Automatable Verification of Sequential Consistency*

Anne E. Condon and Alan J. Hu
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver, B.C. V6T 1Z4
Canada
(condon,ajh)@cs.ubc.ca

November 6, 2001

Abstract

Sequential consistency is a multiprocessor memory model of both practical and theoretical importance. Designing and implementing a memory system that efficiently provides a given memory model is a challenging and error-prone task, so automated verification support would be invaluable. Unfortunately, the general problem of deciding whether a finite-state protocol implements sequential consistency is undecidable. In this paper, we identify a restricted class of protocols for which verifying sequential consistency is decidable. The class includes all published sequentially consistent protocols that are known to us, and we argue why the class is likely to include all real sequentially consistent protocols. In principle, our method can be applied in a completely automated fashion for verification of all implemented protocols.

1 Introduction

Shared memory multiprocessing has become the commercially dominant form of multiprocessing in current systems. In such a system, all processors can load information from or store information to any memory location in a global address space shared by all processors.

Correctly programming a shared memory multiprocessor requires an understanding of when memory stores performed by one processor become visible to loads performed

*The authors were supported in part by research grants from the National Science and Engineering Research Council of Canada. This paper is the full presentation of work that initially appeared in the 2001 Symposium on Parallel Algorithms and Architectures [6].

Real Time	Processor 1	Processor 2
Initially,	memory locations *x* and *y* are both 0.	
Time 1	`ST r1, x`	
2	`ST r2, y`	
3		`LD r2, y`
4		`LD r1, x`

What values are loaded into registers `r1` and `r2`?

Figure 1: Different memory models allow different results. With a serial memory, the only possible outcome is $\texttt{r1} = 1$, $\texttt{r2} = 2$. With sequential consistency, the per-processor program order must be respected, but the relationship between processors is unspecified, so $\texttt{r1} = 0$, $\texttt{r2} = 0$ is also legal, as is $\texttt{r1} = 1$, $\texttt{r2} = 0$, but not $\texttt{r1} = 0$, $\texttt{r2} = 2$. More relaxed models permit ignoring program order in certain circumstances, allowing the two loads to execute out-of-order, resulting in $\texttt{r1} = 0$ and $\texttt{r2} = 2$.

by other processors. A *memory model* provides such an understanding, by giving a formal specification of how the memory system will behave from the programmer's perspective. Perhaps the most intuitive memory model, called "serial memory", requires the memory system to behave as if each memory operation occurred instantaneously and atomically in the exact order in which the processors execute load and store instructions in real time, in the sense that the value of every load operation is that of the most recent store operation. Unfortunately, the hardware and performance overhead required to implement serial memory is prohibitive for large or high-performance multiprocessor systems. Real systems use many techniques to enhance performance, such as store buffers, caches, and out-of-order interconnect networks. Such techniques lead to memory models where the order in which memory operations appear to have been performed may differ from the real-time order in which they executed, might not agree with the order in which operations were executed on a single processor (the "program order"), and might not even be the same order on different processors. (See Figure 1.)

Sequential consistency is a memory model introduced by Lamport [10]. A memory system is sequentially consistent iff there always exists an interleaving of the program orders of all the processors such that each load returns the value of the most recent store to the same address. Sequential consistency is important both as a practical memory model that provides intuitive ease-of-programming while allowing efficient hardware optimizations (e.g. [9]) and also as an extensively studied memory model that can be used to understand other, more relaxed models (e.g. [1]).

Memory systems use intricate finite-state protocols to implement the desired memory model. These protocols are notoriously difficult to design and debug — because the primary objective is performance rather than simplicity — making them natural targets for formal verification.

Ideally, we would like an algorithm that inspects a finite-state protocol and determines automatically whether or not the protocol provides sequential consistency. Unfortunately, the general problem of deciding sequential consistency of a finite-state protocol is undecidable [3].

Real protocols, however, might not be fully general, suggesting that the undecidability result may not be relevant in practice. Suppose we can characterize a class of protocols with the following properties: membership in the class is decidable, all members of the class are sequentially consistent, and all real protocols that implement sequential consistency belong to the class. In that case, automatic verification of real, sequentially consistent protocols reduces to testing for membership in the class. This paper proposes such a protocol class.

The basis for our verification method is a graph-based definition of sequential consistency that arises in the work of Gibbons and Korach [7]. For an execution trace of a protocol, they define a constraint graph with a node for each load and store operation in the trace. The graph has four kinds of edges: edges that enforce program order for each processor, edges that provide a total order over all store nodes to each memory location, edges from each store node to every load node that gets its value from that store, and forced edges from each load node to the store node that follows in the total store order the store node from which the load got its value. A protocol is sequentially consistent if and only if all of its traces have acyclic constraint graphs.

To perform automatic formal verification using this formulation of sequential consistency, we must provide an automatic way to construct the constraint graph and verify that it is acyclic for all possible executions of the protocol. In practice, this suggests that the construction and checking of the constraint graph must be done in (hopefully small) finite state, so that automatic verification based on finite-state model checking [5] is possible.

The remainder of this paper addresses these problems. In Section 3, we introduce a graph description notation tailored to describe constraint graphs, and a finite-state checker to verify that a graph so described is acyclic. We also describe how the graph description notation and checker can be used to verify sequential consistency. In Section 4, we show how real protocols can be annotated with finite-state information, to obtain a finite state observer which generates a description of the constraint graph. Our method of generating this description characterizes a class of protocols for which sequential consistency is decidable, and we argue why all real protocols are likely to belong to this class. Finally, we derive size bounds on the finite-state observer, suggesting that our method is at the edge of what is currently feasible for automatic verification tools.

1.1 Related Work

There has been considerable work over the years on verifying memory system protocols and memory models. For brevity, we mention here only closely related work, pertaining to finite-state verification of protocols with respect to sequential consistency.

Plakal et al. [12] introduce a verification approach based on logical clocks and apply it to a directory based protocol. Our approach is inspired by the logical clocks approach, but in contrast to logical clocks, which are unbounded, our approach reduces verification to a language inclusion problem between finite state automata.

Henzinger et al. [8] propose a very similar approach to ours, using a finite-state observer to reorder loads and stores to construct a witness of sequential consistency. Because of the finite-state limit on reordering, the method is too restrictive to handle

most real protocols. One could view our approach as a generalization of theirs that handles all realistic protocols. We note that Henzinger et al. prove very strong results for protocols in their restrictive class, namely that it is sufficient to reduce verification of a protocol with arbitrarily large parameters (number of processors, number of blocks, number of values per block) to a fixed-parameter problem. In contrast, our method applies to verification of only fixed-parameter protocols.

Nalumasu et al. [11] propose the Test Model-Checking technique, in which a protocol is checked against various predefined finite-state automata that test certain memory model properties. These tests can be considered to be finite-state observers. By combining these tests, it is possible to verify memory models that are close to, but not identical to, sequential consistency. Determining exactly how these test combinations relate to sequential consistency and to the class of protocols we can handle is an open question.

Recently, Qadeer has proposed an approach for automatically verifying that a memory protocol implements a memory model [13]. His approach and ours are superficially similar — both involving automated constructions of finite-state witnesses that a protocol obeys a memory model — but the constructions are quite different. For example, Qadeer's method specifically assumes that the protocol and witness will be model-checked and exploits this assumption to simplify the witness to look for only particular types of error traces. In contrast, our construction flags any violation in any run of the protocol; model checking gives complete verification, but the method is easily adapted to a testing scenario. In their current versions, our approach handles a more general class of protocols than Qadeer's, which does not handle Afek et al's Lazy Caching protocol [2], for example. On the other hand, Qadeer's complexity bounds (on the size of the finite state witness) are better than ours, and he considers memory models other than just sequential consistency. Which method (or what combination of the two methods) will be most useful in practice remains to be determined.

