Exact and Efficient Verification of Parameterized Cache Coherence Protocols

E. Allen Emerson and Vineet Kahlon

Department of Computer Sciences and Computer Engineering Research Center*
The University of Texas, Austin TX-78712, USA

Abstract. We propose new, tractably (in some cases provably) efficient algorithmic methods for exact (sound and complete) *parameterized* reasoning about cache coherence protocols. For reasoning about general snoopy cache protocols, we introduce the *guarded broadcast protocols* model and show how an *abstract history graph* construction can be used to reason about safety properties for this framework. Although the worst case size of the abstract history graph can be exponential in the size of the transition diagram of the given protocol, the actual size is small for standard cache protocols as is evidenced by our experimental results. The framework can handle all 8 of the cache protocols in [19] as well as their split-transaction versions. We next identify a framework called *initialized broadcast protocols* suitable for reasoning about *invalidation-based* snoopy cache protocols and show how to reduce reasoning about such systems with an arbitrary number of caches to a system with at most 7 caches. This yields a provably polynomial time algorithm for the parameterized verification of invalidation based snoopy protocols. Our results apply to both safety and liveness properties. Finally, we present a methodology for reducing parameterized reasoning about directory based protocols to snoopy protocols, thus leveraging techniques developed for verifying snoopy protocols to directory based ones, which are typically are much harder to reason about. We demonstrate by reducing reasoning about a directory based protocol suggested by German [17] to the ESI snoopy protocol, a modification of the MSI snoopy protocol.

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. Email addresses: {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, but of great practical importance, which has led to many heuristics and algorithms for particular cases. In this paper, we present new, tractably (in some cases provably) efficient algorithmic methods for exact parameterized reasoning about cache coherence protocols.

First, for reasoning about general snoopy cache protocols, we introduce the *guarded broadcast protocols* model wherein processes coordinate using broadcast primitives plus boolean guards. A broadcast transmission corresponds to a cache 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-MESI, Firefly, Dragon) that need to determine the presence or absence of the required memory block in other caches. We show how an *abstract history graph* construction can be used to reason about safety properties of guarded broadcasts. In the construction, a path x leading to global state s is represented as a tuple of the form $(a, A) \in S \times 2^S$, where S is the set of local states of the given cache protocol, that reflects not merely the local states present in s but also takes into account the local transitions that were fired along x to get to s, viz., the *history* of s along x. The extra historical information, that our construction stores, permits us to reason about safety properties for an arbitrary number of caches in an *exact* fashion as opposed to the standard abstract graph construction [24] that only takes into account the set of local states present in s and is thus sound but not guaranteed complete. We establish a path correspondence between concrete computations of the original system and paths in the abstract graph which also allows us to *automatically* generate error traces once an erroneous 'abstract state' is detected. 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, thus enabling us to reason about the more expressive framework of guarded broadcast for the same worst case time complexity as ordered broadcasts. In practice, however, the abstract graph tends to be small as is documented by our empirical results.

Next we consider the PMCP for *invalidation based* snoopy protocols, viz., protocols that on a write operation invalidate the memory block being written to in all other caches of the given system [7]. We model such protocols using the new framework of *initialized broadcast protocols*. For this model, we consider the PMCP for formulae of the form $\bigwedge_{i \neq j} \mathsf{A}h(i,j)$ and $\bigwedge_{i \neq j} \mathsf{E}h(i,j)$, where $h(i,j)$ is a LTL$\backslash$X formula over a pair of distinct processes. For such formulae, we show how to reduce reasoning for a system with an arbitrary number of processes to systems with at most a *cutoff* (in fact 7) number of processes. This yields a provably polynomial time algorithm (in the size of the state diagram of a single cache unit) for reasoning about the PMCP for a broad class of linear time properties of invalidation-based protocols, not just safety. Also the use of cutoffs has the important advantage that the large system with, say 100 caches, is very much like the small system with 7 caches. This provides a simple reduction from n to 7 processes that automatically caters to error recovery.

Finally, we consider the PMCP for directory-based protocols wherein information regarding cache states of individual memory blocks is stored in a centralized directory and all transaction regarding cache state lookup, invalidations, updates etc., take place across a network. We use the observation that for most directory based protocols there exists a snoopy protocol with exactly the same states [7] and running essentially the same protocol except that the implementation of each snoopy broadcast transition is broken up into several steps. Since the executions of steps corresponding to different snoopy broadcasts can interleave among themselves, it makes directory-based protocols behaviorally more complex and thus seemingly harder to verify than their snoopy counterparts. However, we demonstrate, using a directory based protocol suggested by German [17], that since all transactions are serviced via the *centralized* directory, it leads to a serialization of steps of snoopy broadcasts in a way that there is limited overlap among steps corresponding to different snoopy broadcasts. We can then establish path correspondences between computation paths of directory based protocols and their snoopy counterparts thereby allowing us to reduce the PMCP for linear time properties from directory based protocols to snoopy ones. Thus techniques developed for reasoning about parameterized snoopy broadcasts can now be leveraged. As an example, we show how to reduce reasoning about this directory based protocol to the ESI snoopy protocol, a modification of the MSI protocol, which was verified using the abstract history graph construction in less than 0.01 secs.

The rest of the paper is organized as follows. We begin by introducing the system model in section 2. In section 3, we present the abstract history graph construction for verifying safety properties of guarded broadcast protocols while cutoff results for initialized broadcast protocols are given in section 4. In section 5, we demonstrate, using the protocol suggested by German, how to reduce reasoning about directory based protocols to snoopy protocols. Applications and experimental results are given in section 6 and we end with some concluding remarks in section 7.

