Theory Comput. Systems **31**, 355–376 (1998)

Theory of Computing Systems
© 1998 Springer-Verlag New York Inc.

Verification of Cache Coherence Protocols by Aggregation of Distributed Transactions*

S. Park and D. L. Dill

Department of Computer Science, Stanford University,
Stanford, CA 94305, USA
spark@cs.stanford.edu
dill@cs.stanford.edu

Abstract. This paper presents a method to verify the correctness of protocols and distributed algorithms. The method compares a state graph of the implementation with a specification which is a state graph representing the desired abstract behavior. The steps in the specification correspond to atomic transactions, which are not atomic in the implementation.

The method relies on an *aggregation function*, which is a type of abstraction function that aggregates the steps of each transaction in the implementation into a single atomic transaction in the specification. The key idea in defining the aggregation function is that it must complete atomic transactions which have committed but are not finished.

This paper illustrates the method on a directory-based cache coherence protocol developed for the Stanford FLASH multiprocessor. The coherence protocol consisting of more than a hundred different kinds of implementation steps has been reduced to a specification with six kinds of atomic transactions. Based on the reduced behavior, it is very easy to prove crucial properties of the protocol including data consistency of cached copies at the user level. This is the first correctness proof verified by a theorem-prover for a cache coherence protocol of this complexity. The aggregation method is also used to prove that the reduced protocol satisfies a desired memory consistency model.

* This research was supported by the Advanced Research Projects Agency through NASA Grant NAG-2-891.

1. Introduction

1.1. *Basic Idea*

Protocols for distributed systems often simulate atomic transactions in environments where atomic implementations are impossible. This observation can be exploited to make formal verification of protocols and distributed algorithms using a computer-assisted theorem-prover much easier than it would otherwise be [34]. Indeed, the techniques described below have been used to verify safety properties of significant examples: the cache coherence protocol for the FLASH multiprocessor which is currently being designed at Stanford [15], [20], a majority consensus algorithm for multiple copy databases [41], [18], and a distributed list protocol [9].

The method proves that an implementation state graph is consistent with a specification state graph that captures the abstract behavior of the protocol, in which each transaction appears to be atomic. The method involves constructing an abstraction function which maps the distributed steps of each transaction to the atomic transaction in the specification. We call this *aggregation*, because the abstraction function reassembles the distributed transactions into atomic transactions.

This method addresses the primary difficulty with using theorem-proving for verification of real systems, which is the amount of human effort required to complete a proof, by making it easier to create appropriate abstraction functions. Although our work is based on using the PVS system from SRI International [33], the method is useful with other mechanical theorem-provers, or manual proofs.

Although finite-state methods (e.g., [32], [10], [17], and [19]) can solve many of the same problems with even less effort, they are basically limited to finite-state protocols. Finite-state methods have been applied to non-finite-state systems in various ways [38], but these techniques typically require substantial pencil-and-paper reasoning to justify. Moreover, it is not obvious how to apply these extensions to the examples we verified using aggregation. Theorem-provers make sure that such manual reasoning is indeed correct, in addition to making available the full power of formal mathematics for proof, so they can routinely deal with problems that cannot yet be solved by any finite-state methods.

For our method to be applicable, the description must have an identifiable set of transactions. Each transaction must have a unique *commit point* [16], from which a state change cannot be aborted (usually, it is the point at which a state change first becomes visible to the specification). The most important idea in the method is that the *aggregation function* can be defined by *completing* transactions that have committed but not yet completed. In general, the steps to complete separate transactions are independent, which simplifies the definition of this function. In our experience, this guideline greatly simplifies the definition of an appropriate aggregation function.

The same idea of aggregating transactions can be applied to reverse engineer a specification where none exists, because the specification with atomic transactions is usually consistent with the intuition of the system designer. We extract a specification model which performs transactions atomically at their commit steps in the implementation, and does nothing at other steps. The extracted specification provides an illusion that the transactions take effect instantaneously at the commit steps in the implementation.

If the extracted specification is not obviously complete or correct, it can instead be regarded as a model of the protocol having an enormously reduced number of states. The amount of reduction is much more than other reduction methods used in model checking, such as partial order reduction, mainly because the state variables of the reduced system are only those relevant to the specification, without variables such as local states and communications buffers.

The major contribution of this work is to reduce the effort required to prove the correctness of certain classes of cache coherence protocols (and possibly other types of distributed algorithms). The methodology requires extracting or defining transaction-oriented specifications. Once the specifications are in this form, aggregation provides a simple method for defining abstraction functions based on completing unfinished transactions across distributed processing elements. The effectiveness of aggregation is shown by the example we present below: FLASH is the most complex multiprocessor cache coherence protocol that has been formally verified using a theorem-prover.

1.2. *Related Work*

1.2.1. *Verification of Cache Coherence Protocols.* One widely used technique for validating cache coherence protocols is finite-state methods (e.g., model checking). Finite-state methods enumerate the states of the reachable state graph of the system, searching for states that violate a specified property [31], [5], [40], [32], [17], [19]. These methods suffer from the state explosion problem: the number of states for nontrivial numbers of processors and cache lines is very large. Another problem with model checkers is that it is very difficult to specify correctness conditions of the protocol using notations such as Murφ or temporal logic. The specification is the corresponding memory model of the protocol so it is required to encode a full memory model in temporal logic.

Symbolic state models proposed by Pong and Dubois [38], [37] reduce the state explosion problem by using symbolic states which abstract away from the exact number of configurations of replicated identical components by recording only whether there are zero, one, or more than zero replicated components. However, as in model checking, there still remains the problem of specifying the protocol: It is not easy to find a set of properties (in their notation), which completely describes the correct behavior of the protocols. Moreover, their method requires the user to write an abstract description of the protocol to be verified, which raises another verification problem: Are the abstract description and the actual protocol equivalent?

Another approach to formal verification is computer-assisted theorem-proving. Theorem-provers make available the full power of formal mathematics for proof, so they can routinely deal with problems that cannot yet be solved by any finite-state methods. However, the major problem with theorem-proving is that considerable labor is required. Consequently, previous theorem-proving approaches have not been able to verify a problem of the scale of a full multiprocessor cache coherence protocol. The most significant result before our work is a manual proof of “lazy caching,” a simple and abstract cache coherence algorithm [2], [13], [21]. It should be noted that using a theorem-prover typically *increases* the labor required to complete a proof compared with manual proof—however, the results are much more likely to be correct.

1.2.2. *Abstraction Function.* The idea of using abstraction functions to relate implementation and specification state graphs is very widely used, especially when manual or automatic theorem-proving is used [30], [29], [22] (indeed, whole volumes have been written on the subject [8]). The idea has also been used with finite-state techniques [19], [11].

