Verifying Sequential Consistency on Shared-memory Multiprocessor Systems*

Thomas A. Henzinger** Shaz Qadeer Sriram K. Rajamani

EECS Department, University of California, Berkeley, CA 94720, USA
{tah,shaz,sriramr}@eecs.berkeley.edu

Abstract. In shared-memory multiprocessors *sequential consistency* offers a natural tradeoff between the flexibility afforded to the implementor and the complexity of the programmer's view of the memory. Sequential consistency requires that some interleaving of the local temporal orders of read/write events at different processors be a trace of serial memory. We develop a systematic methodology for proving sequential consistency for memory systems with three parameters —number of processors, number of memory locations, and number of data values. From the definition of sequential consistency it suffices to construct a non-interfering observer that watches and reorders read/write events so that a trace of serial memory is obtained. While in general such an observer must be unbounded even for fixed values of the parameters —checking sequential consistency is undecidable!— we show that for two paradigmatic protocol classes —*lazy caching* and *snoopy cache coherence*— there exist finite-state observers. In these cases, sequential consistency for fixed parameter values can thus be checked by language inclusion between finite automata.

In order to reduce the arbitrary-parameter problem to the fixed-parameter problem, we develop a novel framework for induction over the number of processors. Classical induction schemas, which are based on process invariants that are inductive with respect to an implementation preorder that preserves the temporal sequence of events, are inadequate for our purposes, because proving sequential consistency requires the reordering of events. Hence we introduce *merge invariants*, which permit certain reorderings of read/write events. We show that under certain reasonable assumptions about the memory system, it is possible to conclude sequential consistency for any number of processors, memory locations, and data values by model checking two finite-state lemmas about process and merge invariants: they involve two processors each accessing a maximum of three locations, where each location stores at most two data values. For both lazy caching and snoopy cache coherence we are able to discharge the two lemmas using the model checker MOCHA.

1 Introduction

Shared-memory multiprocessors are an important class of supercomputing systems. In recent years a number of such systems have been designed in both academia and industry. The design of a correct and efficient shared memory is one of the most difficult tasks in the design of such systems. The shared-memory interface is a contract between the designer and the programmer of the multiprocessor. In general, there is a

* This research was supported in part by the DARPA/NASA grant NAG2-1214, by the SRC contract 97-DC-324.041, by the NSF CAREER award CCR-9501708, by the ARO MURI grant DAAH-04-96-1-0341, and by an SRC fellowship of the third author.

** Also at the Max-Planck Institute for Computer Science, Saarbrücken, Germany.

tradeoff between the ease of programming and the flexibility of shared-memory semantics necessary for an efficient implementation. Not surprisingly, a number of abstract shared-memory models have been developed.

All abstract memory models can be understood in terms of the fundamental *serial-memory* model. A serial memory behaves as if there is a centralized memory that services read and write requests atomically such that a read to a location returns the latest value written to that location. *Coherence*[1] requires that the global temporal order of events (reads and writes) at different processors be a trace of serial memory. *Sequential consistency* [Lam79] ignores the global temporal order and requires only that some interleaving of the local temporal orders of events at different processors be a trace of serial memory. Although sequential consistency is a strictly weaker property than coherence, the absence of a synchronizing global clock between the different processors in a multiprocessor makes a sequentially consistent memory indistinguishable from a serial memory. Compared to coherence, sequential consistency clearly offers more flexibility for an efficient implementation; yet, most real systems that claim to be sequentially consistent actually end up implementing coherence. In an effort to get more flexibility for implementation, memory models that relax local temporal order of events at each processor have been developed in recent years. This has been achieved at the cost of complicating the programmer's interface. These memory models such as weak ordering, partial store ordering, total store ordering, and release consistency [AG96] relax the processor order of events in different ways and provide fence or synchronization operations across which sequentially consistent behavior is guaranteed.

We focus on the verification of sequential consistency for two reasons. First, the interface provided by sequential consistency is clear, easy to understand, and widely believed to be the correct tradeoff between implementation flexibility and complexity of the programmer's view of shared memory. In fact, there is a trend of thought [Hil98] that considers the performance gains achieved by relaxed semantics not worth the added complexity of the programmer's interface and advocates sequential consistency as the shared-memory interface for future multiprocessors. Second, even relaxed memory models have fence operations across which sequentially consistent behavior should be observed. Hence, the techniques developed in this paper will be useful for their verification also.

High-level descriptions of shared-memory systems are typically parameterized by the number n of processors, the number m of memory locations, and the number v of data values that can be written in a memory location. A parameterized memory systems consists of a central-control part C and a processor part P. Both C and P are functions that take values for m and v and return a finite-state process. An instantiation of the system containing n processors, m memory locations, and v data values is constructed by composing $C(m, v)$ with n copies of $P(m, v)$. We would like to verify sequential consistency for all values of the parameters. However, sequential consistency is not a *local* property; correctness for m processors (locations, values) cannot be deduced by reasoning about individual processors (locations, values). The following observations about real shared-memory systems, which we assume in our modeling, are crucial for

[1] Implementors of cache-based shared-memory systems have used the notion of cache coherence for a long time but the definition of coherence as stated here was first given in [ABM93].

our results. We assume that the memory system is *monotonic* and *symmetric* with respect to both the set of locations, and the set of data values. Monotonicity in locations means that every run of the system projected onto a subset of locations is a run of the system with just that subset of locations. Monotonicity in data values means that a sequence is a run of the system with some set of possible data values if and only if it is a run of the system with a larger set of data values. Symmetry in locations means that, if σ is a run of the memory system, and λ_l is a permutation on the set of locations, then $\lambda_l(\sigma)$ is also a run of the memory system. Finally, symmetry in data values means that, if σ is a run of the memory system, and λ_v is any function from data values to data values, then $\lambda_v(\sigma)$ is also a run of the memory system.

Even for fixed values of the parameters, checking if a memory system is sequentially consistent is undecidable [AMP96]. The main reason for the problem being undecidable is that the specification of sequential consistency allows a processor to read the value at a location after an unbounded number of succeeding writes to that location by other processors. In real systems, finite resources such as buffers and queues bound the number of writes that can be pending. It is sufficient to construct a witness that observes the reads and writes occurring in the system (without interfering with it) and reorders them while preserving the order of events in each processor such that a trace of serial memory is obtained. We call such a witness an *observer*. If a finite-state observer exists, then it can be composed with a fixed-parameter instantiation of the memory system and the problem of deciding sequential consistency is reduced to a language-containment check between two finite-state automata which can be discharged by model checking. In the concrete examples we have looked at (see below), we have indeed seen that a finite-state observer exists for fixed values of the parameters.