2 The System Model

We consider systems of the form, U^n, comprised of finite, but arbitrarily many, copies of a process *template*, U, executing asynchronously with interleaving semantics. The template U is formally defined by the 4-tuple $U = (S, \Sigma, R, \mathsf{i})$, where S is a finite, non-empty set of *states*; Σ is a finite set of *labels* including the internal transition label τ, and broadcast send and receive labels of the form $l!!$ and $l??$, respectively; R is the transition relation; and i the initial state. Each transition of R is either an *internal* transition of the form $a \xrightarrow{g:\tau} b$, a broadcast send of the form $a \xrightarrow{g:l!!} b$, or a broadcast receive of the form $a \xrightarrow{g:l??} b$, where g is a boolean guard.

We assume that receives are deterministic, viz., for each label $l!!$ appearing in some broadcast send and for each state a in S, there is a unique corresponding receive transition on $l??$ out of a. The guard g labeling a transition tr of R is either the boolean expression *true* or the *specialized conjunctive* guard $\bigwedge(\mathsf{i})$, or the *specialized disjunctive* guard $\bigvee\neg(\mathsf{i})$, where i is the initial state of U. 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.

To capture block replacement behavior, we also require that templates be *initializable*. [1] This means that from each state a of a protocol, there is an unguarded, internal 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 block. For simplicity, re-initialization transitions and self-loop receptions are not drawn in state transition diagrams of cache protocols (cf. [7]).

We now introduce the following frameworks (a) *Initialized Broadcast Protocols* for dealing with invalidation based snoopy protocols, and (b) *Guarded Broadcast Protocols* for dealing with general snoopy cache protocols, by specifying the types of broadcast transition allowed. The two frameworks are incomparable in that each framework can model a protocol that the other cannot.

Initialized Broadcast Protocols There are two major classes of snoopy cache protocols: *update based* and *invalidation based*. In update based protocols, e.g., *Dragon* and *Firefly*, whenever a shared location is written to by a processor, its value is updated in the caches of all other processors holding that memory block without invalidating the block. In contrast, with invalidation based protocols, e.g, *MESI* and *Berkeley*, on a write operation the memory block being written to is invalidated in all other caches [7]. In this paper, we model invalidation-based protocols using the framework of *Initialized Broadcast Protocols* wherein, each broadcast transition of U is either an (a) i-*flush*: transition $a \xrightarrow{l!!} b$ is called an i-flush iff from each state c of U there is the (unique) matching receive $c \xrightarrow{l??} \mathsf{i}$, or (b) *initialized-broadcast*: transition $tr = a \xrightarrow{l!!} b$ is an initialized broadcast send transition provided that $a = \mathsf{i}$ and every matching reception transition for tr is of the form $c \xrightarrow{l??} d$, where either both $c, d \neq \mathsf{i}$ or both $c, d = \mathsf{i}$.

Guarded Broadcasts In *Guarded Broadcasts*, each broadcast transition tr is of either of the two forms (a) *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. (b) *Push*: Transition $a \xrightarrow{l!!} b$, where $b \neq \mathsf{i}$, is a *push* transition provided that there exists the matching receive transitions $\mathsf{i} \xrightarrow{l??} \mathsf{i}$, $a \xrightarrow{l??} a$ and $b \xrightarrow{l??} b$ in R and for every path $c \xrightarrow{l??} d \xrightarrow{l??} e$, we have $d = e$.

In either framework, given U, the state transition digram for $U^n = (S^n, \Sigma, R^n, \mathsf{i}^n)$, the system with n copies of U, is based on interleaving semantics in the standard way. We write $x.s \in U^n$ to mean that finite computation path x of U^n ends in global state s. For local state a, $num(a, s)$, denotes the number of copies of local state a in s.

The template U for a protocol, such as MSI (figure 1), 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 and tantamounts to relabeling the transitions of the given template to illustrate the link between broadcast sends and their matching receives.

Safety Properties 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

[1] Initializability is not needed for the results in section 3.1.

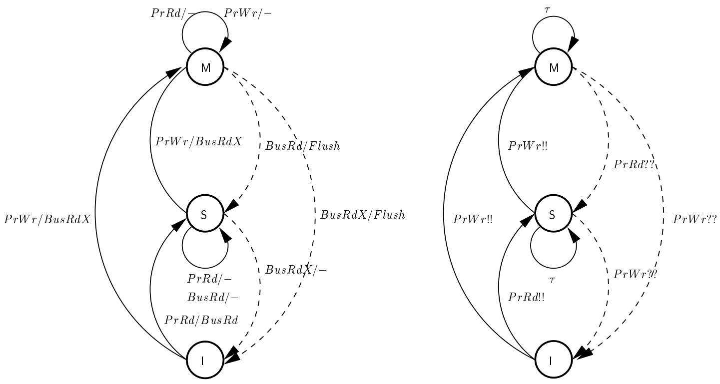

Fig. 1. The MSI Cache Coherence Protocol and its template

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, viz., $U^n \models \bigvee_{i \neq j} \mathsf{EF}(a_i \wedge b_j)$. 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 Model Checking Guarded Broadcasts for Safety Properties

In a split-transaction bus, each transaction is split into two independent sub-transactions: a *request* transaction and a *response* transaction. Other transactions (or sub-transactions) are allowed to interfere (interleave) between them so that the bus can be used while response to the original request is being generated. The advantage is a more effective utilization of the bus. To deal with the non-atomic nature of bus transactions, extra states called transient states are introduced in the state transition diagram of split-transaction based protocols to indicate outstanding bus requests. This however makes snoopy split transaction bus protocols harder to reason about than their 'non-split' counterparts. We now show how to reason about guarded broadcasts, which can model all snoopy protocols in [19] and their split transaction bus versions, using an abstract history graph construction.

3.1 Protocols without conjunctive guards. In this section, we consider guarded broadcasts wherein template U does not have conjunctive guards; but guards of the form *true* or $\bigvee \neg(\mathsf{i})$ 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, and their split-transaction versions.

