Protocol Verification by Aggregation of Distributed Transactions *

Seungjoon Park and David L. Dill

Computer Systems Laboratory
Stanford University
park@turnip.stanford.edu dill@cs.stanford.edu

Abstract. We present a new approach for using a theorem-prover 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 an 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.
We illustrate the method on a simple but nontrivial example. We have successfully used it for other examples, including the cache coherence protocol for the Stanford FLASH multiprocessor.

1 Introduction

Protocols for distributed systems often simulate atomic transactions in environments where atomic implementations are impossible. We believe that this observation can be exploited to make formal verification of protocols and distributed algorithms using a theorem-prover much easier than it would otherwise be. Indeed, we have used the techniques described below to verify safety properties of two significant examples: the cache coherence protocol for the FLASH multiprocessor (currently being designed at Stanford), and for a majority consensus algorithm for multiple copy databases.

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

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

work is based on using the PVS theorem-prover from SRI International [ORSvH95], the method is useful with other theorem-provers, or manual proofs.

Although finite-state methods (e.g. [McM93, DDHY92]) 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, but these techniques typically require substantial pencil-and-paper reasoning to justify. 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, 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.

If the extracted specification is not considered as a complete specification, or is not obviously 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 reduced system is based on the only state variables relevant to the specification, without variables such as local states and communications buffers.

The method described here has been successfully applied to the verification of several protocols for distributed systems including the FLASH cache coherence protocol [KOH+94, Hei93]. The FLASH cache coherence protocol, consisting of more than a hundred different kinds of implement steps, can be reduced to a specification with six kinds of atomic transactions [PD96]. It is then simple to prove interesting properties of the (much smaller) specification, such as the consistency of data at the user level.

Related work

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 [Lyn88, LS84] (indeed, whole volumes have been written on the subject [dBdRR90]). The idea has also been used with finite-state techniques [Kur94, DHWT91].

Ladkin, et al. [LLOR96] have used a refinement mapping [AL91] 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.

A more limited notion of aggregation is found in [Lam82, Lam83], where a state function undoes or completes an unfinished process. The method only aggregates se-

quential steps within a local process, while our method aggregates steps across distributed components. The idea of an aggregated transaction has been used to prove a protocol for data base systems [PKP91], where aggregation is obtained in a local process by showing the commutativity of actions from simple syntactic analysis.

Cohen used an idea similar to aggregation to prove global progress properties by combining progress properties of local processes [Coh93]. 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 [BD94].

In the next section, the basic verification procedure is presented. To illustrate it, we use a distributed list protocol which is a fragment of a distributed cache coherence protocol. In section 3, the protocol is described and we explain how to construct an aggregation function and prove that it has the necessary properties.

2 The verification method

The verification method begins with a description in higher-order logic of the state graph of the implementation of a distributed computation, and a logical description of the state graph of the specification. The implementation description contains a set of *state variables*, which are 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 relation is represented by a set of functions, $\mathcal{F} : 2^{Q \to Q}$, each of which maps a given implementation state to its next state. The implementation is nondeterministic if this set has more than one function.

The description of the specification state graph is similar. 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. We call these *idle* transitions. The idle transitions are necessary for following implementation steps that do not change specification variables. We adopt the convention that components of the specification are primed, so the set of states of the specification is Q', the set of functions is $\mathcal{F}'$, etc.

The verification method is based on the usual notion of an abstraction function. The function, which we call abs, maps implementation states to specification states and must satisfy a commutativity property

$$\forall q \in Q \quad \forall N \in \mathcal{F} \quad \exists N' \in \mathcal{F}' : abs(N(q)) = N'(abs(q)). \tag{1}$$

The most interesting part of the method is how the aggregation idea is used to define this function.

The method relies on the notion that there is a set of *transactions* which the computation is supposed to implement, which are atomic at the specification level (meaning that a transaction occurs during a single state transition in the specification), but non-atomic at the implementation level. Indeed, the transactions in the implementation may involve many steps that are executed in several different components of the implementation. Formally, the transactions in the specification are the specification transition functions.

The method requires that each transaction in the implementation have an identifiable *commit point*. Intuitively, when tracing through the steps of a transaction, the commit point is the implementation step that first causes a change in the specification variables. Implementation states that occur before the transaction or during the transaction but before the commit point are called *pre-commit states* for that transaction. The transaction is *complete* when the last specification variable change occurs as part of the transaction. The states after the commit point but before the completion of the transaction are called *post-commit 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 pre-commit states are known for each transaction. The post-commit states for the transaction are the states that are not pre-commit; the commit point for an transaction is the transition from a pre-commit state to a post-commit state for that transaction; and the completion point is the transition from a post-commit state to a pre-commit state. A state is clean if it is a pre-commit state for *every* transaction.