Ladkin et al. [21] have used a refinement mapping [1] to verify a simple caching algorithm. Their refinement mapping hides some implementation variables, which may have the effect of aggregating steps if the specification-visible variables do not change. Our aggregation functions generalize on this idea by merging steps even when specification-visible variables change more than once. This happens in most cache coherence protocols for distributed systems. For example, in the FLASH protocol described later in this paper, the cache states (specification variables) can change twice during a write transaction: first, if another cache has a copy in exclusive state, the state must transition to invalid; then the requesting cache state changes from invalid to exclusive when the cache receives data. In this case, our aggregation function modifies the cache state and data to complete the transaction.

A more limited notion of aggregation than ours is found in [24] and [25], where a state function undoes or completes an unfinished process. The method only aggregates sequential steps within a local process. The idea of an aggregated transaction has been used to prove a protocol for database systems [36], where aggregation is obtained also in a local process by showing the commutativity of actions from simple syntactic analysis. Ours is the first method that aggregates steps across distributed components.

In program verification, proofs can be simplified by pretending that a statement is atomic if its execution contains at most one access of a shared variable. This is the so-called "single-action rule" [28], [12], [26]. The single-action rule is generalized in [27]. This method classifies program statements as "left-movers" or "right-movers" depending on their commutative properties. Using these properties, the statements are permuted to obtain a coarser-grained version of the program, for which safety properties can be checked.

Cohen used an idea similar to aggregation to prove global progress properties by combining progress properties of local processes [6]. The idea of how to construct our aggregation function was inspired by a method of Burch and Dill for defining abstraction functions when verifying microprocessors [3].

1.3. *Contents of the Paper*

This paper is organized as follows. Section 2 defines the verification goal and Section 3 presents the verification method. Section 4 describes the FLASH cache coherence protocol in two ways: in terms of transactions and per-node-based steps. Section 5 illustrates the method on the FLASH protocol. Using the reduced model obtained by aggregation, Section 6 proves that one of the two distinct modes supported by the protocol implements a sequential consistency memory model. Finally, Section 7 concludes and proposes possible lines of future research.

2. Verification Objective

The goal of formal verification is usually to show that two alternative descriptions of the same behavior are consistent. The notion of consistency varies according to the details

of the verification problem. The aim of this section is to define the objective of formal verification precisely for transaction-oriented protocols. To do so, we must first describe our model of these protocols.

The verification method begins with two logical descriptions: a description of the state graph of the implementation, and a description of the state graph of the specification. The implementation description contains a set of *state variables*, which is partitioned into *specification variables* and *implementation variables*. The set Q of *states* of the implementation is the set of assignments of values to state variables. The description of the implementation also includes a logical formula defining the relation between a state and its possible successors. The next state choices are represented by a set of functions $\mathcal{F}$ from states to states. Each function in $\mathcal{F}$ maps a given implementation state to a possible successor state. The implementation is nondeterministic if $\mathcal{F}$ has more than one function.

We could also have represented the next state choices as a relation, but the "set of functions" representation is much more suitable for the verification method we describe below. It is also easy to represent protocols in this way. For example, languages of iterated guarded commands, such as Murphi [10] and UNITY [4] can be translated directly into the above representation.

The description of the specification state graph is similar to the implementation description. A specification state is an assignment of values to the *specification variables* of the implementation (implementation variables do not appear in the specification). Also, every state in the specification has a transition to itself, which we call an *idle* step. The idle steps are necessary to represent implementation steps that do not change specification variables.

The verification method relies on there being a set of *transactions* which the computation is supposed to implement. A transaction is atomic at the specification level, meaning that it occurs in a single state transition in the specification. However, transactions in the implementation are nonatomic; they may involve many steps that are executed in several different components of the implementation.

Each transaction in the implementation must have an identifiable *commit step*. Intuitively, when tracing through the steps of a transaction, the commit step is the implementation step that first makes an inevitable change in the specification variables. Implementation states that occur before the transaction or during the transaction but before the commit step are called *precommit states* for that transaction. The transaction is *complete* when the last specification variable change occurs. The states after the commit step but before the completion of the transaction are called *postcommit states* for the transaction. A state where every committed transaction has completed is called a *clean* state.

Formally, all of the above concepts can be derived once the postcommit states are known for each transaction. The precommit states for the transaction are the states that are not postcommit; the commit step for a transaction is the transition from a precommit state to a postcommit state for that transaction; and the completion step is the transition from a postcommit state to a precommit state. A state is clean if it is a precommit state for *every* transaction.

We can now describe our objective in formally verifying transaction-oriented protocols. We suppose that the designer of a protocol understands how the implementation steps correspond to the transactions he or she is trying to implement. This correspondence is represented formally as a function that maps each implementation step to the

transaction that it implements. We call this the *transaction mapping* M. Specifically, the transaction mapping maps each commit step to the appropriate atomic transaction; steps that are not commit steps for any transaction map to the *idle* specification step.

There is a natural correspondence between an implementation state q and the specification state $proj(q)$ that is obtained by projecting away the implementation variables (and leaving the specification variables untouched). When q is clean, $proj(q)$ is the specification state that q implements. However, when q is not clean, $proj(q)$ is not necessarily meaningful, and the specification state implemented by q is nonobvious.

An execution of the implementation consists of a clean initial implementation state q and a finite sequence σ of implementation steps from $\mathcal{F}$. The terminal implementation state is $[\sigma](q)$, where $[\sigma]$ is the functional composition of the elements of σ in sequence. Corresponding to the implementation execution is an execution of the specification. The initial state of the execution of the specification is $proj(q)$, and the sequence of specification steps is $M(\sigma)$ (the sequence formed by applying the transaction mapping M elementwise to σ). The terminal state of the specification sequence is $[M(\sigma)](proj(q))$ (the functional composition of the specification steps, applied to the initial specification state).

Our verification goal is to show that if the terminal implementation state is clean, projecting it onto the specification variables yields the terminal specification state. In other words,

$$\begin{aligned}&\forall q \in Q, \quad \forall \sigma \in \mathcal{F}^*:\\ &Clean(q) \wedge Clean([\sigma](q)) \Rightarrow proj([\sigma](q)) = [M(\sigma)](proj(q)).\end{aligned} \tag{1}$$

This definition of the verification objective may raise several questions. First, why do we only require that clean states map to specification states? Extending the definition to unclean states seems to result in more the definition of an aggregation function that we describe below. The goal of this definition is to establish a simpler and more basic notion of correctness that can be used to justify the soundness of the aggregation method.

