than exact. [5]

Similar limitations apply to prior work on PMCP for cache protocols. Pong and Dubois [25] described methods that were sound but not complete, as they were based on

[5] However for frameworks that handle specialized applications domains decisions procedures can be given that are both sound and complete and fully automatic and in some cases efficient ([13, 15, 11, 12, 5, 24]).

conservative, inexact abstractions. In [14] a general framework of parameterized *broadcast protocols* was introduced and it was shown how certain simple cache protocols could be modeled. That framework, however, did not admit guarded transitions, necessary to model many cache protocols such as Illinois (MESI). In [16], it was shown that showed that PMCP for safety over such broadcast protocols of [14] is decidable using the general backward reachability procedure of [1]. However, the backward reachability algorithm of [1] that [16], makes use of, although general, suffers from the handicap that the best known bound for its running time is not known to be primitive recursive [23]. In [22], Maidl, using a proof tree based construction, shows decidability of the PMCP for a broad class of systems including broadcast protocols, but again the decision procedure is not known to be primitive recursive. Moreover [22, 16, 14] do not report experimental results for cache protocols.

More recently, Delzanno [9] uses arithmetical constraints to model global states of systems with many identical caches. This method uses invariant checking via backward reachability analysis of [1] and provides a broad framework for reasoning about cache coherence protocols but the procedure does not terminate on some examples. Furthermore, this technique does not provide a way to generate *error traces* when a bug is detected. In [17], it was shown that for a sub class of broadcast protocols called *entropic* broadcast protocols, a generalization of the Karp-Miller procedure for Petri nets terminates. While mathematically elegant, the model does not allow for boolean guards necessary for modeling protocols like Illinois-MESI, Firefly and Dragon. Also, no explicit bounds were provided on the size of the resulting coverability tree (cf. [23]).

In this paper we have exploited the hierarchical organization inherent in the design of snoopy cache protocols, representing and generalizing this organization using preorders. We then present a specialized variant of the broadcast protocols model called *pre-ordered protocols* tailored to capture snoopy cache coherence protocols. This has allowed us to provide a unified, fully automated and efficient method to reason about parameterized snoopy cache coherence protocols. Our method is unique in meeting all these important criteria: (a) it is sound and complete; (b) it is algorithmic; (c) it is *rapid* meaning reasonably efficient in principle: worst case complexity single exponential. (d) it has broad modeling power: handles all 8 examples from Handy's book; (e) it is *rapid* also meaning demonstrably efficient in experimental practice; each example protocol was verified — for parameterized correctness — in a fraction of a second; and (f) it caters for error trace recovery.

References

1. P. Abdulla, K. Cerans, B. Jonsson, Y. K. Tsay. General Decidability Theorems for Infinite State Systems. *LICS*. 1996.
2. P. Abdulla, A. Boujjani, B. Jonsson and M. Nilsson. Handling global conditions in parameterized systems verification. CAV 1999.
3. P. Abdulla and B. Jonsson. On the existence of network invariants for verifying parameterized systems. In *Correct System Design - Recent Insights and Advances*, 1710, LNCS, pp 180-197, 1999.
4. K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 15, pages 307-309, 1986.

5. T. Arons, A. Pnueli, S. Ruah, J, Xu and L. Zuck. Parameterized Verification with Automatically Computed Inductive Assertions. CAV 2001, LNCS 2102, 2001.
6. M.C. Browne, E.M. Clarke and O. Grumberg. Reasoning about Networks with Many Identical Finite State Processes. *Information and Control*, 81(1), pages 13-31, April 1989.
7. E.M. Clarke, O. Grumberg and S. Jha. Verifying Parameterized Networks using Abstraction and Regular Languages. CONCUR. LNCS 962, pages 395-407, Springer-Verlag, 1995.
8. D. E. Culler and J. P. Singh. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers, 1998.
9. G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. CAV 2000, 51-68.
10. E.A. Emerson and V. Kahlon. This paper, full version. Available at `http://www.cs.utexas.edu/users/{emerson,kahlon}/tacas03/`
11. E.A. Emerson and V. Kahlon. Reducing Model Checking of the Many to the Few. CADE-17. LNCS , Springer-Verlag, 2000.
12. E.A. Emerson and V. Kahlon. Model Checking Large-Scale and Parameterized Resource Allocation Systems. TACAS, 2002.
13. E.A. Emerson and K.S. Namjoshi. Reasoning about Rings. POPL. pages 85-94, 1995.
14. E.A. Emerson and K.S. Namjoshi. On Model Checking for Non-Deterministic Infinite-State Systems. LICS 1998.
15. E.A. Emerson and K.S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems. CAV. LNCS , Springer-Verlag, 1996.
16. J. Esparza, A Finkel and R. Mayr, On the Verification of Broadcast Protocols. LICS 1999.
17. A. Finkel and J. Leroux. A finite covering tree for analyzing entropic broadcast protocols. Proc. VCL 2000. Report DSSE-TR-2000-6, Univ. Southampton, GB.
18. S.M. German and A.P. Sistla. Reasoning about Systems with Many Processes. *J. ACM*, 39(3), July 1992.
19. J. Handy. The Cache Memory Book. Academic Press, 1993.
20. R. P. Kurshan and K. L. McMillan. A Structural Induction Theorem for Processes. PODC. pages 239-247, 1989.
21. D. Lesens, N. Halbwachs and P. Raymond. Automatic Verification of Parameterized Linear Network of Processes. POPL 1997. pp 346-357, 1997.
Parallel Coordination Programs I.*Acta Informatica 21*, 1984.
22. M. Maidl. A Unifying Model Checking Approach for Safety Properties of Parameterized Systems. CAV 2001.
23. K. McAloon. Petri Nets and Large Finite Sets. *Theoretical Computer Science 32*, pp. 173-183, 1984.
24. A. Pnueli, S. Ruah and L. Zuck. Automatic Deductive Verification with Invisible Invariants. TACAS 2001, LNCS, 2001.
25. F. Pong and M. Dubois. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 8, August 1995.
26. A. P. Sistla, Parameterized Verification of Linear Networks Using Automata as Invariants, CAV, 1997.
27. P. Wolper and V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In J. Sifakis(ed) *Automatic Verification Methods for Finite State Systems*, Springer-Verlag, LNCS 407, 1989.