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.

Once a purported aggregation function has been defined, the user must prove that it meets the commutativity requirement (1). The proof consists of a sequence of standard steps, many of which are or could be automated[2]. The initial $\forall q$ and $\forall N$ can be eliminated automatically by *Skolemization*, which is substituting a new symbolic constant for q throughout (when we Skolemize in this presentation, we will not change the name of the quantified variable). This yields a subgoal of the form

$$(N \in \mathcal{F}) \Rightarrow \exists N' \in \mathcal{F}' : abs(N(q)) = N'(abs(q)). \tag{2}$$

The set of implementation steps $\mathcal{F}$ will often be defined with a logical formula of the general form $\exists \boldsymbol{p} : N = N_1(\boldsymbol{p}) \vee N = N_2(\boldsymbol{p}) \vee \ldots$, where $\boldsymbol{p}$ is a tuple of parameters (perhaps ranging over an unknown number of components), and each N_j is a different kind of implementation step. Since the $\exists \boldsymbol{p}$ is in the antecedent of an implication, it can be Skolemized automatically, and the resulting disjunction can be proved by proving a collection of subgoals

$$(N = N_j(\boldsymbol{p})) \Rightarrow \exists N' \in \mathcal{F}' : abs(N(q)) = N'(abs(q)). \tag{3}$$

The existential quantifier $\exists N'$ can be eliminated by the user by manually substituting the definition of the appropriate function for N'. Given j and $\boldsymbol{p}$, the user must supply proper instantiation j' and $\boldsymbol{p}'$ such that the resulting subgoals

$$abs(N_j(\boldsymbol{p})(q)) = N'_{j'}(\boldsymbol{p}')(abs(q)) \tag{4}$$

are provable.

The number of subgoals is equal to the number of transition functions in the implementation. In most cases, the required specification step $N'_{j'}(\boldsymbol{p}')$ is the idle step; indeed,

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

the only non-idle step is that 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 (1) 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 (1) can then be weakened to

$$\forall q \in Q \quad \forall N \in \mathcal{F} \quad \exists N' \in \mathcal{F}' : Inv(q) \Rightarrow abs(N(q)) = N'(abs(q)). \qquad (5)$$

In other words, abs 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.

3 The Distributed List Protocol

We illustrate the concepts of the previous section on a small but somewhat nontrivial example, which we call the "distributed list protocol." The protocol is an abstraction of part of a multiprocessor cache coherence protocol, which maintains a singly-linked list of processors which share a cache line.

The finite-state techniques we have applied do not scale especially well for this protocol. We have tried explicit state methods (specifically our Murφ verifier) with techniques such as symmetry reduction, reversible rule reduction [ID96], and special verification methods for parameterized families of protocols, as well as BDD-based techniques. None of these methods has allowed us to verify systems with more than about 5 list cells, because we do not have a good way of compressing or abstracting states containing linked lists. However, using the method described here, we have verified the protocol for arbitrary or even infinite numbers of list cells.

3.1 The transactions of the protocol

The protocol maintains a circular, singly-linked list of list cell processes, called *cells*. There is a special process called the *head cell* which is always in the list. Cells not in the list may request to be added to the list, and cells in the list may request to be removed. The cells communicate by sending messages over a network that is reliable, but does not preserve the sending order of messages.

Every message used in the protocol has a field *src* that contains the index of the sending cell, and a field *dst* that contains the address of the cell to which it was sent. Additional fields, *old* and *new*, are used in some message types to hold the indices of other cells.

Every cell has state variables for its control state, *state*, and the index of the next cell in the list *next*. When a cell is not in the list, its *next* variable contains the index of the cell itself. The *next* variable of each cell is a specification variable, because the list structure is important for the correctness of the protocol. The variable *state* is an implementation

variable. There are also variables associated with the cells to hold messages that are in transmission.

A cell, other than the head cell, can perform two types of transactions: *add* and *delete*. There is an add_i transaction and a $delete_i$ transaction for each cell i in the protocol (i.e., if there are n cells, there are $2n$ transactions, not 2 transactions). In the following, let i be the index of the cell initiating the transaction.

