Automatic Checking of Aggregation Abstractions Through State Enumeration

Seungjoon Park, Member, IEEE, Satyaki Das, and David L. Dill, Member, IEEE

Abstract—Aggregation abstraction is a way of defining a desired correspondence between an implementation of a transaction-oriented protocol and a much simpler idealized version of the same protocol. This relationship can be formally verified to prove the correctness of the implementation.

We present a technique for checking aggregation abstractions automatically using a finite-state enumerator. The abstraction relation between implementation and specification is checked on-the-fly and the verification requires examining no more states than checking a simple invariant property. This technique can be used alone for verification of finite-state protocols, or as preparation for a more general aggregation proof using a general-purpose theorem-prover. We illustrate the technique on the cache coherence protocol used in the FLASH multiprocessor system.

Index Terms—Abstraction, automatic verification, computer-aided verification, formal methods, model checking, multiprocessor cache coherence protocols.

I. INTRODUCTION

FORMAL verification of a system design compares two different descriptions of the system: the specification describes the desired behavior, and the implementation describes the actual behavior of the system. The implementation is usually given in some (potentially) executable form. There are many specification methods, such as assertions in the implementation code, temporal logic or other logical properties, or automata.

The most appropriate specification for a behavior is often an abstract version of the behavior with coarser-grained atomicity. For example, most cache coherence protocols are intended to appear to the user as though they were executing simple atomic read, write, and other transactions to a shared memory. For efficiency, the implementation is much more complex than this: It implements each memory transaction in a sequence of steps that are executed at different points in the distributed system. As another example, a network communications protocol presents the illusion to a user that a message is transmitted reliably from a sender to a receiver; however, the protocol is more complex, involving checksums, sequence numbers, retries, etc., because it is dealing with an unreliable transport medium. When the implementation and specification are both protocols, verification consists of comparing the two protocols according to some consistency criterion.

In this paper, we propose a technique that checks aggregation abstractions for finite-state systems automatically using a finite-state enumerator. Previously, we developed a proof methodology called “aggregation” for relating a protocol to its abstract version by providing an abstraction function which reassembles individual implementation steps into atomic transactions in a specification protocol [1], [2]. The new method complements the previous method by providing an easy, automatic way to check a finite instance of a description, while the previous method allows proofs of the general case where there are arbitrary numbers of processes.

The aggregation method is applicable when the implementation attempts to simulate each atomic transaction in the specification by one or more sequences of smaller steps for each transaction. Each sequence of steps implementing a transaction must have a particular commit step, which is, in general, the first step where the transaction does something irrevocable. The user provides an aggregation function which maps an implementation state to a specification state by completing any committed but incomplete transactions. Each step of the implementation is assumed to be involved in exactly one transaction. The user must indicate the transaction corresponding to each step. Given a correct correspondence, an aggregation function, and an inductive invariant, a theorem prover can finish the proofs automatically, or semiautomatically depending on its level of automation.

Although the aggregation method makes verification using a theorem-prover much easier than it would otherwise be, use of theorem provers is still more labor-intensive than using algorithmic verification such as finite-state methods, especially when protocols are incorrect. In particular, finite-state methods automate the most difficult part of many verification efforts: the derivation of an adequate inductive invariant.

This paper proposes a technique that checks aggregation abstractions for finite-state systems automatically using a finite-state enumerator. We reduce the problem of checking the aggregation correspondence to the simpler problem of checking an (noninductive) invariant (an “AG property” for those familiar with computation tree logic ((CTL) [3]) by generating a set of propositional properties from the correspondence requirements of the aggregation method. This method can be used alone for verification of finite-state distributed protocols, or to debug aggregation abstractions before theorem proving. Using the finite-state method can greatly reduce the amount of human effort required to complete a proof. Of course, there is a tradeoff: finite-state methods only work for small instances of a system design (e.g., a multiprocessor with three processors and one bit.
of memory), while a theorem prover can show the correctness of all instances of the system, for any number of processors and arbitrarily large memory.

Our technique is practical and much less expensive than other finite-state methods for proving abstraction relations between implementation and specification (for comparison, see Section V). The number of states searched during verification using our technique is the same as checking propositional safety properties on a finite-state implementation of the system. When the aggregation correspondence holds, the method generates only the reachable states of the implementation. When the aggregation does not hold, state generation ceases when the first violating state is detected. The aggregation method is more efficient because the abstraction is between transition rules of source-level descriptions, not between state transitions in state graphs. While it may require more human effort to define the aggregation functions than, say, finding a simulation relation automatically, this is an appropriate tradeoff when the state explosion problem obstructs a more automatic proof.