We motivate our technique with the help of an example. Consider the computation $x = (\mathsf{I}, \mathsf{I}) \rightarrow (\mathsf{I}, \mathsf{S})$ of the system, U^2, comprised of two caches running the MSI protocol. We exploit the observation that we can pump up the multiplicity of each of the local states I, S to be greater than or equal to any arbitrary number n, by firing the transition $\mathsf{I} \stackrel{PrRd!!}{\longrightarrow} \mathsf{S}$ successively n times as shown $(\underbrace{\mathsf{I}, ..., \mathsf{I}}_{2n}) \stackrel{PrRd_1}{\longrightarrow} ... \stackrel{PrRd_n}{\longrightarrow} (\underbrace{\mathsf{S}, ..., \mathsf{S}}_{n}, \underbrace{\mathsf{I}, ..., \mathsf{I}}_{n})$.

On the other hand, consider the computation $y = (\mathsf{I}, \mathsf{I}) \to (\mathsf{I}, \mathsf{M})$. We cannot pump up the multiplicity of local state M, because in order for that to happen, we need to fire the transition $tr = \mathsf{I} \stackrel{PrWr!!}{\longrightarrow} \mathsf{M}$ repeatedly. But a process firing tr, a flush transition, clobbers every other process by forcing it into its initial state. Thus we can have at most one copy of M in any global state.

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 $A = \{s[j] | j \in [1 : n] \wedge j \neq i\}$.

Then the above discussion can be formalized as the following *unbounded pumping property* implicitly shown in the proof of proposition 3.1. Let computation path $x.s \in U^n$ be such that *rep*$(x.s) = (a, A)$. Then given a positive integer p, there exists $y.t \in U^m$, for some m, such that *rep*$(y.t) = (a, A')$, where $A \subseteq A'$ and for each $a' \in A'$, $num(a', t) \geq p$. Thus we can represent $x.s$ by the tuple $(a, A') \in S \times 2^S$, representing a *formal* state with (at least) one copy of a and arbitrarily many copies of each state in A'. Given template U, we now define the *abstract history graph*, $\mathcal{A}_U = (\mathcal{S}_U, \mathcal{R}_U, (\mathsf{i}, \{\mathsf{i}\}))$, as a transition diagram over tuples in $\mathcal{S}_U = S \times 2^S$ that captures the behaviour of a system instance of arbitrary size. To define the transition relation $\mathcal{R}_U$, given a tuple (a, A) and an internal or a broadcast send transition $tr = c \to d$, we introduce the notion of the successor of (a, A) via tr as either the *1-successor*, which covers the scenario when a process in local state a, that (possibly) has multiplicity one fires tr; or the *2-successor* of (a, A), covering the scenario when a process in one of the states in A each of which can be thought of as having arbitrarily large multiplicity fires tr.

Definition (1-successor) Let $(a, A) \in S \times 2^S$ and let transition $tr = a \to b \in R$ 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 $succ_1((a, A), tr) = (b, B)$, where if tr is an internal 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* $\}$.

Definition (2-successor) Let $(a, A) \in S \times 2^S$ and let transition $tr = b \to c \in R$, 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, $succ_2((a, A), tr)$, is defined as the tuple

- $(c, \{c'\} \cup \{\mathsf{i}\})$ if tr is a c'-flush transition
- $(a, A \cup \{c\})$ if tr is an internal transition. Note that since we had arbitrarily many copies of b to start with so even after firing internal 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 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 $\}$. 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 push transition is b itself.

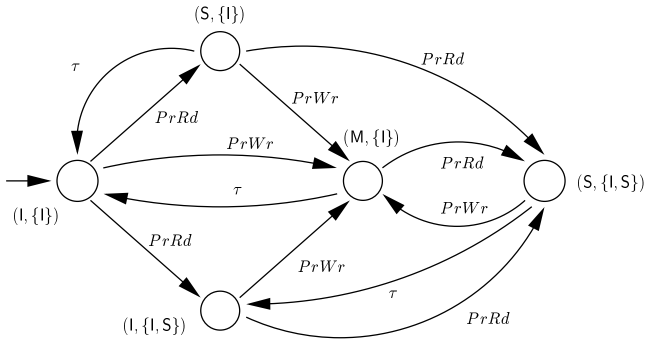

Fig. 2. The abstract history graph for the MSI Cache Coherence Protocol

Definition (Abstract History Graph) Given template $U = (S, \Sigma, R, \mathsf{i})$, the *abstract history graph* of U, is defined as $\mathcal{A}_U = (\mathcal{S}_U, \mathcal{R}_U, (\mathsf{i}, \{\mathsf{i}\}))$, where $\mathcal{S}_U = S \times 2^S$ and $\mathcal{R}_U = \{((a, A), (b, B)) | (b, B) = succ_1((a, A), tr)$ *or* $(b, B) = succ_2((a, A), tr))$ *for some internal or broadcast send transition* tr *of* $U\}$.

As an example, the abstract history graph for the MSI protocol is shown in figure 2. 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. We now establish a 'path correspondence' between finite computations of U^n and finite paths of $\mathcal{A}_U$ starting at $(\mathsf{i}, \{\mathsf{i}\})$. Let $(a, A) \geq (b, B)$ denote $a = b$ and $B \subseteq A$.

Proposition 3.1 (Covering Projection) *For any n and 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 $t \geq rep(x.s)$.*

The tuple t not only stores the set of local states present in s, but also the states that could potentially be present in a global state of a system with sufficiently many copies of U that results by firing (a stuttering) of the same sequence of transitions as were fired along x to get to s. Thus t drags along some 'history' of computation x leading to s and thereby stores more information than $rep(x.s)$.