Second, what if an implementation sequence does not terminate in a clean state? If we verify the above property for all finite sequences, there will either be an extended sequence ending in a clean state where we can check the result, or there is no such sequence. Since it is always possible to stop initiating transactions, the latter condition indicates there is no way to complete the unfinished transactions (e.g., because of deadlock or livelock). In either case, a transaction is not executed incorrectly, although it may fail to complete. We have limited our attention to safety properties in this paper, so failure to complete transactions is beyond our scope of concerns. (Although we have not done it, we believe that a more general treatment of liveness as well as safety would not be difficult.)

3. The Verification Method

In this section we define aggregation functions and show that the existence of an aggregation function is a sufficient condition implying property (1) above. An aggregation function consists of two parts: a *completion function* which changes the state as though the transaction had completed, and a *projection* which hides the implementation variables, leaving only the specification variables.

An aggregation function *aggr* maps implementation states to specification states. In addition, *aggr* must satisfy two conditions. First, it must have a commutative property:

$$\forall q \in Q, \quad \forall N \in \mathcal{F}: \qquad aggr(N(q)) = M(N)(aggr(q)). \tag{2}$$

This resembles the basic definition of an abstraction function, so we consider an aggregation function to be a particular type of abstraction function.

The second property is that an aggregation function must be the same as projection for clean states:

$$\forall q \in Q: \qquad Clean(q) \Rightarrow aggr(q) = proj(q). \tag{3}$$

A clean state is one where all transactions are complete. Therefore, the completion function in *aggr* should have no effect, leaving only the projection.

Existence of an aggregation function implies the verification objective (1). Because the aggregation function has the commutative property (2), it is easy to prove by induction that

$$\forall q \in Q, \quad \forall \sigma \in \mathcal{F}^*: \qquad aggr([\sigma](q)) = [M(\sigma)](aggr(q)).$$

This and property (3) directly imply verification objective (1).

Once a purported aggregation function has been defined, the user must prove that it meets the commutative requirement (2). The proof can be done completely automatically or consists of a sequence of standard steps.[1] The initial $\forall q$ and $\forall N$ can be eliminated automatically by substituting new symbolic constants throughout (when we do this in this presentation, we will not change the name of the quantified variable). This yields a subgoal of the form

$$(N \in \mathcal{F}) \Rightarrow aggr(N(q)) = M(N)(aggr(q)). \tag{4}$$

The number of subgoals is equal to the number of transition functions in the implementation. In most cases, the required specification step $M(N)$ is the idle step; indeed, the only nonidle step is that which corresponds to the commit step in the implementation. We have no global strategy for proving these theorems, although most are very simple.

The above discussion omits an important point, which is that not all states are worthy of consideration. Theorem (2) will generally not hold for some absurd states that cannot actually occur during a computation. Hence, it is usually necessary to provide an *invariant* predicate, which characterizes a superset of all the reachable states. If the invariant is *Inv*, Theorem (2) can then be weakened to

$$\forall q \in Q, \quad \forall N \in \mathcal{F}: \qquad Inv(q) \Rightarrow aggr(N(q)) = M(N)(aggr(q)). \tag{5}$$

In other words, *aggr* only needs to commute when q satisfies the *Inv*. Use of an invariant incurs some additional proof obligations. First, we must prove that the initial states of the protocol satisfy *Inv*, and, second, that the implementation transition functions all preserve *Inv*.

---

[1] We base this comment on our use of the PVS theorem-prover, but we believe the same basic method would be used with others.

4. FLASH Cache Coherence Protocol

The aggregation method has been used to verify the cache coherence protocol used in the Stanford FLASH multiprocessor [20], [15]. This section informally describes that protocol.

The cache coherence protocol is directory-based so that it can support a large number of distributed processing nodes. Each cache line-sized block in memory is associated with a *directory header* which keeps information about the line. For a memory line, the node on which that piece of memory is physically located is called the *home*; the other nodes are called *remote*. The home maintains all the information about memory lines in its main memory in the corresponding directory headers.

The system consists of a set of nodes, each of which contains a processor, caches, and a portion of the global memory. The distributed nodes communicate using asynchronous messages through a point-to-point network. The state of a cached copy is in either *invalid*, *shared* (readable), or *exclusive* (readable and writable).

4.1. *Informal Description of the Protocol*

If a read miss occurs in a processor, the corresponding node sends out a GET request to the home (this step is not necessary if the requesting processor is in the home). Receiving the GET request, the home consults the directory corresponding to the memory line to decide what action the home should take. If the line is *pending*, meaning that another request is already being processed, the home sends a NAK (negative acknowledgment) to the requesting node. If the directory indicates there is a dirty copy in a remote node, then the home forwards the GET to that node. Otherwise, the home grants the request by sending a PUT to the requesting node and updates the directory properly. When the requesting node receives a PUT reply, which returns the requested memory line, the processor sets its cache state to *shared* and proceeds to read.

For a write miss, the corresponding node sends out a GETX request to the home. Receiving the GETX request, the home consults the directory. If the line is *pending*, the home sends a NAK to the requesting node. If the directory indicates there is a dirty copy in a third node, then the home forwards the GETX to that node. If the directory indicates there are shared copies of the memory line in other nodes, the home sends INVs (invalidations) to those nodes. At this point, the protocol depends on which of two modes the multiprocessor is running in: EAGER or DELAYED. In EAGER mode, the home grants the request by sending a PUTX to the requesting node; in DELAYED mode, this grant is deferred until all the invalidation acknowledgments are received by the home. If there are no shared copies, the home sends a PUTX to the requesting node and updates the directory properly. When the requesting node receives a PUTX reply which returns an exclusive copy of the requested memory line, the processor sets its cache state to *exclusive* and proceeds to write.

During the read miss transaction, an operation called a sharing write-back is necessary in the following "three hop" case. This occurs when a remote processor in node R_1 needs a shared copy of a memory line, an exclusive copy of which is in another remote node R_2. When the GET request from R_1 arrives at the home H, the home consults the directory to find that the line is dirty in R_2. Then H forwards the GET to R_2 with the

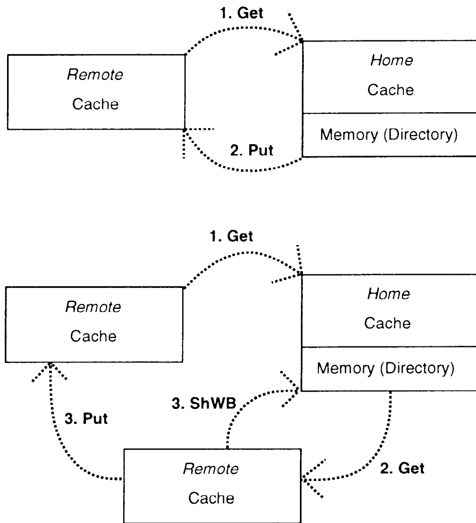

Fig. 1. Processing of a read miss (a GET request) in the FLASH protocol.

source of the message *faked* as R_1 instead of H. When R_2 receives the forwarded GET, the processor sets its copy to *shared* state and issues a PUT to R_1. Unfortunately, the directory in H does not have R_1 on its sharer list yet and the main memory does not have an updated copy when the cached line is in the shared state. The solution is for R_2 to issue an SWB (sharing write-back) conveying the dirty data to H with the source *faked* as R_1. When H receives this message, it writes the data back to main memory and puts R_1 on the sharer list. Figure 1 shows the processing of a read miss in the protocol.