However, our goal is to verify sequential consistency for arbitrary values of the parameters. Towards this end, we first develop a novel inductive proof framework for proving sequential consistency for any number n of processors, given fixed m and v. Inductive proofs on parameterized systems [KM89] use an implementation preorder and show the existence of a *process invariant* such that the composition of the invariant with an additional process is smaller than the process invariant in the preorder. The preorders typically used —for instance, trace containment and simulation— preserve the temporal sequence of events. Since we check a sufficient condition for sequential consistency by the mechanism of an observer that reorders the read/write events of the processors in the system, preorders that preserve the temporal sequence of events do not suffice for our purpose. Our inductive proof strategy first determines a process invariant I_1 of the memory system with respect to the trace-containment preorder to get a finite-state abstraction that can generate all sequences of observable actions for any number of processes. We then find a *merge invariant* I_2 such that (1) the single-process memory system containing I_2 is sequentially consistent, and (2) there is an observer that maps every run σ of $I_2 \| P$ that can be produced in an environment of I_1 to a run σ' of I_2, such that the read/write events in σ' are an interleaving of the read/write events of I_2 and P in σ, and the traces obtained from σ and σ' are identical. Given a run γ of the memory system with $n > 1$ processors, we use the observer to create a run γ' of the memory system with $n - 1$ processors, such that γ and γ' are identical when projected to the events of the first $n - 2$ processors, and the read/write events of the $(n - 1)$-st processor in γ' are an interleaving of the read/write events of the $(n-1)$-st and n-th processors in γ. By doing this n times, we generate a run of the memory system with a single processor, which is sequentially consistent by the base case of the induction.

The induction demonstrates sequential consistency for any number of processors, but given m and v. We would like sufficient conditions under which using fixed values for m and v lets us conclude sequential consistency for all m and v. To that end, we impose three requirements on the process and merge invariants. The first two requirements —symmetry and monotonicity on memory locations— are identical to the corresponding assumptions on the memory system. The third requirement is called *location independence*. A process is location independent if it has the property that a sequence of events is a run of the process with m locations if the m sequences obtained by projecting onto individual memory locations are runs of the process with a single location. We show that if the two invariants satisfy location symmetry, location monotonicity, and location independence, and the observer is location and data independent, then it suffices to do the induction for three memory locations and two data values. As a result, the correctness of the memory system can be proved by discharging two finite-state lemmas using a model checker —one that proves the correctness of the process invariant, and another that proves the correctness of the merge invariant.

Our proof framework can be applied to a variety of protocols; in particular, all cache-coherence protocols described in [AB86] fall into its domain. We demonstrate the method by verifying two example protocols —lazy caching [ABM93] and a snoopy cache-coherence protocol [HP96]. The correctness of lazy caching has been established before by manual proofs [ABM93,Gra94,LLOR99]. The correctness of the snoopy cache-coherence protocol is argued informally in [HP96]. Finite-state observers exist for both these examples. In both cases, the proof of a parameterized system was reduced to finite-state lemmas in the way described above, and discharged by our model checker MOCHA [AHM$^+$98]. Manual effort was required to construct the process and merge invariants, and the observer, and to verify that the assumptions on the memory system and the requirements on the invariants and observer are indeed satisfied.

Related work. We use process induction for the verification of an abstract memory model. We list related work along two axes —work that verifies abstract memory models, and work that verifies systems with an arbitrary number of processes. [MS91,CGH$^+$93,EM95] verify finite instantiations of parameterized memory systems using automatic techniques. [PD95] automatically proves correctness for an arbitrary number of processors but is limited to coherence. [LD92,LLOR99,PD96,PSCH98] verify abstract shared-memory models for all values of parameters but the proofs are not automatic. [GMG91,Gra94,NGMG98] offer sufficient conditions for the satisfaction of sequential consistency that can then be checked on the memory system. [KM89] gives an inductive proof framework for proving the correctness of parameterized systems. [BCG89,WL89,ID96,GS97,EN98] verify parameterized systems but they are not concerned with the specific problem of verifying memory models.

2 Parameterized Memory Systems

2.1 I/O-processes

We use I/O-processes that synchronize on observable actions to model memory systems. Formally, an *I/O-process* A is a 5-tuple $\langle Priv(A), Obs(A), S(A), S_I(A), T(A)\rangle$ with the following components:

- A set $Priv(A)$ of *private actions* and a set $Obs(A)$ of *observable actions*, such that $Priv(A) \cap Obs(A) = \emptyset$. The set $Act(A)$ is the union of $Priv(A)$ and $Obs(A)$. Private actions are outputs, whereas observable actions can be both inputs and outputs. The set of *extended actions* $\Pi(A)$ is given by $Priv(A) \times \{out\} \cup Obs(A) \times \{in, out\}$.
- A finite set $S(A)$ of *states*.
- A set $S_I(A) \subseteq S(A)$ of *initial states*.
- A *transition relation* $T(A) \subseteq S(A) \times \Pi(A) \times S(A)$ satisfying the property that for all $s \in S(A)$ and $\pi \in Obs(A) \times \{in\}$, there is a state $s' \in S(A)$ such that $\langle s, \pi, s'\rangle \in T(A)$.

For all $\pi \in \Pi(A)$, the first component is denoted by $First(\pi)$ and the second component by $Second(\pi)$. A sequence of extended actions $\pi_1, \pi_2, \ldots, \pi_k$ of A is a *run* if there exist states $s_0, s_1, s_2, \ldots, s_k$ such that $s_0 \in S_I(A)$ and $\langle s_i, \pi_i, s_{i+1}\rangle \in T(A)$ for all $0 \leq i < k$. The projection operators $First$ and $Second$ are extended to runs in the natural way. The set of all runs of the I/O-process A is denoted by $\Sigma(A)$. A run is *closed* if $Second(\pi_i) = out$ for all actions π_i in the run. For any set $\beta \subseteq Act(A)$, the *restriction* of the run σ to β is the subsequence obtained by considering the elements from $\beta \times \{in, out\}$ in σ, and is denoted by $[\sigma]_\beta$. For any run σ of I/O-process A, the restriction of σ to $Obs(A)$ is called a *trace*. We say that $tr(\sigma)$ is the trace obtained from the run σ. The set of all traces of the I/O-process A is denoted by $\Gamma(A)$.