The method is compatible with theorem proving because the same description of the specification protocol, the implementation protocol, and the aggregation function can be used for both automatic checking and theorem proving of the aggregation abstraction. It allows the user to debug the implementation, specification, and aggregation functions quickly before invoking the theorem prover for proving the correctness of an unbounded implementation or infinite family of implementations. The state enumerators can also be used to help debug invariants and check some lemmas on examples before trying to prove them formally. Obviously, the same general technique can be used with any program capable of enumerating the reachable states of a system description, including binary decision diagram (BDD)-based model checkers [5]. Even using a BDD-based model checker, aggregation may be more efficient than other verification for abstractions.

II. AGGREGATION ABSTRACTION

This section describes the aggregation method, in general, using a simple illustrative example: distributed cache memory. Formal presentation of the aggregation method with its verification goal can be found in [1]. First, we demonstrate the difficulty in applying a simple abstraction function to verify the protocol. Then we explain how to find a correct abstraction function by reasoning on aggregation.

A. Abstractions and Partially Completed Transactions

There are many approaches to abstraction. Since the details of these methods vary greatly, it is difficult to find a simple general principle underlying them all. However, at a very high level, many of them can be seen to involve proving a property that an abstraction function $F$, from an implementation state to a specification state, satisfies (as shown in Fig. 1)

$$\forall s_i, s'_i; \text{Impl}(s_i, s'_i) \Rightarrow \text{Spec}(F(s_i), F(s'_i)) \quad (1)$$

where $\text{Impl}$ is a relation representing the possible implementation steps, and $\text{Spec}$ is a relation representing specification steps.

We illustrate this approach on a simple example. The implementation of the system, shown in Fig. 2, consists of a main memory and a collection of write-back (WB) caches which contain copies of the data in main memory. The caches communicate with the memory via a network which delivers messages after an arbitrary delay. A cache copy is dirty if it has been updated and the main memory copy has not. One of the transactions in the memory system, called a write back, updates the data in the main memory with the dirty value from the cache.

In our example, the write back transaction is implemented in two steps. First, the cache inserts into the network a message containing the value of the dirty copy, and then the cache deletes its copy. Second, the main memory updates its value using the value from the message in the network, and then deletes the message.

The specification, which is an abstract version of the protocol, is similar to Fig. 2, except that there is no network (the caches are connected directly with the main memory). Variables, such as the network, that appear in the implementation but not in the specification are called implementation variables. In the specification, the write back transaction is performed in a single atomic step, copying the value from the cache to the main memory and then deleting the cache copy. The state transitions in the specification and the implementation for performing the write back transaction are shown in Fig. 3.

In this example, the network is empty if and only if all transactions that have been started have also been completed. In such a state, the abstraction function is very simple: it deletes implementation variables associated with the network and preserves the contents of the cache and main memory. Given this observation, the simplest abstraction function imaginable would be the one that does this for all states, whether the network is empty or not. Call this function $F$.

$F$ fails to be an abstraction function because it violates property (1) in some states. Suppose the network is empty in implementation state $s_i$ and one of the caches in $s_i$ has a dirty copy that is different from the one in main memory. Then, $F(s_i)$ has the same cache and main memory value as $s_i$ does. Suppose $s'_i$.
is a state where the implementation has done the first step of the write back transaction, so $Impl(s_i, s'_i)$ holds. Then $s'_i$ and $F(s'_i)$ have no copy of the dirty value in their cache, and the main memory value is the same as the value in $s_i$ and $F(s_i)$. But, there is no specification step that takes $F(s_i)$ to $F(s'_i)$ (i.e., $Spec(F(s_i), F(s'_i))$ does not hold), since the specification steps always update the main memory when they delete the cache copy.

The previous paragraph illustrates the primary problem with defining abstraction functions in the presence of partially completed transactions. Different parts of the state are updated at different steps of the transaction, so the implementation variables do not correspond directly to the specification variables. Defining the abstraction function so that property (1) holds in all states is made especially difficult because there may be many different transactions of different types in varying stages of completion.

The main idea in aggregation is to start with a simple abstraction function for clean states, where there are no partially completed transactions, and extend it to all states by completing all committed transactions, then applying the abstraction as though the state were clean.

B. The Aggregation Method

The aggregation method provides a systematic way to find an abstraction function for transaction-oriented protocols. To apply the aggregation method, the user should first identify the transactions implemented in the protocol. In our example, there is a write back transaction and some other transactions that were not discussed. Next, the user needs to identify a commit step in each transaction in the protocol. In general, the commit step is the first step where the transaction does something irrevocable. In the example, once the message is sent out, the cache no longer has control of the dirty copy so the first step is the commit step of the transaction.

The user then defines an aggregation function $aggr$ which maps an implementation state to a specification state by first completing any committed but incomplete transactions, then completing all completed transactions, and extend it to all states by completing all committed transactions, then applying the abstraction as though the state were clean.