An *add* transaction can occur when cell i is not in the list, and when the state of cell i is *normal*. The cell i will be added at the head of the list. The transaction consists of three steps:

1. Cell i sends an *add* message to the head cell; cell i changes its state to *w_head* ("wait for *head* message").
2. The head cell sends a *head* message containing the *next* value of the head cell to cell i. Then the head cell stores i in its *next* variable.
3. When cell i receives the *head* message, it stores the value in the message into its *next* variable. Cell i then changes its state back to *normal*.

The specification state variables consist of the collection of *next* pointers of the cells. The *add* transaction in the specification inserts cell i at the front of the list, updating the *next* variables of the head cell and cell i in a single atomic step.

The commit step for the add_i transaction occurs in step 2, which is the first point where a specification variable is modified (*next* of the head cell). Step 1 only modifies implementation variables *state* and *network*, so it begins and ends in pre-commit states for add_i. The state between step 2 and 3 is a post-commit state. Step 3 completes the transaction; it is the point where a specification variable changes for the last time in the transaction. Hence, the state following step 3 is again a pre-commit state for add_i.

The $delete_i$ transaction can occur when a cell's *next* points to a cell other than i (meaning i is in the list) and its *state* is *normal*. The problem with deleting in a distributed singly-linked list is that there is no easy way for cell i to determine its predecessor in the list, which is unfortunate since *next* of the predecessor must be changed to point to the *next* of cell i.

The solution to this problem is to have another message *pred* which circulates around the list at all times[3]. When cell i receives the *pred* message, it can determine its predecessor by examining the *src* field of the message. So, the steps of the $delete_i$ transaction are:

1. Cell i changes its *state* to *w_pred* ("wait for *pred* message").
2. When cell i receives a *pred* message, it sends a *chnext* message ("change next") to the source of the *pred* message which is usually the predecessor of i in the list. The *chnext* message has i in its *old* field and the *next* of cell i in its *new* field. Cell i changes *state* to *w_delack* ("wait for delete-acknowledgment").
3. When a cell j receives the *chnext* message there are several possible cases. The subtleties of these rules handle difficult scenarios, such as the predecessor deleting itself and then being in the midst of adding itself again between cell i's receipt of the *pred* message and the receipt of the *chnext* message.

[3] There is another version of distributed list protocol, in which *pred* message is generated only when necessary.

(a) If the *state* of cell j is not *normal* or *w_pred*, the *chnext* message remains in the network (progress occurs when some other message arrives at cell j).
(b) Otherwise, if the *old* field of the message matches the *next* variable of cell j, the cell changes its *next* to be the *new* of the *chnext* message (*next* of i). Then cell j sends a *delack* message to cell i (*src* of the *chnext* message). Cell j then sends a *pred* message to its *next* cell.
(c) Otherwise, cell j forwards the *chnext* message to its *next* cell. In this case, the cell receiving the *chnext* message is the head cell and one or more new cells were inserted at the head of the list while cell i was being deleted, so the predecessor of cell i is now somewhere further down the list. The true predecessor will eventually receive the *chnext*, causing the case (b) to occur.

4. When cell i receives a *delack*, it changes its *next* variable to i, and changes *state* to *normal*.

The commit step of the $delete_i$ transaction is in case (b) of step 3 above. Step 3 may be repeated several times because of case (c) before a commit occurs, so a state immediately following step 3(c) is a pre-commit state. Step 4 completes the transaction.

The specification handles the delete transaction atomically by removing cell i from the list in the obvious way: it sets the *next* of the predecessor of i to the *next* of i, then sets *next* of i to i.

The *pred* message circulates around the list constantly except when it temporarily disappears during processing of a *chnext* during a delete transaction, so each cell has rules for propagating it. However, processing a *pred* message never affects a specification variable, so there are no transactions associated with it. It is necessary to reason about the processing of *pred* messages during the proof of invariants (discussed below), and also for liveness properties (which are not discussed here).

The above description of the protocol traces through individual transactions. It is easier to make sure that a description is complete if the behavior is described for each component, not each transaction (and, indeed, the above description is not complete). Table 1 gives the rules of cell behavior in pseudo-code on per-cell basis.

3.2 The aggregation function

Here, we define the aggregation function abs for the distributed list example. The key question is how to complete all committed transactions in the current state, especially since the number of cells, and hence the number of committed transactions, is unknown. The general strategy, which has worked for our larger examples as well, is to define a per-component completion function, which is then generalized to a completion function for all of the cells in the system. This is possible because the post-commit steps of different nodes are generally independent.