When a remote node receives an INV, it invalidates its copy and then sends an acknowledgment to the home. There is a subtle case with an invalidation. A processor which is waiting for a PUT reply may get an INV before it gets the shared copy of the memory line, which is to be invalidated if the PUT reply is delayed. In such a case, the requested line is marked as invalidated, and the PUT reply is ignored when it arrives.

A valid cache line may be replaced to accommodate other memory lines. A shared copy is replaced by issuing a replacement hint to the home, which removes the remote from its sharers list. An exclusive copy is written back to main memory by a WB (write-back) request to the home. Receiving the WB, the home updates the line in main memory and the directory properly.

4.2. *Detailed Description of the Protocol*

Each cache line-sized block in main memory is associated with a directory header which keeps information about the line. The directory header consists of several Boolean flags: Local, Dirty, Pending, Head_Valid, and List; a pointer to another node Head_Pointer; a

set of pointers Sharer_List;[2] and the number of sharers in Real_Pointers. The Local bit indicates if the local processor (in the home) contains a cached copy of the line in either shared or exclusive state. The Dirty bit is set if the home thinks that there is a dirty copy of the line in the system. The Pending bit is set if the current request for the memory line is being processed by a third node. The Head_Valid bit indicates whether the Head_Pointer contains a valid pointer to a node. The Head_Pointer entry is simply a cache pointer that is stored in the directory header as an optimization. It keeps a pointer to a remote cache with a dirty copy if there is one, or one of the nodes with a shared copy. The List bit indicates whether Sharer_List contains one or more pointers. Sharer_List is represented abstractly as a set of pointers to the nodes that have a shared copy of the memory line. Real_Pointers contains a count of the number of sharers in the Sharer_List. This count excludes the Head_Pointer and is mainly used to count invalidation acknowledgments.

The FLASH protocol consists of a set of rules which are called *handlers*. Each handler is prefixed with NI (network interface) or PI (processor interface) to indicate where the requests are generated from. PI handlers are initiated by a requesting processor and NI handlers are initiated by a message from the network. The additional notation "Local" or "Remote" indicates whether the processing node is the home of the requested memory address or not. In the following, some handlers of the protocol for processing a read miss are listed; the rest of the handlers are shown in the Appendix.

- **PI.Local.Get:** this handler describes actions of the home when the local processor needs a shared copy of a memory line. If Pending,[3] the local processor is NAKed. Otherwise, if Dirty, the home sends a GET request to Head_Pointer and Pending is set. Otherwise, the data in main memory is copied into the local cache (in *shared* state) and Local is set.
- **NI.Remote.Get:** this handler describes actions of a remote node receiving a GET request. If the cached data in the remote node is in *exclusive* state, it is changed to *shared* and the node sends a PUT reply to the source of the GET (and also SWB to the home if the source is not the home). Otherwise, the node sends an NAK to the source and an NAKC (NAK clear) to the home.
- **NI.Local.Put:** this handler processes a PUT reply to the home. Local is set, Dirty and Pending are reset, and the shared copy is put into the local cache.
- **NI.SharingWriteback:** this handler describes actions of the home receiving an SWB. Dirty and Pending are reset, List is set, Real_Pointers is incremented, the source is added to Sharer_List, and the data is written back into main memory.

5. Verification by Aggregating Distributed Transactions

Using the aggregation method, we have formally verified the protocol at the level of its formal description [35]. The protocol consisting of more than a hundred different implementation steps has been reduced to a model with only six kinds of atomic transac-

[2] Actually, the set of pointers is implemented as a linked list for sharers (within the home) by dynamic pointer allocation [39].

[3] That is, the Pending bit is set in the directory.

tions. Based on the reduced atomic behavior, it is very easy to reason about the protocol, checking safety properties and data consistency of cached copies.

In the following, we illustrate how the protocol is reduced to an atomic model by an aggregation function. The detailed proofs are confirmed by a theorem-prover and some techniques to simplify the proof are presented.

5.1. *Extracting Reduced Model of the Protocol*

Verification requires two descriptions of the same behavior: an implementation and a specification. Sometimes, there is an a priori specification as in the memory model verification in the next section. However, in most practical instances, there is only an implementation. In such cases, we extract a reduced model of the implementation using aggregation. The reduced model concisely captures the behavior of the implementation and serves as a specification.

Recall that to use the aggregation method, we first decide which state variables should be considered specification variables. In cache coherence protocols, the consistency of multiple copies of a memory line is a function of the values and states of cached copies, and the corresponding value in main memory. Therefore, the specification variables should be the state variables representing the data and states of cached copies and the data in main memory.

We construct a reduced model of the protocol, which we use for a specification. The procedure is to trace through a transaction: (1) concatenating the implementation steps, (2) simplifying by substituting values forward through intermediate assignments, and (3) eliminating statements that only change implementation variables. The reduced model is a much simpler version of the protocol which reads and writes only the specification variables. The specification steps update the values and states of cached copies in multiple nodes *atomically*.

The reduced model of the protocol is shown in Table 1. Atom-WB invalidates an exclusive copy and writes the data back to main memory atomically. Atom-INV invalidates a shared copy. Interestingly, the step may have to invalidate an invalid copy in the case where a cache line has been replaced but the sharer list in the directory header is not updated yet. There are two kinds of transactions for a read miss: Atom-Get-1 corresponds to the transaction that the home grants a shared copy to the requester when there is no dirty copy of the memory line; Atom-Get-2 corresponds to the transaction that a node with an exclusive copy grants a shared copy. For the transaction for a write miss, Atom-GetX-1 sends an exclusive copy of a memory line from the home if there are no other copies in remotes; Atom-GetX-2 transfers an exclusive ownership from a dirty node to the requester. The FLASH protocol does not write back when an exclusive ownership is transferred.

5.2. *Commit Steps*

To define the aggregation function *aggr*, we should first identify the commit steps of each transaction in the protocol. The transaction for a read miss begins with sending a GET request to the home. Depending on the directory state of the memory line, the request may be forwarded to a remote node which contains a dirty copy of the line.