Let A_1 and A_2 be two I/O-processes. We say that A_1 *refines* A_2, denoted by $A_1 \preceq A_2$, if (1) $Obs(A_1) \subseteq Obs(A_2)$, and (2) every trace of A_1 is a trace of A_2. The I/O-processes A_1 and A_2 are *compatible* if (1) $Priv(A_1) \cap Act(A_2) = \emptyset$ and (2) $Priv(A_2) \cap Act(A_1) = \emptyset$. The *composition* $A = A_1 \| A_2$ of two compatible I/O-processes A_1 and A_2 is the I/O-process A such that

- $Priv(A) = Priv(A_1) \cup Priv(A_2)$, and $Obs(A) = Obs(A_1) \cup Obs(A_2)$.
- $S(A) = S(A_1) \times S(A_2)$, and $S_I(A) = S_I(A_1) \times S_I(A_2)$.
- $(\langle s_1, s_2\rangle, \pi, \langle t_1, t_2\rangle) \in T(A)$ iff one of the following three conditions holds:
  1. $\pi = \langle a, in\rangle$ and for $k = 1, 2$, if $a \in Act(A_k)$ then $(s_k, \langle a, in\rangle, t_k) \in T(A_k)$ otherwise $s_k = t_k$.
  2. $\pi = \langle a, out\rangle$ and $(s_1, \langle a, out\rangle, t_1) \in T(A_1)$, and if $a \in Act(A_2)$ then $(s_2, \langle a, in\rangle, t_2) \in T(A_2)$ otherwise $s_2 = t_2$.
  3. $\pi = \langle a, out\rangle$ and $(s_2, \langle a, out\rangle, t_2) \in T(A_2)$, and if $a \in Act(A_1)$ then $(s_1, \langle a, in\rangle, t_1) \in T(A_1)$ otherwise $s_1 = t_1$.

Suppose that A_1 and A_2 are compatible I/O-processes. A run $\sigma = \pi_1, \pi_2, \ldots, \pi_k$ of A_1 *can be closed* by A_2 if there is closed run σ' of $A_1 \| A_2$ such that $First(\sigma) = First([\sigma']_{Act(A_1)})$.

2.2 Parameterized memory systems

A parameterized memory system M has three parameters —the number n of processors, the number m of memory locations, and the number v of data values. The parameterized memory system M is built from two parameterized I/O-processes C and P which have two parameters —the number m of memory locations, and the number v of data values. Intuitively, the I/O-process P represents a single processor in the system and C represents a central controller. The I/O-process $M(n, m, v)$ is built from the I/O-processes $C(m, v)$ and $P(m, v)$ by composing $C(m, v)$ and n copies of $P(m, v)$. Given $n > 0$, $m > 0$, and $v > 0$, the memory system $M(n, m, v)$ is an I/O-process that has processors numbered from $1 \dots n$, memory locations numbered from $0 \dots m-1$, and data values numbered from $0 \dots v-1$.

We now formally define a parameterized memory system. Let $\mathbb{N}$ be the set of all non-negative integers. For any $k > 0$, let $\mathbb{N}_k$ denote the set of all non-negative integers less than k. A *parameterized memory system* is a pair $\langle C, P\rangle$ such that both C and P are functions that map $\mathbb{N}\setminus\{0\}\times\mathbb{N}\setminus\{0\}$ to I/O-processes such that for all $m > 0$ and $v > 0$, we have that $Priv(C(m, v)) = PrivNames_C \times \mathbb{N}_m \times (\mathbb{N}_v \cup \{\perp\})$, $Obs(C(m, v)) = ObsNames_C \times \mathbb{N}_m \times (\mathbb{N}_v \cup \{\perp\})$, $Priv(P(m, v)) = PrivNames_P \times \mathbb{N}_m \times (\mathbb{N}_v \cup \{\perp\})$, and $Obs(P(m, v)) = ObsNames_P \times \mathbb{N}_m \times (\mathbb{N}_v \cup \{\perp\})$, where $PrivNames_C$, $ObsNames_C$, $PrivNames_P$, and $ObsNames_P$ are finite sets that satisfy the following properties:

1. $PrivNames_C \cap ObsNames_C = \emptyset$, and $PrivNames_P \cap ObsNames_P = \emptyset$.
2. $PrivNames_C \cap (ObsNames_P \cup PrivNames_P \times \mathbb{N}) = \emptyset$, and $ObsNames_C \cap (PrivNames_P \times \mathbb{N}) = \emptyset$.
3. $R \in PrivNames_P$ and $W \in PrivNames_P$.

The functions $name$, loc, and val are defined on $Act(C(m, v)) \cup Act(P(m, v))$, and extract respectively the first, second, and third components of the actions. Given some m and v, let $RdWr(m, v)$ be the union of the set of *read* actions $\{\langle R, j, k\rangle | j < m \; and \; k < v\}$ and the set of *write* actions $\{\langle W, j, k\rangle | j < m \; and \; k < v\}$.

For all m and v and for all $k > 0$, let $P_k(m, v)$ denote the I/O-process that is obtained from $P(m, v)$ by renaming every private action a to action a', such that (1) $name(a')$ is the pair $\langle name(a), k\rangle$, (2) $loc(a') = loc(a)$, and (3) $val(a') = val(a)$. A parameterized memory system defines a function that maps $\mathbb{N} \times \mathbb{N} \setminus \{0\} \times \mathbb{N} \setminus \{0\}$ to I/O-processes as follows:

$$\begin{aligned} M(0, m, v) &= C(m, v) \\ M(n+1, m, v) &= M(n, m, v) \| P_{n+1}(m, v) \end{aligned}$$

For particular n, m, v, we say that $M(n, m, v)$ is a *memory system*. Note that $M(n, m, v)$ is compatible with $P_{n+1}(m, v)$, due to the renaming of private actions in $P_{n+1}(m, v)$, and the conditions on the names of private and observable actions of C and P described above. The observable actions of $M(n, m, v)$ are the same for all $n > 0$. We define a function $proc$ on the set of actions $\bigcup_{k,m,v} Priv(P_k(m, v))$ such that if a is a private action of $P_k(m, v)$, then $proc(a) = k$.