It is quite simple to complete a committed transaction for a particular cell. If a *head* message destined for cell i exists, an add_i transaction must be completed by simulating the effect of cell i processing the *head* message it receives at the end of the transaction. This processing changes *next* to point to the value *new* field in the message. Changes to implementation variables, such as removing messages from the network, can be omitted from the completion function, as they do not affect the corresponding specification state.

Step	Condition	Action
Initiate Add	$i \neq$headptr $\wedge$ next[i]=i $\wedge$ state[i]=normal	Send *add*⟨src=i⟩ to headptr state[i] := w_head
Process *add*	*add* sent to headptr	Send *head*⟨new=next[headptr]⟩ to *add.src* next[headptr] := *add.src*
Process *head*	*head* sent to i	next[i] := *head.new* state[i] := normal
Initiate Delete	$i \neq$headptr $\wedge$ next[i]$\neq i$ $\wedge$ state[i]=normal	state[i] := w_pred

Process *pred*	*pred* sent to i	if state[i]=normal:
		  Send *pred*⟨src=i⟩ to next[i]
		if state[i]=w_pred:
		  state[i] := w_delack,
		  Send *chnext*⟨old=i, new=next[i]⟩
		  to *pred.src*

Process *chnext*	*chnext* sent to i $\wedge$ *chnext.old*$\neq$next[i] $\wedge$ state[i]$\in${normal,w_pred}	Send *chnext* to next[i]
Process *chnext*	*chnext* sent to i $\wedge$ *chnext.old*=next[i] $\wedge$ state[i]$\in${normal,w_pred}	next[i] := *chnext.new* Send *delack* to *chnext.old* Send *pred*⟨src=i⟩ to *chnext.new*
Process *delack*	*delack* sent to i	next[i] := i, state[i] := normal

Table 1. Formal Description of Distributed List Protocol: The action of a step is executed if its condition holds. Each process consumes the message that triggers it. A message consists of a record with fields *src, new, old*. When a message is created, we use $m\langle f{=}a'\rangle$ to denote that message m has value a' for its record field f. We use $m.f$ to refer to the value of field f in message m. State variables for cells are kept in arrays, state and next.

All of this computation is done solely in cell i, without the involvement or interference of other cells. If there is a *delack* message for cell i, a $delete_i$ transaction must be completed by setting *next* to i. Otherwise, the completion function does nothing.

It is easy to generalize the completion function for one cell to a completion function for all of the cells 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.

If $cc(q[i])$ completes committed transactions on cell i, the completion function for all cells is $\lambda q.\lambda i.cc(q[i])$. When this function is supplied a state q, it returns $\lambda i.cc(q[i])$,[4] which is an array of the completed cell 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.

[4] The notation may be a bit confusing. $\lambda i.cc(q[i])$ 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 cell i_0. This is effectively the same as indexing into an array of completed cell states.

3.3 Extracting specification

Reverse engineering of a specification can be illustrated on the distributed list protocol (indeed, we had to do this). Given only an implementation description, the first step is to identify the specification variables. In the distributed list protocol, we decided that they were the *next* variables for the cells. The next step is to trace through a transaction, concatenating the implementation steps, simplifying by substituting values forward through intermediate assignments, and then eliminating statements that only change implementation variables.

For an add_i transaction in the protocol, the sequence of steps is "initiate add," "process *add*," and "process *head*." The result obtained by the procedure is

$$\begin{aligned} &\mathit{Atomic\_Add}(i)\text{: if } i \neq \text{headptr} \wedge \text{next}[i] = i \text{ then} \\ &\qquad\qquad \text{next}[i] := \text{next[headptr]; next[headptr]} := i. \end{aligned}$$

Similarly, $delete_i$ transaction corresponds to the sequence of steps, "initiate delete," "process *pred*," "process *chnext*," and "process *delack*." The atomic transaction obtained by aggregation is

$$\begin{aligned} &\mathit{Atomic\_Delete}(c, i)\text{: if } i \neq \text{headptr} \wedge \text{next}[i] \neq i \wedge \text{next}[c] = i \text{ then} \\ &\qquad\qquad \text{next}[c] := \text{next}[i]; \text{next}[i] := i. \end{aligned}$$

With the two atomic transactions and idle steps in the specification, we instantiate the subgoals (4) for each implementation steps. The proper instantiation for the proof is shown in table 2.