C. Invariants

The requirements (2) will generally not hold for some absurd states that cannot actually occur during a computation. Hence, when using a theorem-prover, it is usually necessary to provide an inductive invariant predicate which characterizes a superset of all the reachable states. If our example were complete, it might be necessary to prove an invariant that, at any time, there is at most one write back message; otherwise, the aggregation function could result in different main memory contents depending on the order in which it processed write back messages from the network. Other invariants might also be necessary.

If the invariant is $Inv$, the requirement can then be weakened to

$$\forall q \in Q: \text{aggr}(\text{Impl}_i(q)) = \text{Spec}_j(\text{aggr}(q)).$$

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 find an invariant that satisfies (3), and second, we must prove inductively that the invariant is true in the implementation description by showing that the invariant holds at initial states and each implementation step preserves the invariant. From our experience, finding and proving an inductive invariant is the most time consuming part of many verification problems. It is especially difficult to debug faulty invariants using only a theorem prover. One of the great advantages of finite-state verification methods is that they compute an invariant, namely the reachable state space, automatically.

D. Liveness

We have limited our attention to safety properties in this paper, so failure to complete transactions is beyond our scope of concerns. However, there are a number of model checkers which are able to verify properties specified in temporal logics, and we do not expect any difficulties in proving liveness properties, such as that any initiated transactions in the finite-state protocols are eventually completed.
To check the aggregation abstraction automatically, we use a finite-state enumerator which explores all and only the reachable states of the implementation on-the-fly. Because state enumeration generates the exact invariant of the system while searching the state space, the user can check property (2) above without proving property (3).

Given a purported aggregation function and correspondence between implementation steps and specification steps, the requirements are expressed as a predicate which is a Boolean expression in Mur$\varphi$ on the implementation state. The predicate consists of a set of conjuncts corresponding to each implementation step (for convenience, we assumed $Spec_i$ is the corresponding specification step of $Impl_i$)

$$\bigwedge_i aggr(Impl_i(q)) = Spec_i(aagr(q)).$$

(4)

The aggregation abstraction holds if the predicate is true on all the reachable states in the implementation. Therefore, the aggregation abstraction can be automatically checked using any finite-state enumerator which is able to check such propositional properties. Although we used the Mur$\varphi$ verifier [6] for this purpose, the technique could be used with other model checkers, including model checkers based on BDDs [5] or other symbolic representations.

### A. Description Language and Verifier System

Mur$\varphi$ is a high-level description language [7] for modeling finite-state asynchronous concurrent systems. The description allows the declaration of familiar data types, including subranges of integers, arrays, records, and user-defined enumerations. Additionally, procedures and functions can be declared. A Mur$\varphi$ program is an implicit description of a state graph. The program consists of a collection of state variables and transition rules. The states of the graph are assignments to each global state variable with a value in the range of the declared type. The transition rules transform states to states by assigning to the state variables, so they define the edges of the state graph. Each rule has an enabling condition, which is a Boolean expression on the state variables, and an action, which is a statement that modifies the values of the state variables, generating a new state. Each rule is of the form

**Rule condition $\iff$ action statement Endrule.**

The action statement is an arbitrarily complex statement in a fairly conventional programming language with assignment, if-then-else, loops, procedure calls, and local variables.

Mur$\varphi$ has an automatic verifier which generates all of the reachable states of the described system. Execution of a Mur$\varphi$ program begins with one of a set of initial states of the graph. Then the following loop is executed forever: some rule whose condition is satisfied by the current state is chosen and its action evaluated, yielding a new current state. If there are no rules whose conditions are true, the execution halts. Although the action may be a compound statement consisting of a sequence of smaller statements, conditionals, and loops, it is executed atomically—no other rule can be executed before the action completes. When several rule conditions are true at the same time, a choice is made arbitrarily, resulting in several possible executions. The Mur$\varphi$ verifier exercises the rules exhaustively by depth-first or breadth-first search.

Several types of errors can be detected while the verifier explores the state graph. An invariant which is a Boolean expression on the global state variables is checked in any reachable state. An assert statement, which is a Boolean condition specified in an action statement, can also be checked whenever the verifier gets to the specified point to execute the description. The system can detect deadlock states, which are states that have no other states as successors. If a problem of any type is detected, the verifier prints out a diagnostic trace, which is a sequence of states that leads to a state exhibiting the problem.

### B. Checking Aggregation Abstractions using Mur$\varphi$

Normally, a description of a single protocol implementation is specified in Mur$\varphi$, and simple properties such as invariants and in-line assertions are checked. For aggregation, we need also to include a specification protocol and an aggregation function. First, we embed all of the implementation state variables in a single record type; similarly, we embed the specification protocol state variables in a second record type. The aggregation function $aggr$ is written as a function in Mur$\varphi$. The specification steps are also written as functions, which take a specification state (record) as an argument and which return a modified specification state.