Table 1. Reduced model of the FLASH protocol obtained by aggregation of distributed transactions.

	Condition	Atomic action
Atom-WB (p, a)	cache$[p][a]$.state = exclusive	cache$[p][a]$.state := invalid memory$[a]$:= cache$[p][a]$.data
Atom-INV (p, a)	cache$[p][a]$.state = invalid $\vee$ cache$[p][a]$.state = shared	cache$[p][a]$.state := invalid
Atom-Get-1 (p_2, a)	$\neg\exists i$: cache$[i][a]$.state = exclusive	cache$[p_2][a]$.state := shared cache$[p_2][a]$.data := memory$[a]$
Atom-Get-2 (p_1, p_2, a)	cache$[p_1][a]$.state = exclusive $\wedge\ p_1 \neq p_2$	memory$[a]$:= cache$[p_1][a]$.data cache$[p_1][a]$.state := shared cache$[p_2][a]$.state := shared cache$[p_2][a]$.data := cache$[p_1][a]$.data
Atom-GetX-1 (p_2, a)	$\neg\exists i$: cache$[i][a]$.state = exclusive $\wedge$ ($\neg\exists i$: cache$[i][a]$.state = shared $\wedge\ i \neq p_2$)[1]	cache$[p_2][a]$.state := exclusive cache$[p_2][a]$.data := memory$[a]$
Atom-GetX-2 (p_1, p_2, a)	cache$[p_1][a]$.state = exclusive $\wedge\ p_1 \neq p_2$	cache$[p_1][a]$.state := invalid cache$[p_2][a]$.state := exclusive cache$[p_2][a]$.data := cache$[p_1][a]$.data

[1] Additional constraint for DELAYED mode.

These steps do not modify the specification variables, so they are precommit steps of the transactions. The transaction for a write miss is similar.

The commit step occurs when the home, or a remote node with an exclusive copy, sends a PUT or PUTX reply, granting the request. In each case, the state of the cache line in the granting node or main memory is modified. Any future request for the memory line is processed as if the committed reply had been processed by the requesting node, even if that has not actually happened. For instance, if a GETX request arrives at the home from R_1 right after a grant of an exclusive ownership to R_2, the home forwards the GETX to R_2 regardless of whether the PUTX sent to R_2 has arrived there or not. If a request is NAKed, then there is no change in specification variables by the transaction, so, in effect, no action occurs.

The write-back transaction begins with invalidating an exclusive copy and sending a WB request to the home. This is the commit step of the transaction because the state of cached data, a part of the specification variables, is already updated at this moment and the write-back request cannot be denied by the home. The invalidation transaction is similar to this case.

5.3. *Aggregation Function*

Once a transaction is committed, the aggregation function *aggr* simulates the postcommit steps of the transaction to complete it. The postcommit steps in the protocol are the steps that process a PUT and SWB for a read miss, and that process a PUTX for a write miss, and that process a WB for a write-back. Therefore, to complete all the committed transactions, the *aggr* should process all the messages of types PUT, PUTX, WB, and SWB in the network.

The key question is how to complete all committed transactions in the current state, especially since the number of distributed nodes, and hence the number of committed transactions, is unknown. We first define a *per-node* completion function for a node indexed by variable i; the per-node function is then *generalized* to define a completion function for all of the nodes in the system.

It is quite simple to complete a committed transaction for a particular node. If a PUT message destined for node i exists, the transaction for a read miss in node i must be completed by simulating the effect of node i processing the PUT message it receives at the end of the transaction: putting the data in the message into its cache and setting the state to *shared*.[4] The transaction for a write miss is similarly completed by processing a PUTX to node i. If node i is the home, there are two more kinds of messages possibly generated at commit steps: SWB and WB. Note that there exists at most one message of the four types destined to a particular node at any time.

This processing changes values and states of cached copies, and values in main memory. Changes to implementation variables, such as removing messages from the network, and resetting the waiting flag in the processor can be omitted from the completion function, as they do not affect the corresponding specification state. All of this computation is done solely in node i, without the involvement or interference of other nodes.

It is easy to generalize the per-node completion function to a completion function for all of the nodes because the completions do not interact. The global implementation state is an array of cell state records, indexed by the cell indices. Let $cc(q[i])$ be a completion function for cell i, which modifies the state variables for i in the record $q[i]$, and returns a new record of the state variables as modified by the completion of the transaction.

The completion functions are simply performed in parallel. If $cc(q[i])$ completes committed transactions on node i, the completion function for all nodes is $\lambda q.\lambda i.cc(q[i])$. When this function is supplied a state q, it returns $\lambda i.cc(q[i])$,[5] which is an array of the completed node states, i.e., the desired clean global state. The aggregation function is simply the completion function, followed by a projection which eliminates all implementation variables.

5.4. *Specification Steps*

The specification steps corresponding to implementation steps are simply idle steps for precommit steps and postcommit steps. The only nonidle steps are those which correspond to the commit steps of transactions. A complete assignment of atomic transactions of the reduced model to the implementation steps of the protocol is shown in Table 2. Each pair corresponds to a subgoal (4) in Section 3. The condition of an atomic step should be true at the corresponding commit step in the implementation, which is included in the invariant of the system.

---

[4] Processing a PUT message should include the action of invalidation when the line is already invalidation marked due to an INV arrived earlier than the PUT, which results in a negative acknowledged transaction. See the lines suffixed with "inv" in Table 2.

[5] $\lambda i.cc(q[i])$ is a function, which when applied to a particular value of i, say i_0, returns $cc(q[i_0])$, which is the completed state for node i_0. This is effectively the same as indexing into an array of completed node states.

Table 2. Correspondence of protocol steps with atomic transactions (EAGER mode).