Implementation step at node i	Specification transactions
Initiate Add	ε
Process *add*	*Atomic_Add*(*add.src*)
Process *head*	ε
Initiate Delete	ε
Process *pred*	ε
Process *chnext* (Forward)	ε
Process *chnext* (Commit)	*Atomic_Delete*(*i*, *chnext.old*)
Process *delack*	ε

Table 2. Corresponding specification steps for implementation steps in the distributed list protocol

3.4 The invariant

The proofs of the subgoals (4) corresponding to each row in table 2 are simple. PVS can handle them almost automatically. Among the eight subgoals, four have been proved automatically for any state q. However, the rest of the subgoals need some assertions on the state in the system to satisfy the commutativity property. The invariant consisting of several assertions that we need to prove the subgoals is listed below.

- The head cell is always *normal*.
- If a cell is in *normal* or *w_pred* state, there is no *add* message from the cell, *delack* message to the cell, or *chnext* message with *old* field equal to the cell.
- If there is an *add* message from or *head* message to a cell i, then the *next* of the cell is i.
- In a *chnext* message, the *next* of the cell contained in the *old* field of the message must be the same as the *new* field of the message.
- There is at most one message in the network for each transaction currently in progress, and there must be no more than one *pred* message in the network.

The only manual step occurs when proving subgoals of the form $(\forall j : Inv(j)) \Rightarrow Q(i)$, where i is a cell index, which requires eliminating the $\forall j$ by substituting i for j to obtain $Inv(i) \Rightarrow Q(i)$, which can be handled automatically.

Part of the reason that the proof is simple is that we have chosen to represent the network in a non-obvious way. We observe that there is at most one message pertaining to any particular transaction at any time. So the network can be represented with one variable per cell (sometimes associated with the source, sometimes with the destination), plus a single variable for the *pred* message. Hence, instead of proving that there is only one message of a certain type in the network for cell i at any time, we register an error whenever a message in a variable for the network is about to be overwritten, and verify that no error occurs. The description can read a message by accessing the variable instead of choosing one and removing it from a set of messages, which is a bit more difficult to deal with in PVS. It is possible to use similar tricks in the other examples we have done, including the large FLASH protocol.

4 Concluding Remarks

Although, aggregation 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).

We have not considered the important problem of proving liveness properties here. We do not expect that it will prove to be particularly difficult, however.

From this and many other efforts, it has become clear that finding invariants 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 PVS system.

References

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[BD94] Jerry Burch and David Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification, 6th International Conference, CAV'94*, pages 68–80, June 1994.

[Coh93] Ernest Cohen. *Modular progress proofs of asynchronous programs*. PhD thesis, University of Texas at Austin, 1993.

[dBdRR90] J. de Bakker, W. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems. Models, Formalisms, Correctness.: LNCS 430*. Springer-Verlag, 1990.

[DDHY92] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design: VLSI in Computers*. IEEE Computer Society, 1992.

[DHWT91] D. Dill, A. Hu, and H. Wong-Toi. Checking for language inclusion using simulation relation. In *Computer Aided Verification, 3rd International Workshop*, pages 255–265, July 1991.

[Hei93] Mark Heinrich. *The FLASH Protocol*. Internal document, Stanford University FLASH Group, 1993.

[ID96] C. Norris Ip and David Dill. State reduction using reversible rules. In *Proceedings of 33rd Design Automation Conference*, June 1996.

[KOH+94] 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. 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[Kur94] Robert Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton, 1994.

[Lam82] Leslie Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, 2:175–206, 1982.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Language and Systems*, 5(2):190–222, April 1983.

[LLOR96] P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching: An assertional view. *Distributed Computing*, 1996. To appear.

[LS84] S. Lam and A. Shankar. Protocol verification via projection. *IEEE Transactions on Software Engineering*, 10(4):325–342, July 1984.

[Lyn88] N. Lynch. I/O automata: A model for discrete event systems. In *22nd Annual Conference on Information Science and Systems*, March 1988. Princeton University.

[McM93] Ken McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. Boston.

[ORSvH95] 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.

[PD96] Seungjoon Park and David Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proc. 8th ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[PKP91] D. Peled, S. Katz, and A. Pnueli. Specifying and proving serializability in temporal logic. In *Proc. 6th Annual IEEE Symposium on Logic in Computer Science*, pages 232–244, July 1991.