Proposition 3.2 (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 two results, we have

Theorem 3.3 (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 $\mathcal{A}_U$. In the worst case, the size of the abstract graph is O($|U|2^{|U|}$), however, we need only consider the set of tuples reachable from $(\mathsf{i}, \{\mathsf{i}\})$ which, in practice, is much smaller (cf. section 6).

Corollary 3.4 *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.*

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 modify the abstract history graph by adding for every tuple (a, A), a transition of the form $(a, A) \rightarrow (a', \{\mathsf{i}\})$, where either $a' = a$ or $a' \in A$, to $\mathcal{A}_U$. 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 initializing internal transition. Then, path correspondences as in section 3.1 can be shown and so, pairwise reachability can be decided in time O($|U|2^{|U|}$), where $|U|$ is the size of U. Examples include the Illinois-MESI, Dragon and Firefly protocols and their split-transaction versions.

4 Reasoning about Invalidation based Protocols using Cutoffs

In this section, we consider the PMCP for formulae of the form $\bigwedge_{i \neq j} \mathsf{A}h(i, j)$ and $\bigwedge_{i \neq j} \mathsf{E}h(i, j)$, where $h(i, j)$ is a LTL$\backslash$X formula over the local states of U_i and U_j. We show how to reduce reasoning about a system with an arbitrary number of processes (caches) to a system with up to a *cutoff* (in fact 7) number of processes. This immediately yields a polynomial time algorithm for the PMCP at hand. The use of cutoffs has several advantages. First, the small system with a cutoff number of processes is identical to the large system, but with a fewer number of processes, and thus there is no need to construct, for instance, an abstract graph that may have a complex, non-obvious structure. Secondly, it automatically caters to error trace recovery. We later show how to reduce reasoning about LTL$\backslash$X properties from directory-based to snoopy protocols for which these results can be leveraged.

We now present the cutoff result for properties of the form $\bigwedge_{i \neq j} \mathsf{E}h(i, j)$. Since all processes in the systems we consider are copies of a single template U, they are all isomorphic up to renaming. Therefore symmetry considerations dictate that $U^n \models \mathsf{E}h(1, 2)$ iff for each pair i, j, where $i \neq j$, $U^n \models \mathsf{E}h(i, j)$. We shall therefore concentrate only on the formulae $\mathsf{A}h(1, 2)$ and $\mathsf{E}h(1, 2)$.

Proposition 4.1 (Cutoff Result for Finite Paths) *For all $n \geq 7$, $U^n \models \mathsf{E}_{\mathsf{fin}}h(1, 2)$ iff $U^7 \models \mathsf{E}_{\mathsf{fin}}h(1, 2)$, where $\mathsf{E}_{\mathsf{fin}}$ quantities over finite paths only.*

Proof Sketch We present the main ideas behind the proof. The proof of the cutoff result proceeds by establishing a stuttering path correspondence between U^n, where $n \geq 7$, and U^7, viz., constructing a finite stuttering computation path y of U^7 corresponding to a given finite path x of U^n that preserves the local computation paths of processes U_1 and U_2, modulo stuttering, and vice versa.

($\Rightarrow$) Given a finite computation x of U^n, where $n \geq 7$, we show how to construct a finite computation y of U^7 that preserves the local computations of processes U_1 and U_2, modulo stuttering. Towards that end, we parse (the transitions of) x as $x = N_0 I_0 ... I_m N_{m+1}$, where I_i is the ith global transition to be executed along x that results by firing either an i-flush or a transition labeled with $\bigwedge(\mathsf{i})$. Thus N_is are strings of transitions whereas I_is are single transitions. The construction of y proceeds by constructing for each subsequence N_iI_i, a corresponding subsequence $N_i^\prime I_i^\prime$ by projecting onto the local subsequences of N_iI_i of a set P_i of process indices defined below.

In defining P_i, there are two main considerations (a) every projected broadcast receive has a matching send, and (b) the specialized disjunctive guard is true for every projected local transition (the conjunctive guard $\bigwedge(\mathsf{i})$ is automatically true for all projected transitions). Clearly, we need to project on to process indices 1 and 2 as we have to preserve the local computation sequences of U_1 and U_2 modulo stuttering. Also, we need to project onto indices p_3 and p_4 of the processes responsible for firing the solitary global transitions in I_{i-1} and I_i, respectively. Projection on to index p_3 ensures 'continuity' of the local computation of the process responsible for firing the global transition constituting $I_{i-1}^\prime$, while projection on to index p_4 guarantees that every projected receive transition in $I_i^\prime$ has a matching send in $I_i^\prime$. Finally, let $N_i = x_{i'}...x_{i'+l}$ and let a and b be, respectively, the least and second least among all integers $c \in [0:l]$ having the property that $x_{i'+c}[p] \neq \mathsf{i}$, for some $p \in [1:n] \setminus (\{1,2\} \cup \{p_3\} \cup \{p_4\})$. To ensure that the specialized disjunctive guard is true for the projected transitions, we include the indices p_5 and p_6 in P_i, where $x_{i'+a}[p_5] \neq \mathsf{i}$ and $x_{i'+b}[p_6] \neq \mathsf{i}$. Then, we let $P_i = \{1,2\} \cup \{p_3\} \cup \{p_4\} \cup \{p_5\} \cup \{p_6\}$. A seventh process with index p_7, say, is required to ensure that in $N_i^\prime$, every projected initialized broadcast receive transition has a matching broadcast send. Since, by definition, an initialized broadcast send is fired only from the initial state, we use this process, which we (try to) maintain in its initial state i, to fire the required send transition and then 'recycle' it by firing the initializing internal transition to make it transit back to i. The computation y, then results by 'sewing' up the subsequences $N_i^\prime I_i^\prime$ appropriately, in the same relative order as the original subsequences N_iI_i along x. Note that the sets P_i may be different for different i; however, since all processes in our system are isomorphic up to renaming, for each i, U^7 can mimic the local sub-computations of $N_i^\prime I_i^\prime$.