Protocol step at node p	Atomic transaction (specification)
PI.Local.Get.else	ε
PI.Local.Get.put	Atom-Get-1(*home*)
PI.Remote.Get	ε
PI.Local.GetX.else	ε
PI.Local.GetX.putx	Atom-GetX-1(*home*)
PI.Remote.GetX	ε
PI.Local.PutX	Atom-WB(*home*)
PI.Remote.PutX	Atom-WB(p)
PI.Local.Replace	Atom-INV(*home*)
PI.Remote.Replace	Atom-INV(p)
NI.NAK	ε
NI.NAK.Clear	ε
NI.Local.Get.else	ε
NI.Local.Get.put	Atom-Get-1(GET.src)
NI.Local.Get.put.ex[1]	Atom-Get-2(*home*, GET.src)
NI.Local.Get.put.inv[2]	Atom-Get-1(GET.src); Atom-INV(GET.src)
NI.Local.Get.put.ex.inv[1,2]	Atom-Get-2(*home*, GET.src); Atom-INV(GET.src)
NI.Remote.Get.else	ε
NI.Remote.Get.put	Atom-Get-2(p, GET.src)
NI.Remote.Get.put.inv[2]	Atom-Get-2(p, GET.src); Atom-INV(GET.src)
NI.Local.GetX.else	ε
NI.Local.GetX.putx	Atom-GetX-1(GETX.src)
NI.Local.GetX.putx.ex[1]	Atom-GetX-2(*home*, GETX.src)
NI.Remote.GetX.else	ε
NI.Remote.GetX.putx	Atom-GetX-2(p, GETX.src)
NI.Local.Put	ε
NI.Remote.Put	ε
NI.Local.PutXAcksDone	ε
NI.Remote.PutX	ε
NI.Inv	Atom-INV(p)
NI.InvAck	ε
NI.WB	ε
NI.FAck	ε
NI.ShWB	ε
NI.Replace	ε

[1]Decomposed handlers when the home holds an exclusive copy.
[2]Decomposed handlers when the requesting node is invalidation marked.

In the FLASH protocol, some handlers perform commit steps in some cases and not in others. In order to establish the necessary correspondence between implementation steps and specification steps in a proof of property (4), we need to split these handlers into multiple transition functions, each of which either always commit or never commit. For example, the PI.Local.Get handler simply NAKs the local processor if the requested line is pending (precommit step), or sends a request to a remote if there is a dirty copy (precommit step), otherwise, it updates the state and data of the local cache which are

specification variables (commit step). In the first two cases, the reduced model should take idle steps, but in the last case, an Atom-Get transaction should be taken.

The PI.Local.Get handler is decomposed into two different transition functions PI.Local.Get.else and PI.Local.Get.put with disjoint enabling conditions, where the first includes the precommit steps, and the latter corresponds to the commit step. Other handlers are decomposed in the same manner, if necessary. In Table 2, the protocol steps named with suffix "ex" (with superscript 1) correspond to the decomposed handlers when the home holds an exclusive copy. The protocol steps named with suffix "inv" (with superscript 2) correspond to the decomposed handlers when the requesting node is invalidation marked due to an INV arriving earlier than the PUT. Note that these decompositions do not change the original protocol implementation.

Table 2 lists all the transition functions of the protocol in EAGER mode and the corresponding atomic transactions of the reduced model. The atomic transactions are listed with properly instantiated parameters. The table for DELAYED mode would be the same as Table 2 except that ownership transfer (Atom-GetX-1) corresponds to the protocol step which processes the last invalidation acknowledgment.

5.5. *Invariant*

As mentioned in Section 3, we need an invariant which contains several assertions to prove the subgoals. The subgoals corresponding to precommit steps are simply proved to be valid because the specification variables are not modified at all. The theorem prover should handle them automatically (PVS does this). However, some of the other subgoals need some assertions about the system to satisfy the commutative requirement. When a subgoal cannot be proved automatically, the theorem-prover lists the proof obligations which are not satisfied. Frequently, an inadequate invariant can be improved by adding some of these proof obligations.

To check those assertions, we write an invariant which is the logical "and" of the assertions, and prove that it is preserved by every step of the protocol. In the initial state of the system, all the cache lines in the home and remote nodes are in the *invalid* state, the directory header is in the corresponding state, and the network is empty. If the invariant is not strong enough to be preserved by all the implementation steps, we need to strengthen it. Although not intellectually difficult, this was the most time-consuming part of the proof process.

The invariant we eventually derived includes the following assertions. For each memory line:

- There is at most one exclusive copy.
- There is at most one message to each node of type PUT, PUTX, WB, or SWB.
- If a node contains an exclusive copy, then there is no PUT to the home and no PUTX, WB, or SWB to any node.
- If there is a PUTX message being processed, then there is no PUT to the home and no WB or SWB, and no other PUTX to any node.
- A node is waiting for a PUT reply if there is a GET request from the node, a PUT reply to the node, or an invalidation marked.
- A node is waiting for a PUTX reply if there is a GETX request from the node, or a PUTX reply to the node.

- If Dirty in the directory header is false, there is no exclusive copy, no PUT to the home, and no PUTX, WB, or SWB to any node.
- If Pending in the directory header is false, then there is no PUT, PUTX, SWB, FWAK, GET, GETX to the home, no forwarded GET or GETX, no NAKC or INV to any node.
- The cache state in the home is in *invalid* if Local is false, or Pending is true and Dirty is false.

5.6. *Tricks for Using a Theorem Prover*

To make the computer-assisted proofs fast, we have chosen to represent the network in a nonobvious way. We observe that there is at most one request/reply message for a memory line pertaining to any particular node at any time. So the network can be represented with one variable per node per memory line (sometimes associated with the source, sometimes with the destination) for relevant kinds of messages. Hence, instead of proving that there is only one message of a certain type in the network for node i at any time, we register an error whenever a message in a variable is about to be overwritten, and verify that no error occurs. The description can read a message by accessing the variable instead of choosing a message from a set of messages, which is a bit more difficult to deal with in PVS.

5.7. *Liveness*

The aggregation method so far proves only safety properties of the protocol. The protocol described in this paper is a slightly simplified version which does not include the steps dealing with finiteness of software queues used for outgoing messages. For this simplified version of the protocol, proving liveness is not difficult. A sequence of implementation steps for each transaction are successively enabled after initiation of a transaction, because any message in the network is eventually processed assuming strong fairness.

6. Delayed Mode Conforms to Sequential Consistency Memory Model

As mentioned before, the FLASH protocol supports two memory model modes: EAGER and DELAYED. The difference between the two modes lies in when the reply is sent for a GETX request of a processor trying to write. In the EAGER mode the reply can be sent before all the invalidation acknowledgments have been collected, while the DELAYED mode only sends the reply after invalidation acknowledgments have been collected. Therefore, the EAGER mode supports a more aggressive memory model which grants exclusive ownership when there are still old copies valid for reads. This difference is visible to users and may affect the correctness of the synchronization code.

In this section we show that the DELAYED mode implements the sequential consistency memory model [23], if the processors execute instructions in a sequential order one at a time, stalling at each cache miss [14]. For the proof, we use the aggregation method again. This time, the reduced behavior of the DELAYED mode shown in Table 1 is considered the implementation instead of the specification as in the proof of Section 5, and the specification for the sequential consistency memory model is a state graph that

Table 3. DELAYED mode conforms to sequential consistency memory model.

Delayed memory model	Sequential consistency
Load_Delayed	*Load_SC* register[p][r] := memory[a]
Store_Delayed	*Store_SC* memory[a] := register[p][r]
Atomic actions of DELAYED Mode in Table 1	ε

models a collection of processors doing atomic loads and stores. The composition of two aggregation functions is an aggregation function, so this also implies the existence of an aggregation function from the full protocol to a sequential consistency memory model.