On the experimental side, we have implemented a technique related to that presented in this paper and experimented with a substantial, realistic memory system protocol [4]. The general approach is the same as in this paper, but the underlying model for recording and checking constraints is different, resulting in wildly impractical complexity bounds for automatically generating the finite-state witness. Nevertheless, we were able to demonstrate that the method does allow verification, using current model-checking tools, of the sequential consistency of a substantial cache protocol, provided that some human insight is used to generate an efficient witness. In contrast, the present paper presents a revised theoretical framework that encompasses a broader class of protocols, yet allows proving much stronger complexity bounds, suggesting that this work will apply to more protocols and be fully automatable in practice.

2 Definitions

2.1 Protocols

We define a protocol as basically a finite-state machine, but with some specializations to simplify our notation. A **protocol** $\mathcal{P}$ is a tuple $(p, b, v, Q, q_0, \mathcal{A} \cup \mathcal{A}', \delta \cup \delta', \perp)$. The

constants p, b, and v specify the number of processors, memory blocks, and data values in the protocol. The symbol $\perp$ denotes the initial value of each block. The set of states is Q, of which q_0 is the initial state. The set $\mathcal{A}$ is the set of all actions of the protocol that are LD and ST operations, namely actions of the form $\mathrm{LD}(P,B,V)$ and $\mathrm{ST}(P,B,V)$, where $1 \leq P \leq p$, $1 \leq B \leq b$, and $1 \leq V \leq v$. For notational convenience, we use *'s to denote sets of LD and ST actions over all values of a parameter: e.g., ST(*,B,V) denotes the set $\{\mathrm{ST}(P,B,V) \mid 1 \leq P \leq p\}$. Thus, $\mathcal{A} = \mathrm{ST}(*,*,*) \cup \mathrm{LD}(*,*,*)$. $\mathcal{A}'$ is the set of actions of the protocol other than LD and ST operations. Corresponding to $\mathcal{A}$ and $\mathcal{A}'$ there are two transition relations, δ and δ', with $\delta \subseteq Q \times \mathcal{A} \times Q$ and $\delta' \subseteq Q \times \mathcal{A}' \times Q$.

A sequence of actions $A_1, A_2, \ldots, A_k$ is a **protocol run** if there is a sequence of states $q_0, q_1, q_2, \ldots, q_k$ such that for all j, with $1 \leq j \leq k$, the transition $(q_{j-1}, A_j, q_j) \in \delta \cup \delta'$. A **protocol trace** is the subsequence of a protocol run that includes only the actions in $\mathcal{A}$ (i.e., the ST and LD operations). Two protocols $\mathcal{P}$ and $\mathcal{P}'$ are **equivalent** if the set of traces of $\mathcal{P}$ equals the set of traces of $\mathcal{P}'$. Note that the runs and traces of a protocol are finite, so our theory uses regular automata rather than ω-automata.

2.2 Sequential Consistency

Intuitively, a serial trace is one in which each LD returns the value of the most recent (prior to the LD) ST to the same block. If there were no prior STs to that block, the load must return $\perp$. Formally, a trace $T = t_1, t_2, \ldots, t_k$ is a **serial trace** if for all blocks B and values V, for all $1 \leq j \leq k$:

$$(t_j \in \mathrm{LD}(*,B,V)) \Rightarrow \left(\begin{array}{c} (V = \perp) \wedge \forall i_{<j}[t_i \notin \mathrm{ST}(*,B,*)] \\ \vee \\ \exists h_{<j}[t_h \in \mathrm{ST}(*,B,V) \wedge \forall i_{h<i<j}(t_i \notin \mathrm{ST}(*,B,*)] \end{array} \right).$$

A **reordering** of a trace of length k is simply a permutation Π of the numbers from 1 to k. Let $\Pi = \pi(1), \pi(2), \ldots \pi(k)$ be a reordering of a trace T. Let $T' = t_{\pi(1)}, t_{\pi(2)}, \ldots t_{\pi(k)}$. Π is called a **serial reordering** and T' is the corresponding serial trace if Π and T' have the following two properties. First, Π preserves the "per processor" order of T, i.e., for all processors P, if t_a and t_b are operations of processor P then $a < b$ if and only if $\pi^{-1}(a) < \pi^{-1}(b)$. Second, T' must be a serial trace.

A protocol is **sequentially consistent** if all of its traces have a serial reordering.

3 Verifying Sequential Consistency Using Constraint Graphs

In our method for verifying that a protocol is sequentially consistent, a finite-state observer watches a protocol as it executes and gathers information about how to reorder the trace. The observer presents this information, in the form of a finite-state constraint graph, to a checker. A key task of the checker, which is also finite state, is to ensure that the graph is acyclic. Verification reduces to proving that the checker accepts all

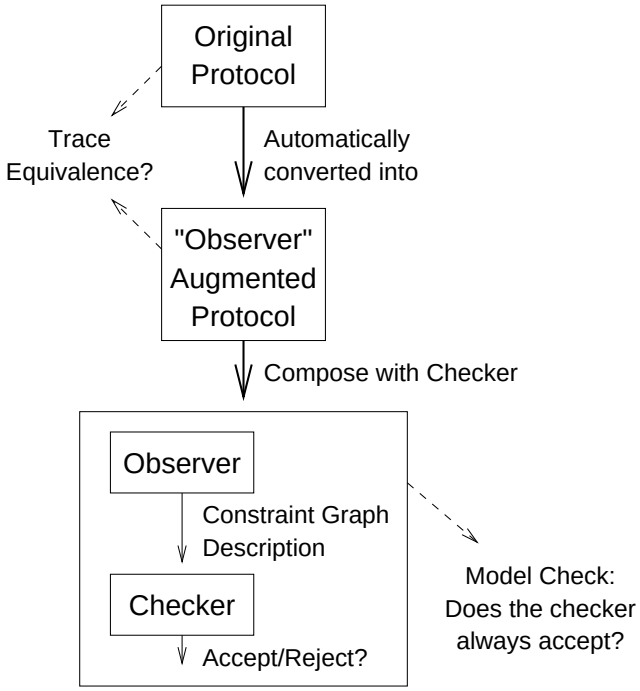

Figure 2: Verification Method Overview. The Observer is simply the original protocol augmented with reordering information. Automatic creation of the observer is discussed in Section 4. The observer generates a description of a constraint graph, which is checked by a finite-state checker. The same checker is used for all protocols. Constraint graphs and the checker are described in Section 3. The trace equivalence check can be omitted in practice because the observer is created in a non-interfering way from the original protocol.

constraint graphs generated by the observer. Figure 2 illustrates the main steps in the verification process. Overall, the method exploits the "less is more" principle: a total reordering of a trace is too much to be collected and checked with a finite number of states, but partial information about the reordering is sufficient to deduce sequential consistency.

We first define sequential consistency using graph-theoretic notation. Application of this definition to protocol verification requires a finite state method for testing if a graph is acyclic. In Section 3.2, we identify a class of graphs for which this test can be done. We describe the finite state cycle-checker in Section 3.3. We combine everything into our verification method in Section 3.4.

3.1 A Graph-Based Definition of Sequential Consistency

A *constraint graph* G for a trace T records ordering constraints on the operations in T which are sufficient to ensure that T has a serial reordering. The nodes of G are labeled by operations of T. Nodes are numbered by consecutive integers, starting from 1, according to their order in the trace. Edges of G include program order edges, along with *inheritance edges*, which indicate from which ST operation a LD inherits its value; *ST order edges*, which provide a total ordering of all ST nodes to the same block, and *forced edges*, which force the constraint that on any path from a ST node to a LD node that inherits its value, there is no other ST node to the same block. More precisely, edges of G must satisfy the following **edge annotation constraints:**

1. Each edge may be annotated as an inheritance, program order, ST order, or forced edge. An edge may have zero or more annotations.

