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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.