($\Leftarrow$) The *lifting* part is simpler. Given a computation y of U^7, we can construct a valid computation x of U^n, where $n \geq 7$, by letting processes $U_1, ..., U_7$ execute exactly the same local computations as in y while the rest of the processes just stutter in their initial states without executing any non-receive transition at all (all receives from i loop back to i). □

The proof technique of proposition 4.1, extends to the case where we consider full paths (and full paths under the assumption of unconditional fairness). We then have the following.

Proposition 4.2 (Cutoff Result for Full Paths) *For all $n \geq 7$, $U^n \models \mathsf{E}h(1,2)$ iff $U^7 \models \mathsf{E}h(1,2)$, where $h(i,j)$ is a LTL\X formula over processes U_i and U_j.*

As a corollary to propositions 4.1 and 4.2, we have the following.

Proposition 4.3 (Efficient Decidability Result) *For initialized broadcast protocols, the PMCP for formulae of the types $\bigwedge_{i \neq j} \mathsf{E_{fin}}h(i,j)$, $\bigwedge_{i \neq j} \mathsf{A_{fin}}h(i,j)$, $\bigwedge_{i \neq j} \mathsf{E}h(i,j)$ and $\bigwedge_{i \neq j} \mathsf{A}h(i,j)$ is decidable in polynomial time in the size of the template U specifying the parameterized family.*

5 Reducing PMCP for Directory Based to Snoopy Protocols

In this section, we present a methodology for reducing the PMCP for (stuttering insensitive) LTL$\backslash$X properties for directory based to snoopy cache protocols thereby enabling us to leverage the techniques developed for snoopy protocols. We exploit the observation that with most directory based protocols one can associate a snoopy protocol with exactly the same local states [7] and executing essentially the same protocol except that the implementation of each snoopy broadcast transition is broken down into several smaller steps that execute asynchronously. We call such transitions *distributed broadcasts*. The interleavings of the steps of different distributed broadcasts makes directory based protocols behaviorally more complex than their snoopy counterparts and thus seemingly harder to reason about. However, the central directory can service only one distributed broadcast at a time, and so in a given computation, x, of the system, $U^n_{\mathsf{Directory}}$, comprised of n caches running the directory based protocol Directory, there is a unique serial order on the way distributed broadcasts are serviced along x. This allows us to construct a computation y of U^n_{Snoop}, where Snoop is the snoopy protocol corresponding to Directory, by letting the snoopy broadcast transitions fire in the same linear order as their distributed counterparts were serviced along x. This path correspondence allows us to reduce reasoning about linear time properties from directory based to snoop based protocols. We demonstrate our technique using a directory based protocol suggested by German [17], which we denote by DIR.

Reasoning about the DIR Directory Based Protocol. In the DIR protocol, each cache is represented as a $client$ process with the directory being represented as the $Home$ process. The variables used in DIR are given below.

type message = $\{empty, req\_shared, req\_exclusive, invalidate,$
$invalidate\_ack, grant\_shared, grant\_exclusive\}$
type cache_state = $\{invalid, shared, exclusive\}$
`channel1`, `channel2_4`, `channel3` : array[1:n] of message
`home_sharer_list`, `home_invalidate_list`: array[1:n] of boolean
`home_exclusive_granted` : boolean
`home_current_command`: message
`home_current_client`: $[1{:}n]$
`cache`: array$[1{:}n]$ of cache_state

Each client has three possible local states, viz., $invalid$, $shared$ and $exclusive$, represented by the variable $cache\_state$. Communication between $client[i]$, the process representing the ith cache, and $Home$, the process representing the directory, takes place via the following variables that are shared pairwise between $client[i]$ and $Home$.

- $channel1[i]$: used by $client[i]$ to request the memory block in the $shared$ or the $exclusive$ state.
- $channel2\_4[i]$: used by $Home$ to send the *invalidation* message or grant (shared or exclusive) access to the memory block request by $client[i]$.
- $channel3[i]$: used by $client[i]$ to send acknowledgements for invalidation requests by $Home$.

3: $(\texttt{h\_c\_co} = empty \wedge \neg(\texttt{ch1}[cl] = empty))$ $\to \texttt{h\_c\_co} := \texttt{ch1}[cl]; \texttt{ch1}[cl] := empty; \texttt{h\_c\_cl} := cl;$ for $i : [1 : n]$ do $\texttt{ho\_in\_l}[i] := \texttt{ho\_sh\_l}[i]$ endfor;
4: $((\texttt{h\_c\_co} = req\_shared \wedge \texttt{heg} \vee \texttt{h\_c\_co} = req\_exclusive) \wedge \texttt{h\_in\_l}[cl] \wedge$ $\texttt{ch2\_4}[cl] = empty)$ $\to \texttt{ch2\_4}[cl] := invalidate; \texttt{h\_in\_l}[cl] := false$
5: $(\neg(\texttt{h\_c\_co} = empty) \wedge \texttt{ch3}[cl] = invalidate\_ack)$ $\to \texttt{h\_sh\_l}[cl] := false; \texttt{ch3}[cl] := empty; \texttt{heg} := false;$
9: $\texttt{h\_c\_co} = req\_shared \wedge \neg\texttt{heg} \wedge \texttt{ch2\_4}[\texttt{h\_c\_cl}] = empty$ $\to \texttt{h\_sh\_l}[h\_c\_cl] := true; \texttt{h\_c\_co} := empty; \texttt{ch2\_4}[\texttt{h\_c\_cl}] := grant\_shared;$
10: $\texttt{h\_c\_co} = req\_exclusive \wedge \wedge_i(\texttt{h\_sh\_l}[i] = false) \wedge \texttt{ch2\_4}[\texttt{ho\_c\_cl}] = empty$ $\to \texttt{h\_sh\_l}[\texttt{ho\_c\_cl}] := true; \texttt{h\_c\_co} := empty; \texttt{heg} := true;$ $\texttt{ch2\_4}[\texttt{h\_c\_cl}] := grant\_exclusive;$