2. For each processor P, if u nodes of G are labeled by operations of P then G has $u-1$ program order edges that define a total order on these u operations, consistent with trace order. There are no other program order edges.

3. For each block B, if u nodes of G are labeled by ST operations to B, then G has $u-1$ ST order edges that define a total order on these u operations. There are no other ST order edges.

4. Each node labeled by LD(P,B,V), $V \neq \perp$, has one incoming inheritance edge from a ST(P',B,V) node (where P may equal P'). There are no other inheritance edges.

5. (a) Let (i,j,k) be a triple of nodes with the property that there is a ST order edge from i to k and an inheritance edge from i to j. Then there is a forced edge on some path from j to k. Specifically, if j is labeled by LD(P,B,V), $V \neq \perp$, then there is either a forced edge directly from j to k or there is a (program order) path from j to another node j', where j' also inherits its value from i, and a forced edge from j' to k.

   (b) Similarly, let j be a node labeled by a LD$(P,B,\perp)$ operation, Then there is a forced edge on a path to the first node in the ST order for block B.

An example of a constraint graph is given in Figure 3. The following claim is implicit in the work of Gibbons and Korach [7] and follows directly from the definition of constraint graph.

Lemma 3.1 *A trace T has a serial reordering if and only if some constraint graph for T is acyclic.*

Proof: Suppose that $T = t_1, t_2, \ldots, t_i$ has a serial reordering Π and let T' be the corresponding serial trace. Let G be the graph obtained from Π as follows. The nodes of the graph G are labeled by operations of T, and are numbered according to their order in T. There is an edge from the node numbered a to node numbered b if and only if

- t_a is an operation of processor P and t_b is the first operation of P to follow t_a in T', in which case the edge is a program order edge, or

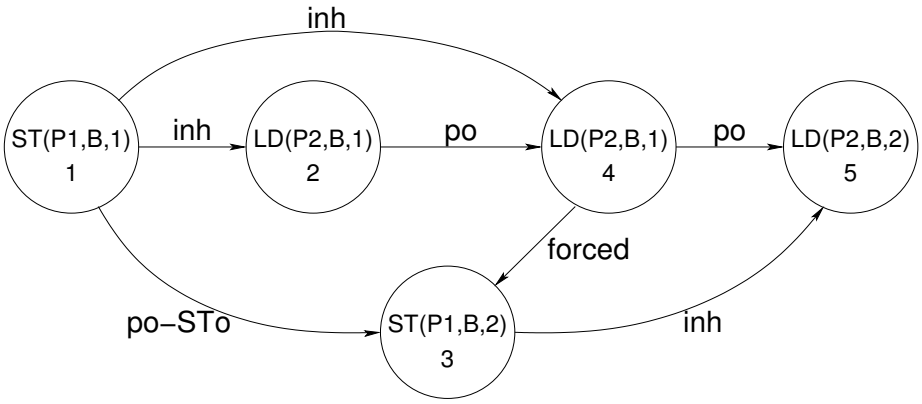

Figure 3: A Constraint Graph. Edge labels indicate inheritance (inh), program order (po), store order (STo), or "forced" edges. The inheritance edge from node 1 to node 4 and the store order edge from node 1 to node 3 forces an edge from node 4 to node 3, which prevents trace orders in which the LD in node 4 does not get its value from the most recent ST.

- t_a is a ST operation to block B and t_b is the first ST operation to block B to follow t_a in T', in which case the edge is a ST order edge, or
- t_b is a LD operation to block B and t_a is the last ST operation in T' to block B before t_b, in which case the edge is an inheritance edge, or
- for some i, the triple (i,a,b) is such that there is a ST order edge from i to b and an inheritance edge from i to a, in which case the edge is a forced edge, or
- a is a node labeled by a LD$(P,B,\perp)$ operation, and t_b is the first ST operation to block B in T' (if any), in which case the edge is a forced edge.

It is straightforward to show that graph G is a constraint graph for T, in which every edge has at least one annotation. In particular, edges of G satisfy edge constraint 2 because T' respects the program order of T and by construction (bullet 1), and satisfy edge constraint trivially 3 by construction (bullet 2). To see that edges of G satisfy edge constraint 4 note that, since T' is a serial trace, there is no ST operation to block B in T' earlier than any LD$(P,B,\perp)$ operation, and so G does not have any inheritance edge into a node labeled LD$(P,B,\perp)$ for any P and B. Also, by construction (bullet 3), all other LD(P,B,V) nodes have one incoming inheritance edge from the most recent ST to block B in T', and this ST node must be in ST$(*,B,V)$, again since T' is serial. Edge constraint 5 is satisfied trivially by construction (bullets 4 and 5). Moreover, G is acyclic because all edges other than forced edges respect the order Π, and forced edges cannot introduce cycles. To see the latter fact, note that if triple (i,j,k) is as in edge constraint 5, with all three nodes labeled by operations to block B, then there is a ST order edge from i to k in G and k must follow i in T', and thus must follow j since i is the last ST to B preceding j.

Conversely, suppose that G is an acyclic constraint graph for trace T. Then in fact any total order of the node numbers of G that respects the edges of G is a serial reordering of T. To see this, let T' be the trace corresponding to a total ordering of the nodes of G. The program order edges of G ensure that T' respects program order. The inheritance, forced, and ST order edges together ensure that the trace is serial: by definition, a node labeled LD(P,B,V) has an inheritance edge from a ST node in ST$(*,P,V)$, and a forced edge ensures that the next ST in ST order must follow the LD node. Finally, note that at least one such total order of the nodes of G must exist since G is acyclic. □

3.2 Node Bandwidth Bounded Graphs

For verification purposes, we are interested in constraint graphs (with ordered nodes) that are *node bandwidth bounded*. We denote the set $\{1,2,\ldots,i\}$ by $\mathbf{N}_i$. We say that a graph with node set $\mathbf{N}_n$ is *k-node bandwidth bounded* if for all i, at most k nodes in $\mathbf{N}_i$ have edges to or from nodes in the set $\mathbf{N}_n - \mathbf{N}_i$. For example, the graph in Figure 3 is 3-node-bandwidth bounded. Note that node bandwidth boundedness is a property of both the graph and a fixed node ordering. Also, note that our definition differs from the usual edge-based notion of graph bandwidth, e.g., the number of edges between nodes in $\mathbf{N}_i$ and $\mathbf{N}_n - \mathbf{N}_i$ may be unbounded. For brevity, we omit the word "node" and simply refer to bandwidth bounded graphs.

We will represent a directed, k-bandwidth bounded graph G as a string, called a k-graph descriptor, in a way that facilitates a finite state test that a graph is acyclic. For later convenience, nodes and edges of G may have labels from some finite alphabets $\mathcal{A}$ and $\mathcal{E}$, respectively. (In our application, $\mathcal{A}$ will be the set of trace operations, and symbols in $\mathcal{E}$ will denote the edge annotations of section 3.1.) Intuitively, our graph description notation simply lists nodes by number and edges as pairs of node numbers, with additional labels (if any) immediately following the node or edge to which they belong. A naive approach numbers all nodes and lists them in order. For example, the graph in Figure 3 corresponds to the description:

> 1, ST($P1,B,1$), 2, LD($P2,B,1$), (1,2), inh, 3, ST($P1,B,2$), (1,3), po-STo, 4, LD($P2,B,1$), (1,4), inh, (2,4), po, (4,3), forced, 5, LD($P2,B,2$), (3,5), inh, (4,5), po

Our approach is like the naive approach, but is finite-state by exploiting k-bandwidth boundedness. In our approach, node numbers are not used directly to identify nodes and edges. Rather, each node may have one or more ID's (identification numbers) between 1 and $k+1$. When all edges in or out of the node with ID i have been listed, i may be used to identify another node. The graph in Figure 3 is 3-bandwidth bounded, so we can describe it as:

> 1, ST($P1,B,1$), 2, LD($P2,B,1$), (1,2), inh, 3, ST($P1,B,2$), (1,3), po-STo, 4, LD($P2,B,1$), (1,4), inh, (2,4), po, (4,3), forced, 1, LD($P2,B,2$), (3,1), inh, (4,1), po

In this example, once all edges into the first four nodes of the graph have been listed, the number 1 is recycled to refer to node 5.