2.3 Sequential consistency

Let $Memop(n, m, v)$ be the union of the sets $\{\langle\langle R, i\rangle, j, k\rangle | 0 < i \leq n \ and \ j < m \ and \ k < v\}$ and $\{\langle\langle W, i\rangle, j, k\rangle | 0 < i \leq n \ and \ j < m \ and \ k < v\}$. Thus $Memop(n, m, v)$ denotes the set of read and write operations of $M(n, m, v)$. The functions $name$, loc, and val, which were originally defined on actions of $P(m, v)$ and $C(m, v)$, can be defined analogously on actions of $M(n, m, v)$. Thus, the four functions $name$, loc, val, and $proc$ are defined on all members of $Memop(n, m, v)$. We use $Memop$ to denote the set $\bigcup_{n,m,v} Memop(n, m, v)$.

Let $\sigma = \pi_1, \pi_2, \ldots, \pi_k$ be a sequence in $Memop^*$, the set of finite sequences with elements from $Memop$. The *abstraction* of σ, denoted by $\Lambda(\sigma)$, is a labeled directed graph $\langle V, E, L\rangle$, where V is a finite set of vertices, $E \subseteq V \times V$, and L is a function from V to $Memop(n, m, v)$, such that (1) $V = \{1, 2, ..., k\}$, (2) for all $i \in V$, we have that $L(i) = \pi_i$, and (3) for all $x, y \in V$, we have that $\langle x, y\rangle \in E$ iff $proc(L(x)) = proc(L(y))$ and $x < y$. We observe that for every sequence $\sigma \in Memop^*$, the abstraction $\Lambda(\sigma)$ is an acyclic graph. Thus, we can obtain total orderings of the vertices in $\Lambda(\sigma)$ that respect the dependencies specified by its edges. Since the edges form a partial order, several such total orders, which are called linearizations of σ, may exist. Formally, a one-to-one mapping $f : V \to V$ is a *total order* of $\Lambda(\sigma) = \langle V, E, L\rangle$ if for all $x, y \in V$, whenever $\langle x, y\rangle \in E$ we have that $f^{-1}(x) < f^{-1}(y)$. If f is a total order of $\Lambda(\sigma)$, then the sequence $L(f(1)), L(f(2)), \ldots, L(f(|V|))$ of actions in $Memop(n, m, v)$ is a *linearization* of σ.

We are interested in defining which sequences from $Memop^*$ are serial. Intuitively, a sequence from $Memop^*$ is serial if it can be produced by serial memory where each read from a location returns the value written by the last write to that location. We state this formally below. Let $\sigma = \pi_1, \pi_2, \ldots, \pi_k$ be a sequence in $Memop^*$. We define $lastwrite_\sigma$ as a function that associates with each position i in σ, the position j in σ where the most recent write to the location $loc(\pi_i)$ was done. Formally, $lastwrite_\sigma$ is a mapping from the from the set $\{1, 2, ..., k\}$ to $\{1, 2, ..., k\} \cup \{\bot\}$ such that $lastwrite_\sigma(i) = j$ if there exists a j such that $j \leq i$, $loc(\pi_i) = loc(\pi_j)$, $name(\pi_j) = \langle W, n_1\rangle$ for some n_1, and there does not exist any j' with $j < j' \leq i$, $name(\pi_{j'}) = \langle W, n_2\rangle$ for some n_2, and $loc(\pi_{j'}) = loc(\pi_i)$; otherwise $lastwrite_\sigma(i) = \bot$. The sequence σ is *serial* if for all $i \leq k$, if $lastwrite_\sigma(i) \neq \bot$, then $val(\pi_i) = val(\pi_{lastwrite_\sigma(i)})$.

For the following definitions, we extend the abstraction function Λ to operate on arbitrary sequences σ by first restricting it to actions in $Memop$. Formally, for any σ, we have that $\Lambda(\sigma) = \Lambda([\sigma]_{Memop})$. We extend Λ to operate on sequences of extended actions by operating it on the first component of each extended action. Formally, if σ is a sequence of extended actions, then $\Lambda(\sigma) = \Lambda(First(\sigma))$. Let $\Sigma_M = \bigcup_{n,m,v} \Sigma(M(n, m, v))$. Then Λ is defined for all sequences in Σ_M.

Definition 1 (Observer). *Let M be a parameterized memory system. A function Ω from Σ_M to $Memop^*$ is an* observer *for the memory system $M(n, m, v)$ if for every run $\sigma \in \Sigma(M(n, m, v))$, the sequence $\Omega(\sigma)$ is a linearization of σ. The observer Ω is a* serializer *for $M(n, m, v)$ if for every run $\sigma \in \Sigma(M(n, m, v))$, the sequence $\Omega(\sigma)$ is serial.*

Definition 2 (Sequential consistency [Lam79]). *Let M be a parameterized memory system. The memory system $M(n, m, v)$ is* sequentially consistent *if it has a serializer. The parameterized memory system M is sequentially consistent if $M(n, m, v)$ is sequentially consistent for all $n > 0$, $m > 0$, and $v > 0$.*

2.4 Assumptions on parameterized memory systems

In order to reduce the proof of sequential consistency of the parameterized memory system to finite state model checking obligations, we make some assumptions about memory systems. We first state a few additional definitions. Let σ be a run of the memory system $M(n, m, v)$. We denote by $\sigma|_j$ the run σ restricted to the jth memory location. Formally, we have $\sigma|_j = [\sigma]_\beta$, where $\beta = \{a \mid a \in Act(M(n, m, v)) \; and \; loc(a) = j\}$. For $j > 0$, we denote by $\sigma_{<j}$ the run σ restricted to memory locations numbered less than j; that is, $\sigma|_{<j} = [\sigma]_\beta$, where $\beta = \{a \mid a \in Act(M(n, m, v)) \; and \; loc(a) < j\}$.