The sequential consistency memory model is specified in the right column of Table 3. The model consists of two transactions Load_SC and Store_SC which read and write data between the registers and main memory, atomically. The specification variables model the main memory and registers. The caches are now implementation variables, which are not visible to the memory model specification.

There are two interpretations for what it means for a distributed memory to be sequentially consistent [23], [7], [2]. The first is that every terminating multiprocessor program must produce one of the possible "results" (contents of memory and registers) that could be produced by a multiprocessor with a single shared atomic memory. In this interpretation, the model of Table 3 is exactly sequentially consistent.

The second interpretation is that for every program, every sequence of memory transactions produces a result that is allowed by sequential consistency. Under this interpretation, the model of Table 3 is not fully general: every sequence of memory transactions produces a sequentially consistent result, but it omits other sequences that also produce sequentially consistent results. In this case, an implementation of the table will be sequentially consistent but not fully general. We prove using aggregation that the DELAYED mode is consistent with Table 3, which implies that it is sequentially consistent under either interpretation.

In order to model registers in the implementation, we add a couple of steps to the reduced model which load and store a cached copy, respectively. The step Load_Delayed in Table 4 simulates a processor loading a memory location by reading a cached datum into a designated register if the copy is in a shared or an exclusive state. The step Store_Delayed simulates a processor storing a memory location by writing a datum into a cache line if it has an exclusive ownership of the memory line.

The commit step of the load transaction in the protocol is Load_Delayed and that of the store transaction is Store_Delayed. The aggregation function should simulate a delayed update of main memory by immediately writing back an exclusive copy, if it exists. Table 3 shows correspondence of specification steps with each step of the reduced model for the DELAYED mode. The remaining six steps correspond to idle steps.

The proof involves proofs of property (4) for eight implementation transition functions with the following invariant of the system: if a cached line is in a shared state, then main memory has the same data as in the cache and there is no exclusive copy; and there

Table 4. Reduced model of the FLASH protocol in the DELAYED mode.

	Condition	Action
Load_Delayed	cache[p][a].state = shared $\vee$ cache[p][a].state = exclusive	register[p][r] := cache[p][a].data
Store_Delayed	cache[p][a].state = exclusive	cache[p][a].data := register[p][r]
Atomic actions of DELAYED Mode in Table 1	See Table 1	See Table 1

exists at most one exclusive copy. It is easy to see that the invariant is true in the system which consists of the eight transitions of the reduced model for the DELAYED mode.

What have we really proved? The composition of the two aggregation functions, from FLASH to the reduced model to sequential consistency, may be extended inductively to sequences of steps. If a multiprocessor program is executed on FLASH, the execution will contain interleaved steps of various memory transactions. This function maps a sequence of steps on FLASH to a sequence of high-level memory transactions (and idle steps) in our model of sequential consistency. Since the aggregation function preserves the variables for processor registers unchanged between transactions, the visible result of a terminating program on FLASH is guaranteed to be the same as the result on the sequential consistency model.

7. Conclusion

We proposed a verification method for protocols and distributed algorithms. The method provides an easy and systematic way to find an *aggregation function* used for verification. The method substantially reduces the amount of labor required, so it significantly extends the capability of computer-assisted theorem-proving for distributed systems. Owing to the generality of the higher-order logic we used as a formalism, we have been able to validate protocols with an arbitrary number of processors.

The method has been successfully applied to the verification of the FLASH cache coherence protocol, which is too large and complicated to prove using a finite-state method. For several years, we had believed that proving the correctness of protocols of the complexity of the FLASH cache coherence protocol was well beyond the capability of a general-purpose theorem-prover. The aggregation method has broken through this barrier.

The aggregation method as described can be applied to many protocols, we have only tried a few. It may need to be generalized and many generalizations are conceivable: multiple commit points and reversing transactions instead of completing.

The aggregation method can be further automated to even greater advantages. We want more automation in defining an aggregation function, finding invariants of a system, and detailed proofs. From this and many other efforts, it has become clear that finding invariants is the most time-consuming part of many verification problems. More computer assistance is needed, especially for large problems.

Acknowledgments

We would like to thank Sam Owre and Natarajan Shankar at SRI International for their help with the PVS system, and Phil Gibbons and the anonymous referees for their comments and suggestions.

Appendix. Detailed Description of the FLASH Protocol (EAGER Mode)

This section contains a complete list of the handlers of the FLASH protocol in EAGER mode.

- **PI.Local.GetX:** this handler describes actions of the home when the local processor needs an exclusive copy. If Pending, the processor is NAKed. Otherwise, if Dirty, the home sends a GETX request[6] to Head_Pointer and Pending is set. Otherwise, the data in main memory is copied into the local cache (in *exclusive* state) and Local and Dirty are set. In the last case, if Head_Valid, which indicates there are shared copies in remote nodes, the home sends INVs to Head_Pointer and the nodes in Sharer_List, Pending is set, Head_Valid is reset, and the number of invalidations is written in Real_Pointers.
- **PI.Remote.Get(X):** this handler describes actions of a remote node when the processor needs a shared (or an exclusive) copy. The remote node sends a GET (or GETX) request to the home.
- **PI.Local.PutX:** this handler writes back a cached exclusive copy in the home. Dirty is reset (and Local, if not Pending) and the cached copy to the main memory is written back.
- **PI.Remote.PutX:** this handler writes back a cached exclusive copy in a remote node. The remote node sends a WB request to the home.
- **PI.Local.Replacement:** this handler replaces a shared copy in the home. Local is reset.
- **PI.Remote.Replacement:** this handler replaces a shared copy in a remote node. The remote node sends an RPL request to the home.
- **NI.NAK:** this handler describes actions of a node receiving an NAK reply. The processor clears its waiting flag and invalidation mark.
- **NI.NAKC:** this handler describes actions of the home receiving an NAKC. Pending is reset.
- **NI.Local.Get:** this handler describes actions of the home receiving a GET request from a remote node. If Pending, the home sends an NAK to the source. Otherwise, if Dirty and not Local, Pending is set and the home forwards the GET to Head_Pointer with source faked as the original requester. Otherwise, if Dirty and Local, then writes back the exclusive copy in the local cache to main memory, sends a PUT reply to the source, and Dirty is reset, Head_Valid is set, and Head_Pointer is set to the source. Otherwise, the home sends a PUT reply to the source. If Head_Valid,

[6] The original protocol uses a different request UPGRADE for an exclusive copy, rather than using GETX, when the cache has a shared copy. The reason is to enhance performance by avoiding unnecessary data transfer. However, the two requests are processed in the same manner except whether the reply contains the cached data or not. We did not model the UPGRADE request in the verified description.