As will become clear from our formal definition below, a node may have more than one ID (with respect to a given k-graph descriptor). This is useful, for example, when modeling the following situation: the value of a ST node in the constraint graph is in multiple cache locations of a finite state protocol, in which case it is convenient that these location addresses are the graph IDs for the ST node.

More formally, with respect to some fixed k and symbol alphabets $\mathcal{A}$ and $\mathcal{E}$, we define a *node descriptor* to be a symbol in $\mathbf{N}_{k+1}$, possibly followed by a symbol in $\mathcal{A}$ (that is, a node ID possibly followed by a node label) and an *edge descriptor* to be a symbol of the form (i, j) where $i, j \in \mathbf{N}_{k+1}$, possibly followed by a symbol in $\mathcal{E}$. A k-**graph descriptor** is simply a sequence of node descriptors and edge descriptors, along with symbols from the set $\{\text{add-ID}(I, I') \mid 1 \leq I, I' \leq k+1\}$. Intuitively, the add-ID(I, I') symbol causes the ID I' to be added to the node with ID I, if any (and I' is no longer associated with any other node).

Testing if a string is a proper graph descriptor (does not have two consecutive symbols from $\mathcal{A}$, for example), is easily done in finite state.

Let s be a k-graph descriptor. The graph G *represented by* s has a number of nodes equal to the number of node descriptors of s, with the ith node having the label (if any) of the ith node descriptor. Associated with each prefix s' of s is a set of *active* nodes, each of which has a non-empty set of ID's. Here, for each node i, we define the ID-set of i with respect to s', denoted by ID-set(i, s'), as follows. If s' has fewer than i node descriptors, then ID-set(i, s') is empty. If s' has exactly i node descriptors, and ends with a node descriptor which has ID I, then ID-set$(i, s') = \{I\}$. Next, suppose that s' has more than i node descriptors.

- If $s' = s'', I$ and $I \in$ ID-set(i, s''), then ID-set(i, s') is defined to be ID-set$(i, s'') - \{I\}$. (ID I is now being used to label another node, and so is no longer in the ID-set of the ith node.)
- If $s' = s'', \text{add-ID}(I, I')$ and $I \in$ ID-set(i, s''), then ID-set(i, s') is defined to be ID-set$(i, s'') \cup \{I'\}$. (Add I' to the ID-set of node i.)
- If $s' = s'', \text{add-ID}(I', I)$ with $I \neq I'$ and $I \in$ ID-set(i, s''), then ID-set(i, s') is defined to be ID-set$(i, s'') - \{I\}$. (Again, ID I is now being used to label another node, and so is no longer in the ID-set of the ith node.)
- Otherwise, ID-set(i, s') = ID-set(i, s''). (No change to the ID-set of the ith node.)

Then, the edges of G are defined as follows: for each prefix of the form $s', (I, I')$ of s, if for some pair (i, j) of nodes of G, $I \in$ ID-set(i, s') and $I' \in$ID-set(j, s') then edge (i, j) is in G. Moreover, if $s', (I, I'), \beta$ is also a prefix of s then the edge (i, j) has label β.

The next lemma shows that any k-bandwidth bounded graph can be represented by a k-graph descriptor in which the size of the ID set for active nodes is exactly 1.

Lemma 3.2 *Any k-bandwidth bounded graph can be represented by a k-graph descriptor.*

Proof: We prove a slightly stronger property, namely that any k-bandwidth bounded graph G can be represented by a k-graph descriptor s in which all of the nodes in $\mathbf{N}_{n-1}$

with edges to node n, plus n itself, are in the active set associated with s, and the size of the ID-set for each active node is exactly 1. The proof proceeds by induction on the number of nodes of G. For simplicity, we ignore edge and node labels, but these can be trivially be added to an (unlabeled) graph descriptor. The base case, when G has one node, is also trivial to prove.

For the induction step, let G have $n > 1$ nodes. Let G' be the graph obtained by removing node n and all of its incident edges from G. Let graph G'' be obtained by adding to G' edge $(i, n-1)$ if and only if edge (i,n) is in G, and edge $(n-1,i)$ if and only if edge (n,i) is in G. Let E'' be the set of edges added to G', in order to obtain G''. We claim that G'' is also k-bandwidth bounded. In fact, for all $i, 1 \leq i \leq n-2$, the set of nodes in $\mathbf{N}_i$ with edges to or from $\mathbf{N}_{n-i}$ in graph G is the same as the set of nodes in $\mathbf{N}_i$ with edges to or from $\mathbf{N}_{n-1-i}$ in graph G''.

By the induction hypothesis, G'' has a k-graph descriptor, s'' in which all of the nodes in $\mathbf{N}_{n-2}$ with edges to node $n-1$, plus $n-1$ itself, are in the active set associated with s''. To obtain a k-graph descriptor for G, first remove all descriptors of edges in E''. Next, append to s'' a node descriptor for node n. The ID for this node descriptor can be found as follows. If there are only k active nodes associated with s', then some ID in the range $\{1,\ldots,k+1\}$ is not in the ID-set of any of these nodes (since each active node has an ID-set of size 1). Otherwise, of the $k+1$ active nodes, one does not have any edge to or from node n. The ID of this node can then be recycled for node n. If s' is the string s'' with the node descriptor for node n appended, then all nodes of G with edges to or from n, plus n itself, are in the active set associated with s'. Finally, append to s' the edge descriptors of each edge to or from node n, to obtain the k-graph descriptor s for graph G. □

3.3 Checking for Cycles in a Bandwidth Bounded Graph

Lemma 3.3 *There is a finite state cycle-checker that, given as input a k-graph descriptor, accepts if and only if the string represents an acyclic graph.*

Proof: While reading node descriptors, edge descriptors, and add-ID symbols from left to right in the input string, the cycle-checker maintains a so-called *active graph* containing at most $k+1$ nodes, in which each node has an associated ID-set. The checker ignores node and edge labels. Upon reading a node ID or edge pair, the cycle-checker does the following:

- Suppose that a node ID, say I, or an add-ID(I',I) symbol is read. If there is a node with ID-set $\{I\}$ in the active graph, then for all pairs of edges $(H,I),(I,J)$ in the active graph (where H,I,J refer to node IDs) a new edge (H,J) is added, if not already in the graph. (The edge (H,J) is referred to as the contraction of (H,I) and (I,J)). Then the node with ID-set $\{I\}$ and all incident edges are removed from the graph. Otherwise, if I is in the ID-set of some node that has an ID-set of size greater than 1, then I is removed from the ID-set of this node. Finally, if the symbol read is a node-ID I, then a new node with ID I is added to the graph and if the symbol read is add-ID(I',I) then I is added to the ID-set of the node whose ID-set contains I' (if any).

- When edge (I, I') is read, an edge is added from node with ID I to the node with ID I'. If addition of this edge introduces a cycle in the graph, the automaton rejects.

If, upon reaching the end of the string, the checker has not rejected, it accepts. Correctness of the checker follows from the fact that the edge contraction plus node removal done in the first test of the checker preserves cycles in the graph. □

3.4 Observer-Checker Verification Method

In our method for protocol verification, the *observer* generates the same set of traces as the protocol, but augments each trace with a description of a k-bandwidth bounded graph. Given a run of the observer, the *checker* checks that the graph is an acyclic constraint graph for the trace.