Assumption 1 (Location symmetry) *Let $\lambda : \mathbb{N}_m \rightarrow \mathbb{N}_m$ be a permutation function on the set of memory locations. Extend λ to actions, extended actions and extended action sequences in the natural way. Then,*

1. *for all $\sigma \in \Sigma(C(m, v))$, we have that $\lambda(\sigma) \in \Sigma(C(m, v))$, and*
2. *for all $\sigma \in \Sigma(P(m, v))$, we have that $\lambda(\sigma) \in \Sigma(P(m, v))$.*

Assumption 2 (Location monotonicity)

1. *If $\sigma \in \Sigma(C(m, v))$, then for all $j \leq m$, we have $\sigma|_{<j} \in \Sigma(C(j, v))$.*
2. *If $\sigma \in \Sigma(P(m, v))$, then for all $j \leq m$, we have $\sigma|_{<j} \in \Sigma(P(j, v))$.*

Assumption 3 (Data symmetry) *Let $\lambda : \mathbb{N}_v \cup \{\bot\} \rightarrow \mathbb{N}_v \cup \{\bot\}$ be any function on the set of data values, such that $\lambda(x) = \bot$ iff $x = \bot$. Extend λ to actions, extended actions and extended action sequences in the natural way. Then,*

1. *for all $\sigma \in \Sigma(C(m, v))$, we have that $\lambda(\sigma) \in \Sigma(C(m, v))$, and*
2. *for all $\sigma \in \Sigma(P(m, v))$, we have that $\lambda(\sigma) \in \Sigma(P(m, v))$.*

Assumption 4 (Data monotonicity) *For all m, n, v_1, v_2, if $v_1 \leq v_2$, then*

1. *for all $\sigma \in Act(C(m, v_1))^*$, we have $\sigma \in \Sigma(C(m, v_1))$ iff $\sigma \in \Sigma(C(m, v_2))$, and*
2. *for all $\sigma \in Act(P(m, v_1))^*$, we have $\sigma \in \Sigma(P(m, v_1))$ iff $\sigma \in \Sigma(P(m, v_2))$.*

Note that the function λ in assumption 1 above is a permutation on the set of locations, whereas the function λ in assumption 3 could be any arbitrary function on the set of data values. Let λ_0 be the function from $Act(M(n, m, v))$ to $Act(M(m, n, v))$, which changes the location attribute to 0. Formally, $\lambda_0(a) = a'$ such that $name(a') = name(a)$, $loc(a') = 0$, and $val(a') = val(a)$. We extend λ_0 to extended action sequences in the natural way. The observer Ω is *location independent* if for all j, we have that $\Omega(\lambda_0(\sigma|_j)) = \lambda_0(\Omega(\sigma)|_j)$. The observer Ω is *data independent* if for every function $\lambda : \mathbb{N}_v \cup \{\bot\} \rightarrow \mathbb{N}_v \cup \{\bot\}$ such that $\lambda(x) = \bot$ iff $x = \bot$, we have that $\Omega(\lambda(\sigma)) = \lambda(\Omega(\sigma))$.

Proposition 1. *Suppose the parameterized memory system M satisfies assumptions 1–4. For all $n > 0$, the following two statements are equivalent:*

1. *There is a location and data independent serializer for $M(n, n, 2)$.*
2. *There is a location and data independent serializer for $M(n, m, v)$ for all $m > 0$ and $v > 0$.*

Suppose we fix the number of processors to n. Due to the above proposition it suffices to consider only n locations and 2 data values, if the serializer we design is location and data independent. Since our objective is to prove sequential consistency for an arbitrary number of processors, we give a method based on induction over the number of processors for this. The inductive step in the method considers two processors and designs a serializer-like function for them. Then an argument similar to the one used in proving Proposition 1 will let us show that it is enough to perform the inductive step for fixed numbers of memory locations and data values.

3 Reducing Sequential Consistency to Finite-state Proof Obligations

3.1 Induction on the set of processors

We show how to check sequential consistency of $M(n, m, v)$ for all $n > 0$ by induction over the number of processors. We do not need any of the assumptions 1–4 for the results in this section.

We note that every trace of $\Gamma(M(n, m, v))$ can be obtained by a run in $M(n + 1, m, v)$ in which the $(n + 1)$-st processor does not perform any output action. Hence $\Gamma(M(n, m, v))$ is contained in $\Gamma(M(n + 1, m, v))$ for all n. We would like to analyze a processor in an environment consisting of an arbitrary number of processors. Hence, we would like an upper bound on the trace set $\Gamma(M(n, m, v))$ for all n. A sufficient condition for this upper bound is captured by process invariants [KM89]. A function I_1 with two arguments m and v is a *possible process invariant* for the parameterized memory system M if for all m and v, we have that $I_1(m, v)$ is a I/O-process such that that (1) $Obs(I_1(m, v)) = Obs(M(n, m, v))$ for all $n > 0$ (recall that the set of observable actions of $M(n, m, v)$ is the same for all $n > 0$), and (2) $I_1(m, v)$ is compatible with $P(m, v)$.

Definition 3 (Process invariant). *Let I_1 be a possible process invariant for the parameterized memory system M. The function I_1 is a* process invariant *of M if the following condition is true for all m and v:*

$$[A_{I_1}(m, v)] \quad \begin{array}{ll} \textit{1.} & C(m, v) \preceq I_1(m, v) \\ \textit{2.} & I_1(m, v) \| P(m, v) \preceq I_1(m, v) \end{array}$$

Proposition 2. *Suppose I_1 is a process invariant of the parameterized memory system M. Then, for all $n > 0$, $m > 0$, and $v > 0$, we have that $M(n, m, v) \preceq I_1(m, v)$.*

If the parameterized memory system M is sequentially consistent, then by our definition, there exists an observer Ω for M such that for every sequence σ of memory operations of $M(n, m, v)$, the function Ω produces a rearranged sequence σ' such that (1) σ' is serial, and (2) σ and σ' agree on the ordering of the memory operations of each individual processor. We wish to provide an inductive construction that produces such an observer for arbitrary n. The construction uses the notion of a generalized processor called a merge invariant, and a witness function that works like an observer for a two-processor system consisting of the merge invariant and $P(m, v)$.

Recall that $RdWr(m, v)$ is the set of private actions of $P(m, v)$ that represent read and write operations. For technical reasons, we want the memory operations of the merge invariant to be named differently than those of $P(m, v)$. Let $Rd'(m, v) = \{\langle R', j, k\rangle | j < m \; and \; k < v\}$, and let $Wr'(m, v) = \{\langle W', j, k\rangle | j < m \; and \; k < v\}$. Let $RdWr'(m, v)$ denote the union of $Rd'(m, v)$ and $Wr'(m, v)$. We define the function $prime$ on $RdWr(m, v)$ by $prime(\langle R, j, k\rangle) = \langle R', j, k\rangle$ and $prime(\langle W, j, k\rangle) = \langle W', j, k\rangle$. We define the function $unprime$ on $RdWr'(m, v)$ by $unprime(\langle R', j, k\rangle) = \langle R, j, k\rangle$ and $unprime(\langle W', j, k\rangle) = \langle W, j, k\rangle$. We extend $prime$ and $unprime$ to sequences of actions in the natural way. We say that the sequence $\sigma' \in RdWr'(m, v)^*$ *rearranges* the sequence $\sigma \in (RdWr(m, v) \cup RdWr'(m, v))^*$ if σ' is an interleaving of $prime([\sigma]_{RdWr(m,v)})$ and $[\sigma]_{RdWr'(m,v)}$.