List is set, Real_Pointers is incremented, the source is added to Sharer_List. If not Head_Valid, Head_Valid is set, Head_Pointer is set to the source.

- **NI.Local.GetX:** this handler describes actions of the home receiving a GETX request from a remote node. If Pending, the home sends an NAK to the source. Otherwise, if Dirty and not Local, then Pending is set and the home forwards the GETX to Head_Pointer with source faked as the original requester. Otherwise, if Dirty and Local, the home sends a PUTX reply to the source with the exclusive data from the local cache, and Local is reset, Head_Valid is set, and Head_Pointer is set to the source. Otherwise, the home sends a PUTX to the source with the data in main memory.

  In the last case, if not Dirty and Head_Valid, Dirty is set, List is reset, and if Head_Pointer is not equal to the source, Pending is set, and the home sends an INV to the Head_Pointer, and Head_Pointer is set to the source. If Local, invalidates the local copy, and if List, the home send INVs to all the nodes in Sharer_List and sets Real_Pointers to the number of invalidations. Otherwise, if not Dirty and not Head_Valid, Head_Valid and Dirty are set, Local is reset, and Head_Pointer is set to the source.
- **NI.Remote.GetX:** this handler describes actions of a remote node receiving a GETX request. If the cached data is in the *exclusive* state, it is invalidated and the node sends a PUTX reply to the source (and a forward acknowledgment FWAK to the home if the source is not the home). Otherwise, the node sends an NAK to the source and an NAKC to the home.
- **NI.Local.PutX:** this handler processes a PUT reply to the home. Local is set, Head_Valid and Pending are reset, and the exclusive copy is put into the local cache.
- **NI.Remote.Put:** The shared copy is put into the cache.
- **NI.Remote.PutX:** The exclusive copy is put into the cache.
- **NI.Inval:** Receiving an INV, the remote node invalidates the cached copy and sends an INVAK to the home. If the node was waiting for a PUT(examining its waiting flag), it marks the line invalidated.
- **NI.InvalAck:** this handler describes actions of the home receiving an INVAK. Real_Pointers is decremented. If it reaches to zero, Pending is reset (and Local if not Dirty).
- **NI.Writeback:** this handler describes actions of the home receiving a WB request. Dirty and Head_Valid are reset and the data is written back into the main memory.
- **NI.ForwardAck:** this handler describes actions of the home receiving an FWAK. Pending is reset. If Dirty, Head_Pointer is set to the source.
- **NI.Replacement**: this handler describes actions of the home receiving an RPL. The source is removed from Sharer_List if found and Real_Pointers is decremented.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[2] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Language and Systems*, 15(1):182–205, January 1993.

[3] J. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification*, 6*th International Conference*, *CAV* '94, pages 68–80, June 1994.

[4] K. M. Chandy and J. Misra. *Parallel Program Design—a Foundation*. Addison-Wesley, Reading, MA, 1988.

[5] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, March 1995.

[6] E. Cohen. Modular progress proofs of asynchronous programs. Ph.D. thesis, University of Texas at Austin, 1993.

[7] W. W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[8] J. de Bakker, W. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems*. *Models*, *Formalisms*, *Correctness*, LNCS 430. Springer-Verlag, Berlin, 1990.

[9] D. L. Dill. Tutorial at 33rd Design Automation Conference, June 1996.

[10] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *Proc*. *International Conference on Computer Design*: *VLSI in Computers*, 1992.

[11] D. Dill, A. Hu, and H. Wong-Toi. Checking for language inclusion using simulation relation. In *Computer Aided Verification*, 3*rd International Workshop*, pages 255–265, July 1991.

[12] T. Doeppner, Jr. Parallel program correctness through refinement. In *Proc*. *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 155–169, January 1977.

[13] Rob Gerth. Verifying sequentially consistent memory problem definition. Eindhoven University of Technology, April 1993.

[14] P. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proc*. 3*rd ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[15] M. Heinrich. The FLASH Protocol. Internal document, Stanford University FLASH Group, 1993.

[16] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Language and Systems*, 12(3):463–492, July 1990.

[17] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[18] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Language and Systems*, 16(2):259–303, March 1994.

[19] R. Kurshan. *Computer-Aided Verification of Coordinating Processes*: *The Automata-Theoretic Approach*. Princeton, 1994.

[20] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proc*. 21*st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[21] P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching: an assertional view. Available at http://www.research.digital.com/SRC/tla.

[22] S. Lam and A. Shankar. Protocol verification via projection. *IEEE Transactions on Software Engineering*, 10(4):325–342, July 1984.

[23] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[24] L. Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, 2:175–206, 1982.

[25] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Language and Systems*, 5(2):190–222, April 1983.

[26] L. Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4:59–68, 1990.

[27] L. Lamport and F. Schneider. Pretending atomicity. Technical Report 44, SRC Digital, 1989.

[28] R. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[29] N. Lynch. I/O automata: a model for discrete event systems. In *Proc*. 22*nd Annual Conference on Information Science and Systems*, March 1988.

[30] N. A. Lynch. *Distributed Algorithms*. Morgan Kauffman, Los Altos, CA, 1996.

[31] K. McMillan and J. Schwalbe. Formal verification of the gigamax cache-consistency protocol. In *Proc*. *International Symposium on Shared Memory Multiprocessing*, pages 242–251, 1991.

[32] K. McMillan. *Symbolic Model Checking*. Kluwer, Boston, 1993.

[33] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[34] S. Park and D. L. Dill. Protocol verification by aggregation of distributed transactions. In *Computer Aided Verification*, 8*th International Conference*, *CAV* '96, pages 300–310, July 1996.

[35] S. Park and D. L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proc*. 8*th ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, June 1996.

[36] D. Peled, S. Katz, and A. Pnueli. Specifying and proving serializability in temporal logic. In *Proc*. 6*th Annual IEEE Symposium on Logic in Computer Science*, pages 232–244, July 1991.

[37] F. Pong. Symbolic state model: a new approach for the verification of cache coherence protocols. Ph.D. thesis, University of Southern California, 1995.

[38] F. Pong and M. Dubois. The verification of cache coherence protocols. In *Proc*. 5*th ACM Symposium on Parallel Algorithms and Architectures*, pages 11–20, 1993.

[39] R. T. Simoni. Cache coherence directories for scalable multiprocessors. Ph.D. thesis, Stanford University, 1992.

[40] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods*. *IFIP WG* 10.5 *Advanced Research Working Conference*, *CHARME* 95, pages 21–34, 1995.

[41] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

Received October 1996, *and in final form August* 1997.