Let $\mathcal{P}$ be a protocol. Let $\mathcal{A}$ be the set of LD and ST operations of $\mathcal{P}$. An *observer* for $\mathcal{P}$ is itself a finite state protocol. The alphabet (set of actions) of an observer consists of the symbols used in a k-graph descriptor for some k, in which the node label set is $\mathcal{A}$ and the edge label set $\mathcal{E}$ is {**inh, po, forced, STo, po-STo, po-inh, po-forced**}, where **inh**, **po**, **STo** and **forced** indicate inheritance, program order, ST order and forced edges, respectively, and **po-STo, po-inh**, and **po-forced** denote edges with two annotations. Note that each run of an observer contains a trace as a subsequence, namely the subsequence of symbols from $\mathcal{A}$.

Definition 3.1 *An observer O for $\mathcal{P}$ is a* witness for $\mathcal{P}$ *if (i) the set of traces of O equals the set of traces of $\mathcal{P}$, and (ii) each run of O describes an acyclic constraint graph (as defined in section 3.1).*

Theorem 3.1 *Let $\mathcal{P}$, O be protocols. If O is a witness for $\mathcal{P}$, then $\mathcal{P}$ is sequentially consistent. Moreover, testing whether O is a witness for $\mathcal{P}$ can be reduced to the language inclusion problem for finite state automata.*

Proof: If protocol O is a witness for protocol $\mathcal{P}$, then sequential consistency of $\mathcal{P}$ follows directly from property (i) of Definition 3.1 above and from Lemma 3.1.

In testing whether O is a witness for $\mathcal{P}$, the check for property (i) can trivially be reduced to the language equivalence problem for finite state automata. In practice, this check is trivial by construction, since the observer is a noninterfering augmentation of the protocol.

We next describe the *checker*, a finite state automaton that checks property (ii) of Definition 3.1. We assume that k (the bandwidth bound) is fixed. The alphabet of the checker equals that of the observer. Given as input a run r of observer O, the checker does the following:

- Run the cycle-checker of Lemma 3.2 for k-bandwidth bounded graphs on r. If the cycle-checker rejects, then reject. Otherwise, r is an acyclic, k-bandwidth bounded graph.
- In concert, check that edges satisfy the edge annotation properties listed in section 3.1 - we next describe in detail below how this check is done. If the edge annotation properties are satisfied, then accept, else reject.

By the definition of a witness in section 3.1, the checker accepts if and only if r describes an acyclic constraint graph.

The check for part (i) can be done in a finite number of states, using the finite state cycle checker of section 3.3.

We now show that the edge annotation checks needed for part (ii) can also be done with a finite number of states. To perform these checks, the checker associates each node in the active graph with its label (namely LD or ST operation) as well as its ID-set.

First, consider the check that the operations of each processor are totally ordered by program order edges. The checker associates two logical bits with each active node, called *program-edge-in* and *program-edge-out*. These are initially set to false. When an edge (I, I') with program order label is added to the graph, the checker does not annotate the edge with its label. Rather, if *program-edge-out* of the node with ID I is set to true, the checker rejects, since there must be more than one program order edge out of I. Otherwise I's *program-edge-out* bit is set to true. Similarly, if *program-edge-in* of the node with ID I' is set to true, the checker rejects, since there must be more than one program order edge into of I'. Otherwise the *program-edge-in* bit of I' is set to true. Finally, the checker tests that, over all nodes, exactly one node has *program-edge-in* set to false when it is removed from the active graph (this is the first node in program order) and that exactly one node has *program-edge-out* set to false (this is the last node in program order), and that these two nodes be distinct. All of this can be done in finite state. The check that the ST operations to a given block are totally ordered by ST order edges is similar.

Next, we describe the check that each node labeled by LD(P,B,V), $V \neq \perp$, has one incoming inheritance edge from a node with label in ST(*,B,V). The checker maintains a single logical bit, *inheritance-edge-in*, for each node in the active graph with a LD(P,B,V) label where $V \neq \perp$. When an inheritance edge (I, I') is added to the graph, the checker rejects if I' does not have an associated *inheritance-edge-in* bit or if the *inheritance-edge-in* bit of I' is already true. Also, if I''s operation is LD(P,B,V), the checker rejects if the label of node I is not in ST$(*,B,V)$. Otherwise, *inheritance-edge-in* of I' is set to true. Finally, if the *inheritance-edge-in* label of a node is false when the node is removed from the active graph, then the checker rejects.

It remains to describe how to check that the constraints on forced edges are satisfied, assuming that the checks on program order, ST order, and inheritance edges above are satisfied. For each triple (i, j, k) of nodes with the properties that j is labeled by LD(P,B,V), there is a ST order edge from i to k and there is an inheritance edge from i to j, it must be checked that there is either a forced edge directly from j to k or there is a (program order) path from j to another node j', where j' also inherits its value from i, and there is a forced edge from j' to k.

For this check, a variable called *forced-edge-on-path-to* is associated with each LD node j when an inheritance edge is added into j. This variable is uninitialized unless there is already a ST order edge from i to some node k, where i is the node from which j inherits its value. In the latter case, the *forced-edge-on-path-to* variable is initialized to point to node k. Otherwise, this initialization happens when the ST order edge is added from i to k. In addition, inheritance edges and forced edges are labeled as such, and when two edges (H, I) and (I, J) are contracted, where (I, J) is a forced edge and H and I are in the same program order (i.e. are labeled with operations by the same

processor P), the resulting contracted edge (H, J) is labeled as a forced edge.

The test is complicated by the fact that node j may no longer be part of the active graph (i.e. have a non-empty ID-set) before a forced edge is added on some program order path from j to k. To handle this, the checker defers the removal of any LD j from the graph until either (i) the variable *forced-edge-on-path-to* is initialized, say to k, and there is a forced edge from j to k, or (ii) another LD node j' is added to the active graph, where j' follows j in program order and inherits its value from the same node as j does. The graph maintained by the checker consists of active nodes and deferred nodes, and deferred nodes are no longer maintained once one of the conditions (i) or (ii) is met. Thus, the number of nodes whose removal from the graph is deferred is bounded, since for each ST node i in the active graph, there is at most one deferred LD node j per program order. This ensures that the check can be done in finite state.

The last edge annotation property that needs to be checked is as follows: for each node j labeled by a LD$(P, B, \perp)$ operation, there is a forced edge on a path to the first node in the ST order for block B. For each processor P and block B, the checker does not remove from the graph the last node, say j, labeled LD$(P, B, \perp)$ in P's program order, until it has identified the first node, say k, in the ST order for block B. The node k is identified when its ID-set becomes empty and it has no incoming ST order edge. Once k is identified and j is no longer an active node of the graph, the checker rejects if there is no forced node j to k. □

4 Verification of Real-World Protocols

We claim that every real-world sequentially consistent protocol has a finite state witness observer and that the observer can be generated automatically from the protocol. To provide intuition that supports this claim, we first argue informally that a weaker property holds for real-world sequentially consistent protocols, namely that the witness graph corresponding to each protocol run is bandwidth bounded. Later in this section we make this intuition precise, and also show the stronger property that the witness graph corresponding to each run is not only bandwidth bounded but can be generated in finite state from the run.

Let R be a run of a protocol and let R_1 be a prefix of R. Let R_2 be the corresponding suffix of R, so that $R = R_1R_2$. We need to show that if we view the operations of R as nodes of a constraint graph, the number of operations of R_1 with edges to operations of R_2 is bounded. We consider each type of edge in turn. It is easy to see that at most p operations of R_1 have program order edges to operations of R_2, namely the last operation in each processor's program order, if any.