Fig. 3. Transitions for $Home$ (Directory)

Clients cannot communicate amongst themselves. The transitions for $Home$ and $client$ processes are given in the guarded command format in figures 3 and 4, respectively. We abbeviate `home_current_client`, `home_current_command`, `home_sharer_list`, `home_invalidate_list` and `home_exclusive_granted` as `h_c_cl`, `h_c_co`, `h_sh_l`, `h_in_l` and `heg`, respectively, and the communication channels `channel1`, `channel2_4` and `channel3` as `ch1`, `ch2_4` and `ch3`, respectively.

1: $(\texttt{cache}[cl] = invalid \wedge \texttt{ch1}[cl] = empty) \to \texttt{ch1}[cl] := req\_shared$
2: $((\texttt{cache}[cl] = invalid \vee \texttt{cache}[cl] = shared) \wedge \texttt{ch1}[cl] = empty)$ $\to \texttt{ch1}[cl] := req\_exclusive$
6: $(\texttt{ch2\_4}[cl] = invalidate \wedge \texttt{ch3}[cl] = empty)$ $\to \texttt{ch2\_4}[cl] := empty; \texttt{ch3}[cl] := invalidate\_ack; \texttt{cache}[cl] := invalid;$
7: $(\texttt{ch2\_4}[cl] = grant\_shared) \to \texttt{cache}[cl] := shared; \texttt{ch2\_4}[cl] := empty;$
8: $\texttt{ch2\_4}[cl] = grant\_exclusive \quad \to \texttt{cache}[cl] := exclusive; \texttt{ch2\_4}[cl] := empty;$

Fig. 4. Transitions for $Client$ (Cache)

We now show how to reduce verification of DIR to that of the *ESI* snoopy protocol, defined below.

The ESI Snoopy Cache Protocol. The template for the ESI protocol is defined as $U = (\{\mathsf{I}, \mathsf{S}, \mathsf{E}\}, \{PrRd, PrWr\}, R, \mathsf{I})$, where the transition relation R consists of the broadcast send transition $\mathsf{I} \xrightarrow{PrRd!!} \mathsf{S}$ with the matching receives $\mathsf{E} \xrightarrow{PrRd??} \mathsf{I}$, $\mathsf{S} \xrightarrow{PrRd??} \mathsf{S}$ and $\mathsf{I} \xrightarrow{PrRd??} \mathsf{I}$; and the I-flush broadcast $\mathsf{I} \xrightarrow{PrWr!!} \mathsf{M}$. The symbols E, S and I denote, respectively, the *exclusive*, *shared* and *invalid* states.

Establishing the Stuttering Path Correspondence. Let U^n_{DIR} represent a system with n clients running the directory based protocol DIR. We begin by showing how the variables used in the DIR protocol impose a relative ordering on the execution of the transitions of the protocol. For transitions (numbered) j, k, we say that j *pre-empts* k, denoted by jPk, to denote the fact that along any global computation of U^n_{DIR}, between any two firings of k (possibly by different clients), there must be at least one firing of j. We write $(j + k)Pm$ to mean that either jPm or kPm, and $j_0P...Pj_k$ to mean that for all $l \in [1 : k]$, $j_{l-1}Pj_l$. For transition j and index $i \in [1 : n]$, we write j_i to indicate that the execution of transition j modifies the local variables of U_i, the process representing the ith client, or the communication variables, `channel1`$[i]$, `channel2_4`$[i]$ and `channel3`$[i]$, shared pairwise between U_i and $Home$.

We first show that $(9+10)P3$. Note that variable `home_current_command` must be set to *empty* for transition 3 to be enabled and that can be done only by firing transitions 9 or 10. Thus one of 9 or 10 has to be fired for 3 to be fired (except for the first time). Also, every time 3 is fired it sets `home_current_command` to a non-*empty* value thus disabling itself and so again one of 9 or 10 has to be fired for 3 to fire again. Similarly, we may show that $3P(9+10)$ (via `home_current_command`) and for $i \in [1 : n]$, 4_iP6_i (via `channel2_4`), 6_iP5_i (via `channel3`) and $3P4_i$ (via `home_invalidate_list`).

Let x be a global computation of U^n_{DIR}. Since $3P(9 + 10)P3$, therefore we have the crucial observation that along x, the firing of 3 alternates with the firing of either 9 or 10. Note that firing transitions 9 or 10 sets the value of `home_current_command` to *empty* thus disabling transition 4. Thus along x, the firing of transition 4 is always sandwiched between the firings of 3 and one of 9 or 10. Consider a firing of 4_j along x. Then the value of `home_current_command` during the last firing of 3 along x is either $req\_exclusive$ or req_shared. If the value is $req\_exclusive$, then since 4_j has been fired therefore after firing the last 3, `home_invalidate_list`$[j] = true =$ `home_sharer_list`$[j]$, and thus transition 5 (the only transition to change the value of `home_sharer_list`$[j]$ to *false*) has to be fired for 10 to be enabled to fire again. Also, since $4_jP6_jP5_j$, we have that the firing of transitions 4_j, 6_j and 5_j is sandwiched between the firing of 3 and 10. If the value of `home_current_command` is req_shared, then we can similarly show that the firing of transition 4_j, 6_j and 5_j is again sanwiched between the firings of 3 and 9. Note that the first scenario, viz., the firing of 3; followed by the firing of the triplet 4_j, 6_j and 5_j for appropriate indices $j \in [1 : n]$; followed by $Home$ firing 10 corresponds to the firing of the snoopy broadcast of *ESI* labeled with $PrWr$ in a distributed fashion. Analogously, the second scenario, viz., the firing of 3; followed by the firing of 4_j, 6_j and 5_j for appropriate j; followed by $Home$ firing 9 corresponds to the firing of the snoopy broadcast of *ESI* labeled with $PrRd$.