A function I_2 with two integer arguments m and v is a *possible merge invariant* for the parameterized memory system M if for all m and v, we have that $I_2(m, v)$ is an I/O-process such that (1) $Obs(I_2(m, v)) = Obs(P(m, v))$, (2) $RdWr'(m, v) \subseteq Priv(I_2(m, v))$ and $RdWr(m, v) \cap Priv(I_2(m, v)) = \emptyset$, and (3) $I_2(m, v)$ is compatible with both $P(m, v)$ and $C(m, v)$. Let I_1 be a process invariant of M and I_2 be a possible merge invariant of M. A function Θ from $\bigcup_{m,v} \Sigma(I_2(m, v)||P(m, v))$ to $\bigcup_{m,v} \Sigma(I_2(m, v))$ is a *merging function* if $\sigma \in \Sigma(I_2(m, v)||P(m, v))$ implies $\Theta(\sigma) \in \Sigma(I_2(m, v))$.

Definition 4 (Merge invariant). *Let I_1 be a process invariant and let I_2 be a possible merge invariant for the parameterized memory system M. The function I_2 is a* merge invariant *of M with respect to I_1 if there exists a merging function Θ such that the following two conditions are true for all m and v:*

[$B1_{I_2}(m, v)$] *For every closed run σ of $I_2(m, v)||C(m, v)$, the sequence $unprime([\sigma]_{RdWr'(m,v)})$ is serial.*

[$B2_{I_2,I_1,\Theta}(m, v)$] *For every run σ of $I_2(m, v)\|P(m, v)$ that can be closed by $I_1(m, v)$, we have that $\Theta(\sigma)$ rearranges σ, and $[\sigma]_{Obs(I_2(m,v))} = [\Theta(\sigma)]_{Obs(I_2(m,v))}$.*

Note that the I/O-process $I_2(m, v)||C(m, v)$ is a single-processor memory system. We say that the merging function Θ is a *witness* for $B2_{I_2,I_1,\Theta}(m, v)$ if Θ makes condition $B2_{I_2,I_1,\Theta}(m, v)$ true.

Let I_1 be a process invariant of M. Suppose I_2 is a possible merge invariant of M, and Θ is a merging function such that $B1_{I_2}(m, v)$ and $B2_{I_2,I_1,\Theta}(m, v)$ are true for some m and v. For some $n > 0$, consider the process $I_2(m, v)\|M(n, m, v)$, which can be written as $I_2(m, v)\|P_n(m, v)\|M(n - 1, m, v)$. Consider any closed run σ of $I_2(m, v)\|M(n, m, v)$. Clearly there is a run σ' of $I_2(m, v)\|P_n(m, v)$ that is closed by a run of $M(n - 1, m, v)$ to produce σ. Since I_1 is a process invariant of M, we have that σ' is closed by a run of $I_1(m, v)$. Therefore, using Θ we can rearrange σ' to obtain a run $\Theta(\sigma')$ of $I_2(m, v)$ which is closed by a run of $M(n - 1, m, v)$. Thus we have managed to rearrange a closed run of $I_2(m,v)\|M(n,m,v)$ into a closed run of $I_2(m,v)\|M(n-1,m,v)$. By repeating this procedure we eventually obtain a run of $I_2(m,v)\|C(m,v)$, which is sequentially consistent by condition $B1_{I_2}(m,v)$. Since every run of $M(n,m,v)$ is also run of $I_2(m,v)||M(n,m,v)$, it follows that n applications of Θ effectively produce an observer Ω which is a serializer for $M(n,m,v)$. The existence of such an observer implies the sequential consistency of $M(n,m,v)$.

Theorem 1. *Let M be a parameterized memory system. If I_1 is a process invariant of M and I_2 a merge invariant of M with respect to I_1, then M is sequentially consistent.*

Suppose that we manage to come up with possible invariants I_1 and I_2, and a merging function Θ. How do we verify for all m and v that $A_{I_1}(m,v)$, $B1_{I_2}(m,v)$, and $B2_{I_2,I_1,\Theta}(m,v)$ hold? In the following two sections, we describe sufficient conditions whereby proving these obligations for fixed values of m and v will let us conclude that they hold for all m and v.

3.2 Reduction to a fixed number of memory locations

In this section, we use assumptions 1 and 2 on the parameterized memory system. Further, we impose requirements on the process and merge invariants and the merging function that will reduce the verification problem to one on a fixed number of memory locations. The first two requirements are identical to assumptions 1 and 2 on the parameterized memory system.

Requirement 1 (Location symmetry) *Let $\lambda : \mathbb{N}_m \to \mathbb{N}_m$ be a permutation function on the set of memory locations. Extend λ to actions, extended actions and extended action sequences in the natural way. We require for the possible process invariant I_1 and the possible merge invariant I_2 that*

1. *for all $\sigma \in \Sigma(I_1(m,v))$, we have that $\lambda(\sigma) \in \Sigma(I_1(m,v))$, and*
2. *for all $\sigma \in \Sigma(I_2(m,v))$, we have that $\lambda(\sigma) \in \Sigma(I_2(m,v))$.*

Requirement 2 (Location monotonicity) *We require for the possible process invariant I_1 and the possible merge invariant I_2 that*

1. *if $\sigma \in \Sigma(I_1(m,v))$ then for all $j \leq m$, we have $\sigma|_{<j} \in \Sigma(I_1(j,v))$, and*
2. *if $\sigma \in \Sigma(I_2(m,v))$ then for all $j \leq m$, we have $\sigma|_{<j} \in \Sigma(I_2(j,v))$.*

For any run σ of $I_2(m,v)$, we define $tr'(\sigma)$ as the restriction of σ to $Obs(I_2(m,v)) \cup RdWr'(m,v)$. Let $\Gamma'(I_2(m,v))$ be the set $\{tr'(\sigma) | \sigma \in \Sigma(I_2(m,v))\}$. Recall that λ_0 is a function that changes the location attribute of an action to 0.