We next consider inheritance edges; here we appeal to our understanding of how real-world sequentially consistent protocols work. These protocols create "views" of a block via ST operations, then copy these views into various protocol storage locations (such as queues, network message packets, or caches of other processors) where they can be read via the LD operation, and eventually delete or overwrite views. Multiple views of a block may exist in the protocol state. For example, one processor may do a ST to a block, thus creating a new view, while stale views of the block still exist in other caches. We call a ST operation of R_1 *inh-active* if one or more copies of the

value (view) written by that ST is stored in the protocol state upon completion of run R_1. If a ST is inh-active, its value may be inherited by LDs in R_2. A key point is that, since the protocol is finite-state, only a constant number of STs of R_1 can be inh-active. Moreover, in real-world protocols, LDs of R_2 that inherit their values from STs of R_1 can only do so from STs of R_1 that are inh-active, because these LDs obtain their values from storage locations of the protocol.

Third, we consider ST order edges. Again, we appeal to a property of real-world protocols here, namely that for all runs, for each block B, the order of STs to B in the run is in fact the same as the order of the STs in the corresponding serial reordering. Thus, if we call ST nodes of R_1 with no outgoing ST order edge *STo-active* nodes, the number of STo-active nodes is at most the number of blocks b of the protocol. (Our class of verifiable protocols will actually be defined in section 4.2 to encompass protocols that do not satisfy this per-block real-time ST reordering property.)

Finally, we consider forced edges. The only LD nodes of R_1 that may have forced edges to STs of R_2 are those LDs which inherit their values from STo-active STs of R_1. For each STo-active operation S of R_1 and each processor P, at most one LD of processor P in R_1 need have a forced edge to a node in R_2, namely the last LD in P's program order that inherits its value from S. (This follows from edge constraint 5 of section 3.1.) Call such a LD operation a *forced-active* LD. Thus, the number of forced-active LDs of R_1 is at most pb. In addition, there may be ST nodes of R_1 that have incoming forced edges from LD nodes in R_2. Call these *forced-active* STs. Each forced-active ST is the immediate successor of an inh-active ST in ST order; thus, the number of forced-active STs is bounded by the number of inh-active STs, and therefore is bounded.

In section 4.1 we define a class of protocols for which the inheritance edges of a constraint graph can be generated in finite state. Protocols in this class have two properties, motivated by our informal arguments above. First, on a LD transition, the value of the LD is obtained from a known storage location of the protocol. Second, by tracking the movement of data among protocol storage locations, it is possible to automatically infer which ST conferred its value to each storage location. Then in section 4.2 we describe conditions under which the ST order edges of a constraint graph can be generated in finite state. In section 4.3, we define a class Γ of protocols that simultaneously satisfy the conditions of sections 4.1 and 4.2. We show that for protocols in Γ, the forced edges of a protocol run can also be generated in finite state, and conclude that such protocols have finite state observers.

4.1 Tracking Labels for Protocols

When a LD is performed by a protocol, how can we tell from which ST it inherits its value? We need to know from which storage location l of the protocol the LD gets its value, and which ST operation conferred its value to location l. We now describe protocols with *tracking labels* which provide an automatic way to infer this knowledge. While real protocol descriptions do not explicitly have tracking labels, for all sequentially consistent protocols known to us, with an appropriate protocol description language the labeling could be generated automatically from the protocol description.

First, we must formalize the concept of storage locations. We have previously defined a protocol as a finite-state machine. Now, let us augment the finite-state machine with a finite number L of storage locations, each able to hold a value chosen from some finite domain. The state of an augmented protocol is defined to be an $(L+1)$-tuple consisting of the state of the finite-state machine, followed by the values in each storage location. Transitions can change the finite-state machine state, as well as assign values (chosen from the finite domain) to all storage locations. Obviously, protocols with storage locations are theoretically exactly as expressive as our original definition of protocols, because the augmented state space is still finite. Explicitly defining storage locations, however, allows capturing how real protocols are described in practice.

In practical descriptions of real memory system protocols, the memory block values that are stored and loaded are not encoded arbitrarily into the protocol state. Instead, they are explicitly held in storage locations, for example, in caches, queues, network messages, memory, etc. Furthermore, the set of operations performed on these storage locations is very restricted. In particular, new values are injected into the protocol only by ST transitions, since the memory system does not create data. All other assignments of memory block values are simply copies from previously-assigned storage locations or perhaps a predefined value indicating an invalid value.

If a protocol uses storage locations in this manner, we can add tracking labels that help determine which ST provided the value for each LD. The tracking labels are of two types.

- Each transition in δ (where δ is the set of transitions on LD and ST operations) is labeled by a location identifier $l \in [1,L]$. Intuitively, the operation is read from or written to location l. Formally, the LD/ST tracking function is a mapping $f : \delta \to [1,L]$.
- For each transition t in δ' (where δ' is the set of transitions on actions other than LD and ST operations) and each $l \in [1,L]$, the copy tracking label, $c_l(t)$, indicates whether the value stored in location l is unchanged by the transition t or whether it has been copied from another location, namely $c_l(t)$. Formally, for each l, there is a copy tracking function $c_l : \delta' \to [1,L]$ (with $c_l(t) = l$ if the value is unchanged).

In typical protocol descriptions arising in practice, LD and ST operations explicitly deal with some storage location in the implementation, and values are copied from one storage location to another in assignment statements, so it would be easy to generate the tracking functions directly from the description. Note that although the actual value of L is enormous for a real memory system — being the sum of the sizes of the entire memory plus any caches, queues, and buffers — in practice, memory system protocols are verified for small values of p, b, and v, so L tends to be moderate as well.

Intuitively, for every run R and location l of a protocol $\mathcal{P}$ with tracking labels, the ST index of l with respect to R is either 0 or is the index of the ST operation from which location l inherits its value upon completion of run R. Formally, the *ST index*, denoted by **ST-index**(R,l), can be defined inductively using the tracking labels as follows.

1. If $|R| = 0$ then **ST-index**$(R,l) = 0$.

2. If $R = R', A$, if the transition t taken on A is a ST operation with tracking label l, and if A is the ith trace operation of R, then **ST-index**$(R, l) = i$. Otherwise, if A is not a LD or ST operation then **ST-index**$(R, l) =$ **ST-index**$(R', c_l(t))$. Otherwise, **ST-index**(R, l) = **ST-index**(R', l).

Example: An example to illustrate ST indexes and tracking labels is given in Figure 4. This example describes a run of an extremely simple protocol with two processors, $P1$ and $P2$, and three blocks, $B1, B2$, and $B3$. Each processor has two cache locations in which values of blocks can be stored (Figure 4(b), Initial State). Thus, there are four locations in all: $P1$'s locations are numbered 1 and 2, and $P2$'s locations are numbered 3 and 4. In the figures, each location contains information about which block is being stored there, if any, and what its value is.

Part (a) of the figure shows a short run R of the protocol. R is of length four and has three ST operations and one "Get-Shared" operation. The Get-Shared operation causes the value of $B1$ stored in location 1 by $P1$ after the first action of R to be copied to location 3 of $P2$; it is reminiscent of how values of blocks can be shared or copied in real protocols, albeit highly simplified. Part (b) of the figure shows the protocol state changing for each action in R. The tracking label of each transition corresponding to each action in run R is also given. The first operation of R, ST$(P1, B1, 1)$ has tracking label 1, indicating that $B1$'s value is written to to location 1. The second operation, ST$(P2, B2, 2)$, has tracking label 4; thus $B2$'s value is written into location 4. The third action of R is not a LD or ST operation and so there are four copy tracking labels $c_1, \ldots, c_4$ associated with this action, one per location. Note that $c_3 = 1$ since the value now stored in location 3 is copied from location 1, but $c_i = i$ for $i = 1, 2$, and 4, since the contents of locations 1, 2, and 4 are unchanged by the Get-Shared action. The last operation of R, ST$(P1, B3, 3)$, has tracking label 1 indicating that block $B1$ is overwritten by $B3$ in location 1. Thus, upon completion of run R, the ST index of each location is given by part (c) of the figure. □

Let $R', \mathrm{LD}(P, B, V)$ be a prefix of R in which the $\mathrm{LD}(P, B, V)$ operation is the jth trace operation of R. Intuitively, if the LD operation gets its value from location l and location l inherits its value from the ith trace operation of R (which must be a ST operation), then (i, j) is an inheritance edge. More precisely, let t be the transition taken on the LD operation, and let the tracking label of t be l. Then, if **ST-index**$(R', l) \neq 0$ the edge (**ST-index**$(R', l), j)$ is an *inheritance edge* of R.