The distributed versions of the snoopy broadcasts labeled with $PrRd$ and $PrWr$ are denoted by $d\text{-}PrRd$ and $d\text{-}PrWr$, respectively. Thus the firing of all except the first and last steps, viz., 1, 2, 7 and 8, of each distributed broadcast are sandwiched between the firing of transitions 3 and (9+10). We call these transitions, including transitions 3 and (9+10), the *body* of the distributed transition. The crucial observation is that the bodies of different distributed transitions do not overlap as once 3 is executed by a process, one of 9 or 10 has to be executed by the same process for 3 to be executed again possibly, by a different process, to begin executing the body of another transition. Thus given computation x of U^n_{DIR}, we can arrange all the distributed broadcast transitions fired along x in a sequence $d\text{-}tr_0$, $d\text{-}tr_1$,... based on the order in which their bodies were executed. We say that a distributed transition $d\text{-}tr$ is fired by process U_k of U^n_{DIR} iff the entry transition of $d\text{-}tr$ sets the value of `home_current_client` to k. Let transition $d\text{-}tr_j$ be fired by process U_{i_j} of U^n_{DIR}. Let y be the computation sequence of U^n_{ESI} that results by firing the snoopy broadcasts $tr_0, tr_1, ...$ in the order listed with transition tr_j being fired by process U_{i_j} of U^n_{ESI}. Conversely, given a computation path y of U^n_{ESI}, we can construct a computation path x of U^n_{DIR} by replacing the firing of each snoopy broadcast tr_j by process U_{i_j} of U^n_{ESI} by the firing of all steps of $d\text{-}tr_j$ successively back to back by process U_{i_j} of U^n_{DIR}. This establishes the desired path correspondence.

For the DIR protocol, we are required to verify that in any global state u of U^n_{DIR}, $(u[1] \neq u[2] \wedge u[1] = exclusive) \Rightarrow u[2] = invalid$. Towards that end, it suffices to check the following: $\forall n : U^n_{\mathsf{DIR}} \models \neg\mathsf{EF}(a_1 \wedge b_2)$, where $(a, b) \in \{(exclusive, exclusive), (exclusive, shared)\}$, viz., none of the pairs $(exclusive, exclusive)$, $(exclusive, shared)$ of the DIR protocol is pairwise reachable. The next result reduces reasoning about pairwise reachability for the DIR to the *ESI* protocol.

Proposition 5.1 (Reduction for Safety) For $a, b \neq invalid$, $U^n_{\mathsf{DIR}} \models \mathsf{EF}(a_1 \wedge b_2)$ iff $U^n_{\mathsf{ESI}} \models \mathsf{EF}(a_1 \wedge b_2)$.

Thus it suffices to check that none of the pairs (E, E), (E, S) is pairwise reachable for the ESI protocol. This took 0.01 secs using the abstract history graph technique, and 0.02 secs using the cutoff technique.

The above technique of establishing stuttering path correspondences also works, in general, for LTL$\backslash$X formulae. In [4], it was shown that the property $\mathsf{A}(\mathsf{G}($`channel1`$[1]$ = `request_shared` $\Rightarrow \mathsf{F}($`channel2_4`$[1]$ = `grant_shared`$)))$, viz, once a block is requested in the *shared* state by a cache then it is eventually granted shared access, fails. However, if we assume unconditional fairness, viz., every process fires infinitely often, then the property holds. We now modify the ESI protocol by introducing the intermediate local states rS and rE, standing for $request\_shared$ and$request_exclusive$, respectively. Before executing a broadcast send to the *exclusive* (*shared*) state, we first transit via an *internal* transition to rE (rS) and then fire the broadcast send labeled with $PrWr!!$ ($PrRd!!$) to transit to the *exclusive* (*shared*) state. Then the above liveness property can be reduced to the PMCP for $\mathsf{A}(\mathsf{G}(\mathsf{rS} \Rightarrow \mathsf{F}(\mathsf{S})))$ for the modified ESI. This property has a cutoff of 7 and was verified to hold under assumption of unconditional fairness in 0.02 secs [2]. Note that the property fails if we do not assume fairness. In that case an error trace is automatically generated for the 7 process instance. No manual effort as in [4] is required to validate the erroneous path in the abstraction, an advantage of using cutoffs.

6 Applications and Experimental Results

We consider PMCP for all the snoop based cache protocols presented in [19] (MSI, MESI, Illinois-MESI, MOESI, Berkeley, Synapse N+1, Dragon, Firefly) and the split-transaction version of the MESI protocol. Using the abstract history graph, each of the above protocols was verified in at most 0.01 secs. Although in the worst case the

[2] Technically, we verify the LTL$\backslash$X expressible assertion *fair* $\Rightarrow \mathsf{G}(\mathsf{rS} \Rightarrow \mathsf{F}(\mathsf{S}))$.

number of reachable abstract states in the modified abstract history graph for template $U = (S, R, \Sigma, \mathsf{i})$ 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 was 6, against a worst case possibility of $4 \times 2^4 = 64$ states. In conclusion, the abstract history graph construction seems to work well in practice. In fact, it seems to work even better than the polynomial time cutoff method which too is very efficient requiring only a fraction of a second to verify each invalidation based protocol. This, however, may be due to the fact that whereas the abstract history graph was built directly from the description of the protocol using a separately written code, for the cutoff method we used SMV, possibly resulting in extra overheads from compilation of the protocol specifications, building BDDs etc. The experiments were carried out on a machine with a 797MHz Intel Pentium III processor and 256 Mb RAM.