Requirement 3 (Location independence) *We require for the possible process invariant I_1 and the possible merge invariant I_2 that*

1. *$\sigma \in \Gamma(I_1(m,v))$ if $\sigma \in Act(I_1(m,v))^*$, and for all $0 \leq j < m$, we have $\lambda_0(\sigma|_j) \in \Gamma(I_1(1,v))$, and*
2. *$\sigma \in \Gamma'(I_2(m,v))$ if $\sigma \in Act(I_2(m,v))^*$, and for all $0 \leq j < m$, we have $\lambda_0(\sigma|_j) \in \Gamma'(I_2(1,v))$.*

Consider a merging function Θ. We say that Θ is *location independent* if whenever $\sigma \in \Sigma(I_2(m,v) \| P(m,v))$, then $\Theta(\lambda_0(\sigma|_j)) = \lambda_0(\Theta(\sigma)|_j)$ for all $j < m$.

Theorem 2. *Let M be a parameterized memory system satisfying assumptions 1 and 2. Let I_1 be a possible process invariant and let I_2 be a possible merge invariant for M satisfying requirements 1–3. Then the following conditions hold for all $v > 0$:*

1. *$A_{I_1}(1,v)$ is true iff $A_{I_1}(m,v)$ is true for all $m > 0$.*
2. *$B1_{I_2}(1,v)$ is true iff $B1_{I_2}(m,v)$ is true for all $m > 0$.*
3. *There is a location-independent witness Θ satisfying $B2_{I_2,I_1,\Theta}(l,v)$ for $l \leq 3$ iff there is a location-independent witness Θ' satisfying $B2_{I_2,I_1,\Theta'}(m,v)$ for all $m > 0$.*

The condition $l \leq 3$ in the last item of the above theorem comes from the fact that a witness Θ needs to preserve three orderings while rearranging a run of $I_2(m,v) \| P(m,v)$ —(1) the order of memory operations in $I_2(m,v)$, (2) the order of memory operations in $P(m,v)$, and (3) the order of observable actions in $I_2(m,v) \| P(m,v)$. If Θ does not preserve these orderings, and if Θ is location independent, we can prove that there exists a run σ of $I_2(3,v) \| P(3,v)$ such that either $\Theta(\sigma)$ does not rearrange σ, or $[\sigma]_{Obs(I_2(3,v))} \neq [\Theta(\sigma)]_{Obs(I_2(3,v))}$.

3.3 Reduction to a fixed number of data values

In this section, we assume that the memory system satisfies assumptions 3 and 4. Recall the definition of a data-independent observer.

Theorem 3. *Let M be a parameterized memory system satisfying assumptions 3 and 4. For all $n > 0$, $m > 0$, and $v > 0$, if Ω is a data-independent observer for the memory system $M(n,m,v)$, then Ω is a serializer for $M(n,m,2)$ iff Ω is a serializer for $M(n,m,v)$.*

Consider a merging function Θ. We say that Θ is *data independent* if for all v, and for every function $\lambda : \mathbb{N}_v \cup \{\perp\} \rightarrow \mathbb{N}_v \cup \{\perp\}$ such that $\lambda(x) = \perp$ iff $x = \perp$, we have that $\Theta(\lambda(\sigma)) = \lambda(\Theta(\sigma))$. Suppose that the witness for $B2_{I_2,I_1,\Theta}(m,v)$ is data independent. Then the implicit observer function that is produced for $M(n,m,v)$ as a result of n applications of the witness is also data independent.

Corollary 1. *Let M be a parameterized memory system satisfying assumptions 1–4. Let I_1 be a possible process invariant and let I_2 be a possible merge invariant for M satisfying requirements 1–3. Let Θ be a location and data independent merging function. Suppose $A_{I_1}(1,2)$ and $B1_{I_2}(1,2)$ are true, and Θ is a witness for $B2_{I_2,I_1,\Theta}(3,2)$. Then $M(n,m,v)$ is sequentially consistent for all $n > 0$, $m > 0$, and $v > 0$; that is, M is sequentially consistent.*

4 Two Applications: Lazy Caching and Snoopy Coherence

We show how the theory developed in the previous section can be used to verify sequential consistency of memory systems with an arbitrary number of processors, locations and data values using a model checker. We consider two specific memory protocols, namely the lazy caching protocol from [ABM93] and a snoopy cache-coherence protocol from [HP96].

For each of these protocols, we first argue that assumptions 1–4 are satisfied by the memory system, and that requirements 1–3 are satisfied by the process and merge invariants. Then, we design a witness Θ and argue that it is location and data independent. The following observations provide justification for our informal arguments:

– The invariants and the witness have the property that they never base their decisions on data values. Thus, they are data independent by design.
– The memory system inherently enforces a total order on the writes to every location. In fact, every memory system we know of has this property. Our merge witness respects this total order for every location. Let M be a parameterized memory system and let Θ be a merging function. Let σ be a run of M and let j be any location. The order of writes in $\Theta(\sigma)|_j$ is the same as the total order of writes to location j in σ. Every read to a location reads the value written to that location by some earlier write. The witness also respects this causal relationship between the writes and the reads. If two reads of location j access the value written by the same write, then the witness places them in their temporal order of occurrence in $\Theta(\sigma)|_j$. Thus, the ordering of events to a location j is independent of the events to other memory locations and determined solely by the temporal sequence of events to location j. Hence, our witness is naturally location independent.

We finally discharge the three proof obligations of Corollary 1 using our model checker MOCHA.

Lazy Caching. The lazy caching protocol allows a processor to complete a write in its local cache and proceed even while other processors continue to access the “old” value of the data in their local caches. Each cache has an output queue in which writes are buffered and an input queue in which reads are buffered. In order to satisfy sequential consistency, some restrictions are placed on read accesses to a cache when either writes or updates are buffered. A complete description of the protocol can be found in [ABM93].

The I/O-process $C(m, v)$ for this protocol is the trivial process with a single state that accepts all input actions. The I/O-process $P(m, v)$ is a description of one processor and cache in the system. The set $Priv(P(m, v))$ has actions with three different names: *read*, *write*, and *update*. An *update* action occurs when a write gets updated to a local cache from the head of its input queue. There is one action for each combination of these names with locations, processors and data values —a total of $3 \times n \times m \times v$ private actions. The set $Obs(P(m, v))$ has actions with one name: *serialize*. A *serialize* action occurs when a processor takes the result of a local write from the head of its output queue and transmits it on the bus. The *serialize* action does not identify the processor which did the action. Thus, a processor has $m \times v$ different observable actions.