For any run R of a protocol with tracking functions f and c_l, $1 \leq l \leq L$, let the *inheritance graph* of R with respect to these tracking functions be the graph whose nodes are the trace operations of R, numbered by their order in R, and whose edges are the inheritance edges of R. This graph is L-bandwidth bounded, where L is the total number of locations in a state of the protocol. This is because, for any prefix R' of R, at most L ST operations are "active", in the sense that they are indexed in the set $\{$**ST-index**$(R', l)\}$ and thus may be in future inheritance edges. Indeed, we have the following lemma.

Lemma 4.1 *Let $\mathcal{P}$ be a protocol with L locations and tracking functions $f, \{c_l\}$. There is a finite state automaton that, given a run R of $\mathcal{P}$, generates a descriptor of the inheritance graph of R.*

Protocol Run $R = \text{ST}(P1, B1, 1), \text{ST}(P2, B2, 2), \text{Get-Shared}(P2, B1), \text{ST}(P1, B3, 3)$

(a)

Protocol Operation	Tracking Labels	Resulting State
Initial State		$P1$: location / contents: 1 / $\perp$; 2 / $\perp$ — $P2$: location / contents: 3 / $\perp$; 4 / $\perp$
ST($P1, B1, 1$)	1	$P1$: location / contents: 1 / $B1:1$; 2 / $\perp$ — $P2$: location / contents: 3 / $\perp$; 4 / $\perp$
ST($P2, B2, 2$)	4	$P1$: location / contents: 1 / $B1:1$; 2 / $\perp$ — $P2$: location / contents: 3 / $\perp$; 4 / $B2:2$
Get-Shared(P2, B1)	c_1: 1; c_2: 2; c_3: 1; c_4: 4	$P1$: location / contents: 1 / $B1:1$; 2 / $\perp$ — $P2$: location / contents: 3 / $B1:1$; 4 / $B2:2$
ST($P1, B3, 3$)	1	$P1$: location / contents: 1 / $B3:3$; 2 / $\perp$ — $P2$: location / contents: 3 / $B1:1$; 4 / $B2:2$

(b)

ST-index(R,1)	3
ST-index(R,2)	0
ST-index(R,3)	1
ST-index(R,4)	2

(c)

Figure 4: ST Index and Tracking Labels. Part (a) shows a short protocol run of length 4. Part (b) shows the protocol running, with the corresponding tracking labels and updates to the state. Part (c) lists the ST-index of each location with respect to the run.

Proof: The generator generates the graph while executing the protocol on run R, and outputs an extended graph descriptor. Upon transition $t = (q, A, q')$, the generator does the following:

- If A is a ST operation and t has tracking label l then output "l, A". (Recall that this adds a new node to the graph with ID l and label A.)
- For each l, if $c_l(t) \neq l$ then output "add-ID($c_l(t), l$)". (Intuitively, the ST node with ID $c_l(t)$ is being copied to location l, so l is added to the set of IDs for this ST node. More generally, the number of IDs of a ST node equals the number of copies of the ST in the protocol state.)
- If A is a LD operation and t has tracking label l then output "$L+1, A, (l, L+1)$, inh". (This causes a new node with ID $L+1$, labeled A, to be added to the graph, and an inheritance edge to be added into A.)

□

4.2 Finite State ST Reordering

We now consider the ST order edges. Intuitively, in order to guarantee that sequential consistency will be decidable, we restrict the protocols to those in which a bounded-size, finite-state automaton can generate the ST order edges. This restriction guarantees that, in theory, a decision procedure could try all possible automata. In practice, all memory protocols of which we are aware obey very strong versions of this restriction.

Let R be a run of protocol $\mathcal{P}$. A *ST order graph* for R is a graph whose nodes are the trace operations of R, numbered by their order in R. As in section 3.1, for each block B, if there are u ST operations to B in R then there are $u-1$ ST order edges in the graph which define a total order on these u operations.

A *ST order generator* for $\mathcal{P}$ is a finite state automaton that, given run R as input, generates a k-graph descriptor that describes the ST order graph, for some k. Let $|\mathcal{G}|$ denote the number of states in a ST order generator $\mathcal{G}$, and let $|\mathcal{P}|$ denote the number of states in the protocol $\mathcal{P}$. We impose a further requirement that $|\mathcal{G}| \leq |\mathcal{P}|$.

Protocols implemented in practice actually obey the even stronger restriction that $|\mathcal{G}| = 0$. In other words, they obey the *real-time ST reordering* property that for all traces, for each block B, the trace order of STs to B is in fact the same as the corresponding serial reordering. Thus, the ST order generator is trivial.

We believe it unlikely that a real protocol will ever be designed that requires $|\mathcal{G}| > |\mathcal{P}|$. Having $|\mathcal{G}| > |\mathcal{P}|$ would imply that the correct ST order depends on more information about the behavior of the protocol than the protocol itself is able to distinguish. Although the undecidability result of Alur et al. [3] demonstrates the existence of protocols for which no finite-state ST order generator exists, practical protocol designs manage complexity by ensuring that all needed information is always encoded in the protocol state.

One well-known protocol that does require a non-trivial (but still finite-state, and much smaller than our restriction) ST order generator is the Lazy Caching protocol of Afek et al. [2], but this protocol has not been implemented in a real machine. We now

explain briefly why the lazy caching protocol has a finite state ST order generator. In the lazy caching protocol, a processor P can perform a ST(P,B,V) operation at any time, whereupon the pair (B,V) is added to an out-queue for processor P. Another operation of the protocol, namely **memory-write**, removes a (block,value) pair from a queue of a processor, and writes the value into memory. Thus, it is the order in which the **memory-write** steps are performed, rather than the order in which STs are done in real time, that determine the serial ordering of the ST operations to a block.

We next describe a ST order generator for a single block B of the protocol, which, given a run R of the lazy caching protocol, generates a descriptor of the graph whose nodes correspond to the STs to block B in run R and whose edges are the ST order edges for these nodes. While scanning the run from left to right, each time an operation in ST(*,B,V) appears in the run, the generator adds a new node to a graph that it maintains in its internal state. The generator also records the program order of these nodes. The first time that a **memory-write** operation is done to block B at processor P, the generator records that this is done by processor P. On the next memory-write to block B, say by processor P', the generator adds a ST-order edge from the first ST node in the program order of P to the first node in the program order of P'. On each subsequent memory-write to block B, say by processor P'', the generator adds an edge from the last ST in the ST order determined by the edges added so far, to the first node in the program order of P'' which does not yet have an incoming ST order edge. Each time a node or edge is added to the graph, the corresponding node or edge descriptor is output by the generator. Furthermore, once a node has both an incoming and outgoing ST order edge, it can be removed from the internal graph maintained by the generator, thus keeping the internal graph finite.

The overall ST order generator is the cross product of the ST order generators for each block B. The size of the internal graph maintained by the ST order generator is bounded as a function of the sizes of the queues of the processors, since this limits the number of STs that can already have appeared in the run but are not yet serially ordered by a memory-write operation.