Protocol	Abstract History Graph		Cutoff Method	
	# of Abstract States	user time (secs.)	Total # of BDD Nodes	user time (secs)
MSI	5	≤ 0.01	7913	0.02
MESI	6	≤ 0.01	8287	0.02
Illinois	6	≤ 0.01	7711	0.02
MOESI	7	≤ 0.01	10284	0.04
N+1	5	≤ 0.01	7913	0.02
Berkeley	5	≤ 0.01	7689	0.03
Firefly	6	≤ 0.01	NA	NA
Dragon	8	≤ 0.01	NA	NA
Split MESI	82	≤ 0.01	NA	NA

7 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 (e.g, [2, 3, 5, 20]). 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. [3]

Similar limitations apply to prior work on PMCP for cache protocols. Some concrete examples of verification of cache protocols can be found in [6, 22]. Pong and Dubois [24] described general methods that were sound but not complete, as they were based on conservative, inexact abstractions. In [16], it was shown that the PMCP for safety over broadcast protocols [14] is decidable using the general backward reachability procedure of [1]. In [21], Maidl, using a proof tree based construction, shows decidability of the PMCP for a broad class of systems including broadcast protocols, but the decision procedure is not known to be primitive recursive. Moreover [14, 16,

[3] However for frameworks that handle specialized domains, sound and complete, fully automatic and, in some cases, efficient decision procedures can be given ([9, 10, 13, 15, 23]).

21] do not report experimental results for cache protocols. In [8], Delzanno uses arithmetical constraints to model global states of systems with many identical caches. His method uses invariant checking via backward reachability analysis of [1] and provides a broad framework for reasoning about cache coherence protocols but his procedure does not terminate on some examples. More recently, a decision procedure based on a modification of the backward reachability algorithm that guarantees termination for all snoopy cache protocols has been given in [12]. However, the backward reachability algorithm of [1] that [8, 12, 16], make use of, although general, suffers from the handicap that the best known bound for its running time is not known to be primitive recursive. Furthermore, this technique does not provide a way to generate *error traces* when a bug is detected. An elegant cutoff method that can verify the DIR protocol was given in [23], but it was sound and not complete and worked only for safety properties. Also in [4], a broad technique was proposed for the verification of WS1S systems that can handle the DIR protocol as an example, but again the resulting technique was sound but not complete.

In this paper, we made three distinct contributions to the parameterized model checking of cache coherence protocols.

First, to reason about general snoopy broadcast protocols, we introduced the framework of *Guarded Broadcast Protocols*. It is both a generalization and a significant simplification of ordered broadcast protocols [11] which required identification of a pre-order on the set of local states of the protocol. The extra transient states found in *split-transaction* bus protocols prevent the imposition of the necessary pre-order. Our new guarded protocol framework eliminates the need to impose a pre-order on protocol states and thereby caters readily for split transactions. This framework is broadly applicable, handling safety properties, and catering for all 8 snoopy protocols in Handy [19], even in their split transaction formulations.

Second, we presented the framework of *Initialized Broadcast Protocols*, establishing provably efficient reasoning about safety and liveness of invalidation based snoopy protocols. We showed that a system with an arbitrary number of caches could be reduced to a system with at most 7 caches. This yields a fully automatic and provably efficient polynomial time algorithm for verifying parameterized invalidation based snoopy cache protocols. Cutoffs have the added important advantage that the small system with 7 caches is a precise replica of large system with n caches, up to size. This not only makes the reduction simple but also caters automatically for error recovery as there is an error in a large system iff there is one in the system with the cutoff number of processes.

Third and last, we described a method for reducing parameterized reasoning about directory based protocols to reasoning about snoopy protocols. We have illustrated the method using the DIR directory based protocol as an example. We then leverage the above cutoff and abstract history graph techniques developed for snoopy protocols to reason about linear time properties of parameterized directory based protocols, which typically are much harder to reason about, in an exact fashion.

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. Baukus, Y. Lakhnech, K. Stahl. Parameterized Verification of a Cache Coherence Protocols: Safety and Liveness, VMCAI 2002, LNCS 2294, pages 317-330.
5. 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.
6. E.M. Clarke, O. Grumberg, H. Hirashi, S. Jha, D. E. Long, K. L. McMillan and L. A. Ness. Verification of the Futurebus+cache coherence protocol. In *Proc. 11th Int. Symp. on Computer Hardware Description Languages and their Applications*, 1993.
7. D. E. Culler and J. P. Singh. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers, 1998.
8. G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. CAV 2000, 51-68.
9. E.A. Emerson and V. Kahlon. Reducing Model Checking of the Many to the Few. CADE 2000.
10. E.A. Emerson and V. Kahlon. Model Checking Large-Scale and Parameterized Resource Allocation Systems. TACAS 2002.
11. E.A. Emerson and V. Kahlon. Rapid Parameterized Model Checking of Snoopy Cache Protocols. TACAS 2003.
12. E.A. Emerson and V. Kahlon. Model Checking Guarded Protocols. LICS 2003.
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. S.M. German. Private communication.
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. M. Maidl. A Unifying Model Checking Approach for Safety Properties of Parameterized Systems. CAV 2001.
22. K. McMillan and J. Schwalbe. Formal Verification of the Gigamax Cache Consistency Protocol. In *Proc. Int. Symp. on Shared Memory Multiprocessors*, pp 242-251, 1991.
23. A. Pnueli, S. Ruah and L. Zuck. Automatic Deductive Verification with Invisible Invariants. TACAS 2001, LNCS, 2001.
24. 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.