The process invariant I_1 is such that for all m and v, the I/O-process $I_1(m, v)$ simply generates all possible sequences of *serialize* actions. It is trivial to see that I_1 is a process invariant. The merge invariant I_2 is exactly the same as P. The merging function Θ is non-trivial. It queues *write* actions and delays them until the corresponding *update* action is seen by all processors. It also delays *read* actions until the corresponding *write* has been made visible. The witness preserves processor order, never bases decisions on data values, and respects the total order of writes that is inherent to the lazy-caching protocol. By design, the witness is location and data independent.

We used MOCHA [AHM+98] to verify that the merging function Θ is a witness for the merge invariant for three locations and two data values. This obligation had about 60 latches and required MOCHA about 4 hours to check on a 625 MHz DEC Alpha 21164.

Snoopy Cache Coherence. The snoopy coherence protocol has a bus on which all caches send messages, as well as "snoop" and react to messages. Each location has a state machine which is in one of three states: **read-shared**, **write-exclusive**, or **invalid**. If a location is in **read-shared** state, then a cache has permission to read the value. If a location is in **write-exclusive** state, then a cache has permission to both read and write the value. In order to transition to **read-shared** or **write-exclusive** states, the cache sends messages over the bus, and other caches respond to these messages. There is also a central memory attached to the bus. When a location is not held in **write-exclusive** by any cache, the memory owns that location and responds to read requests for that location.

The I/O-process $C(m, v)$ for this protocol models the central memory, and $P(m, v)$ models one processor with a local cache. The process $C(m, v)$ has no private actions. It has observable actions with four different names: *read-request*, *write-request*, *read-response*, and *write-response*. The process $P(m, v)$ has private actions with two different names: *read* and *write*, and the same set of observable actions as $C(m, v)$. None of the observable actions identify the processor that did the action.

The process invariant is such that for all m and v, we have that $I_1(m, v)$ is a generalization of the processor $P(m, v)$. The processor is generalized so that it can send a *read-request* for a location even if it already has the location in **read-shared**, and a *write-request* even if it already has the location in **write-exclusive**. The merge invariant I_2 is identical to I_1 with the additional capability to execute private *read* and *write* actions. The merging function Θ preserves temporal order of occurrence of reads and writes. This simple witness works because the snoopy protocol implements coherence. Again, by design the witness is data and location independent. We used MOCHA to verify that $I_1(1, v)$ is a process invariant and $I_1(3, v)$ is a merge invariant. MOCHA required less than 15 minutes to check these.

References

[AB86] J. Archibald, J.-L. Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Computer Systems*, 4(4):273–298, 1986.

[ABM93] Y. Afek, G. Brown, M. Merritt. Lazy caching. *ACM Trans. Programming Languages and Systems*, 15(1):182–205, 1993.

[AG96] S.V. Adve, K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, S. Tasiran. MOCHA: Modularity in model checking. In *CAV 98: Computer Aided Verification*, LNCS, pp. 521–525. Springer-Verlag, 1998.

[AMP96] R. Alur, K.L. McMillan, D. Peled. Model-checking of correctness conditions for concurrent objects. In *Proc. 11th IEEE Symp. Logic in Computer Science*, pp. 219–228, 1996.

[BCG89] M.C. Browne, E.M. Clarke, O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81(1):13–31, 1989.

[CGH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proc. 11th IFIP WG10.2 Conf. Computer Hardware Description Languages and their Applications*, pp. 15–30, 1993.

[EM95] A.T. Eiriksson, K.L. McMillan. Using formal verification/analysis methods on the critical path in system design: a case study. In *CAV 95: Computer Aided Verification*, LNCS 939, pp. 367–380. Springer-Verlag, 1995.

[EN98] E.A. Emerson, K.S. Namjoshi. Verification of a parameterized bus arbitration protocol. In *CAV 98: Computer Aided Verification*, LNCS 1427, pp. 452–463. Springer-Verlag, 1998.

[GMG91] P.B. Gibbons, M. Merritt, K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proc. 3rd ACM Symp. Parallel Algorithms and Architectures*, pp. 292–303, 1991.

[Gra94] S. Graf. Verification of a distributed cache memory by using abstractions. In *CAV 94: Computer Aided Verification*, LNCS 818, pp. 207–219. Springer-Verlag, 1994.

[GS97] S. Graf, H. Saidi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, LNCS 1254, pp. 72–83. Springer-Verlag, 1997.

[Hil98] M.D. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer*, 31(8):28–34, 1998.

[HP96] J.L. Hennessy, D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann, 1996.

[ID96] C. N. Ip, D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996.

[KM89] R.P. Kurshan, K.L. McMillan. A structural induction theorem for processes. In *Proc. 8th ACM Symp. Principles of Distributed Computing*, pp. 239–247, 1989.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, C-28(9):690–691, 1979.

[LD92] P. Loewenstein, D.L. Dill. Verification of a multiprocessor cache protocol using simulation relations and higher-order logic. *Formal Methods in System Design*, 1(4):355–383, 1992.

[LLOR99] P. Ladkin, L. Lamport, B. Olivier, D. Roegel. Lazy caching in TLA. To appear in *Distributed Computing*.

[MS91] K. L. McMillan, J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proc. Symp. Shared Memory Multiprocessors*, Inf. Process. Soc. Japan, pp. 242–251, 1991.

[NGMG98] R. Nalumasu, R. Ghughal, A. Mokkedem, G. Gopalakrishnan. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In *CAV 98: Computer Aided Verification*, LNCS 1427, pp. 464–476. Springer-Verlag, 1998.

[PD95] F. Pong, M. Dubois. A new approach for the verification of cache coherence protocols. *IEEE Trans. Parallel and Distributed Systems*, 6(8):773–787, 1995.

[PD96] S. Park, D.L. Dill. Protocol verification by aggregation of distributed transactions. In *CAV 96: Computer Aided Verification*, LNCS 1102, pp. 300–310. Springer-Verlag, 1996.

[PSCH98] M. Plakal, D.J. Sorin, A.E. Condon, M.D. Hill. Lamport clocks: verifying a directory cache-coherence protocol. In *Proc. 10th ACM Symp. Parallel Algorithms and Architectures*, pp. 67–76, 1998.

[WL89] P. Wolper, V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *CAV 89: Computer Aided Verification*, LNCS 407, pp. 68–80. Springer-Verlag, 1989.