4.3 The Γ Protocol Class

Let $\mathcal{P}$ be a protocol. Let $f,\{c_l, 1 \leq l \leq L\}$ be tracking functions and let $\mathcal{G}$ be a ST order generator (with $|\mathcal{G}| \leq |\mathcal{P}|$). With respect to $f,\{c_l\}$, and $\mathcal{G}$, for each run R of $\mathcal{P}$, let $W(R)$ be the graph whose nodes are the trace operations of R. The edges of $W(R)$ are the inheritance edges of the inheritance graph with respect to f and $\{c_l\}$, the ST order edges given by $\mathcal{G}$, the forced edges implied by these inheritance and ST order edges, and the program order edges given by the order of operations in R.

Definition 4.1 *A protocol $\mathcal{P}$ belongs to the class Γ if for some tracking functions $f,\{c_l\}$ and some ST order generator $\mathcal{G}$, for all runs R of $\mathcal{P}$, the graph $W(R)$ is an acyclic constraint graph.*

Theorem 4.1 *Every protocol in Γ has a finite state witness observer.*

Proof: We describe a finite state observer O that, given $\mathcal{P}$ in Γ, along with associated tracking functions $f,\{c_l\}$ and ST order generator $\mathcal{G}$, converts a run R of $\mathcal{P}$ into a

descriptor for a constraint graph $W(R)$.

O adds each LD and ST operation of R to the graph as the operation is read. From Claim 4.1 and section 4.2, the inheritance and ST order edges can be generated in finite state. It is also trivial to generate the program order edges.

It remains to extend the observer so that forced edges are also generated. For this purpose, each node N' labeled by a LD(P,B,V) operation remains in the active graph maintained by the observer until one of the following events occurs. Let the inheritance edge to N' be from node N. (i) Another node, N'', labeled by LD(P,B,V) is added to the graph, along with inheritance edge (N,N''). Node N' can now be removed because there is a path of program order edges from the N' to N''. (ii) A ST order edge from N, say to node S, is present in the graph. In this case, a forced edge is added from N' to S.

The number of LD nodes that need to be in the active graph for the purpose of generating forced edges is bounded by p (the number of processors) times the number of ST nodes with no outgoing ST order edges. The latter number is bounded, since the ST order graph is bandwidth bounded. In addition, if ST node S has an incoming ST order edge (N,S) where the value of the ST labeling N may be read by future LDs, then S must be maintained in the active graph. The number of such ST nodes S is at most L.

Thus, the witness graph is bandwidth bounded, where the bound depends only on $\mathcal{G}$, L, p, and b and does not otherwise depend on R, and so the observer is finite state. □

To summarize, we have shown the following. Let $\mathcal{P}$ be a protocol for which tracking labels can be generated automatically and the real-time ST reordering property holds (or more generally, for which a ST order generator exists). Then, sequential consistency can be verified by an algorithm that first generates the observer from the protocol in a noninterfering fashion (so that the the set of traces of the observer equals those of the protocol) and then uses a model checker (based on our cycle-checker) to verify that every graph descriptor generated by the observer describes an acyclic constraint graph. Note that the checker is independent of the protocol.

4.4 Size of Observer

In order to apply our constraint graph method to the verification of a protocol, the major obstacle will be the size of the observer. In addition to the protocol state, the observer needs to maintain in its state a subgraph of the constraint graph that may have a number of nodes up to the bandwidth bound of that graph. Here, we describe an upper bound on the number of bits of extra state required by the observer, under reasonable assumptions.

First, we bound the bandwidth of the constraint graphs of a protocol $\mathcal{P}$ with L locations. We consider here the case that the protocol has real-time ST ordering, and that the value of a ST is stored in some protocol location at least until the ST following it in ST order has been done. In this case, with respect to a prefix of a run, at most L distinct ST nodes may be actively stored in protocol locations and thus may have future outgoing inheritance edges. Up to pb LD nodes may contribute to the bandwidth needed for generating forced edges. Nodes needed for generation of program order edges and ST order edges are already counted among these nodes, so the total bandwidth is bounded by $L+pb$.

For each active node of the constraint graph, the node label must be stored. This requires up to $\lg p + \lg b + \lg v + 1$ bits. Here lg denotes the ceiling of log to the base 2; 1 bit indicates whether the label is a LD or ST, and parameters P, B, and V are represented using the other bits. Also, IDs for each ST node are needed, in order to generate inheritance edges. An addition $L \lg L$ bits are needed to store IDs.

Edges of the constraint graph must also be represented. If the active nodes are stored in a linear array, no extra storage is needed for edges. Roughly, this is because the nodes can be stored in an order consistent with the partial order of the constraint graph, so that graph edges can be inferred. For example, in the linear array order, a ST to block B is followed (not necessarily contiguously) by LD nodes that inherit its value, and no other ST to the same block separates them, so inheritance edges are completely determined by the linear order.

Thus, an upper bound on the number of bits of extra state needed by the observer (in addition to the protocol state) is $(L + pb)(\lg p + \lg b + \lg v + 1) + L \lg L$ bits. This upper bound is likely to be substantially less than the number of bits in the protocol itself. Real memory system protocols, however, are already roughly at the limits of current model checking tools, so any additional state is problematic in practice. Fortunately, some simple optimizations should help to reduce the size of the observer. For example, the value of a node is needed only to check that each LD gets the same value as the ST from which it supposedly inherits its value. This check can be done independently from the cycle-testing check, thereby saving $\lg v$ bits per node.

5 Future Work

Understanding how the size of the observer can be reduced, perhaps by imposing further assumptions on the class of protocols to be handled, is an important direction for future work from a practical point of view, and will help to relate this work to that of Qadeer [13]. Extending these techniques to other memory models is another important direction of this research.

Experimental results will be needed to assess the applicability of our results in practice. We intend to apply our techniques to substantial memory system protocols using model checking tools and explore means to combat state explosion.

An interesting theoretical question is whether the problem of testing sequential consistency is undecidable for protocols that are bandwidth bounded. The reduction used in the undecidability result of Alur et al. [3] exploits protocols that are not bandwidth bounded.

Finally, we note that our method can also be used for testing that a particular run of a protocol does not violate sequential consistency, building on the approach proposed by Gibbons and Korach [7]. The finite-state observer and checker could be simulated together with detailed implementation descriptions that are too complex for formal verification.

Acknowledgments

We thank Mark Hill, Dan Sorin, Manoj Plakal and the other members of the Wisconsin Multifacet group for sharing their insights and intuition about proving sequential consistency.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.

[2] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1), January 1993.

[3] Rajeev Alur, Ken McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. In *Eleventh Symposium on Logic in Computer Science*, pages 219–228. IEEE, 1996.

[4] Tim Braun, Anne E. Condon, Alan J. Hu, Kai S. Juse, Marius Laza, Michael Leslie, and Rita Sharma. Proving sequential consistency by model checking. In *International High-Level Design, Validation, and Test Workshop*. IEEE, 2001.

[5] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Workshop on Logics of Programs*, pages 52–71, May 1981. Published 1982 as Lecture Notes in Computer Science Number 131.

[6] Anne E. Condon and Alan J. Hu. Automatable verification of sequential consistency. In *13th Symposium on Parallel Algorithms and Architectures*, pages 113–121. ACM, 2001.

[7] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, August 1997.

[8] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In *Computer-Aided Verification: 11th International Conference*, pages 301–315. Springer, 1999. Lecture Notes in Computer Science Vol. 1633.

[9] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, pages 28–34, August 1998.

[10] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *ACM Transactions on Computer*, 28(9):690–691, September 1979.

[11] Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkedem, and Ganesh Gopalakrishnan. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In *Computer-Aided Verification: 10th International Conference*, pages 464–476. Springer, 1998. Lecture Notes in Computer Science Vol. 1427.

[12] M. Plakal, D. Sorin, A. Condon, and M. Hill. Lamport Clocks: Verifying a directory cache coherence protocol. In *Symposium on Parallel Algorithms and Architectures*, pages 67–76, 1998.

[13] Shaz Qadeer. On the verification of memory models of shared-memory multiprocessors. Research Report 175, Compaq Systems Research Center, 2001.