A straightforward way of checking the predicate (4) is using an invariant of Mur$\varphi$. We specify the propositional predicate (which may be a big Boolean expression) as a single invariant and then run the verifier to check it on every reachable state. Similarly, this can be done using other model checkers: e.g., a CTL model checker by specifying the same condition as an AG property.

However, the requirements can be checked more efficiently if we exploit the property that each requirement (2) for an implementation step matters only when the step is enabled and that not all the implementation steps are enabled from a state. Requirement (2) can be written as a conjunction of properties, each of which deals with a particular implementation step. Using the in-line assert commands of Mur$\varphi$, each conjunct of the predicate can be checked separately on-the-fly only when the corresponding step is enabled and generates a next state by executing its action statement. To this end, we add an assert statement to each rule of the implementation description as shown in the following (the original rule was of the form **RULE CONDITION $\iff$ Begin ACTION-STATEMENT; Endrule;**):

**Rule "Transition relation for Impl_i"**

**CONDITION**

$$\iff$$ \begin{verbatim}
Var i0, i1: ImplState;
Begin
  i0 := current_state;
  ACTION-STATEMENT;
  i1 := current_state;
  Assert Spec_i (aggr(i0)) = aggr(i1);
Endrule;
\end{verbatim}
The two local variables \(\tilde{z}_0\) and \(\tilde{z}_2\) contain the implementation state before and after the execution of the rule, respectively. The assert statement expresses the corresponding conjunct of the predicate using the functions defined for specification steps and the aggregation function.

The Mur\(\varphi\) verifier will automatically check the assertions on-the-fly on all the reachable states while exploring the state space of the implementation description. Note that the specification steps are written as functions and called and computed in local variables inside the assert statements, while the implementation steps are written as statements which are executed to generate next states in the description. The specification state is not saved between rule executions—only the implementation contributes to the states. Therefore, the number of states explored by the verifier is still the same as that of the implementation description. Consequently, the amount of memory needed to check the abstraction is the same as that needed to check the reachable states of the implementation alone.

IV. EXAMPLE: FLASH CACHE COHERENCE PROTOCOL

This section illustrates our technique on the cache coherence protocol used in the Stanford Flash multiprocessor system [8]–[10].

A. Informal Description of the Protocol

The system consists of a set of nodes, each of which contains a processor, caches, and a portion of global memory of the system. For a memory line, the node on which that piece of memory is physically located is called home; the other nodes are called remote (Fig. 4). The distributed nodes communicate using asynchronous messages through a point-to-point network. The state of a cached copy is either invalid, shared (readable), or exclusive (readable and writable). The cache coherence protocol is directory-based: each cache line-sized block in main memory (called a memory line) is associated with a directory header. The home maintains all the information about memory lines in its main memory in the corresponding directory headers. The protocol is too complex to describe completely, but let us consider some of the more difficult cases.

If a read miss occurs in a node, say \(R_3\), the node sends out a get request to the home \(H\) for the memory line (if \(R_3\) is the home of the memory line, this step is handled locally without involving the protocol). When \(H\) receives the GET request [GET (1) in Fig. 5(a)], it consults the directory corresponding to the memory line to decide what action should be taken. If the directory header has its pending bit set, meaning that another request is already being processed, \(H\) sends a NAK (negative acknowledgement) to \(R_3\).

Otherwise, in the simple case where there are only shared copies, \(H\) grants the request by sending a PUT to \(R_3\) and updates the directory properly. When \(R_3\) receives a PUT reply, which contains the requested memory line, the processor sets its cache state to shared and proceeds to read.

If the directory indicates there is an exclusive copy of the memory line held by another remote node \(R_2\), then \(H\) forward the GET to \(R_2\) with the source of the message faked as \(R_4\) instead of \(H\) [GET (2) in Fig. 5(a)]. When \(R_2\) receives the forwarded GET, the processor sets its copy to shared and sends a PUT to \(R_4\) [PUT (3) in Fig. 5(b)]. At this time, the node \(R_2\) also sends a sharing write-back (SHWB) containing the exclusive data to \(H\) with the source faked as \(R_5\) instead of \(R_2\) [SHWB (3) in Fig. 5(b)]. When \(H\) receives this message, it writes the data back to main memory and puts \(R_4\) on the sharer list in the directory header [Fig. 5(c)]. Another view of the transaction is shown in Fig. 6.

For a write miss in a node \(R_3\), the requesting node sends a GETX to the home, \(H\), for the memory line. When \(H\) receives the GETX request, it consults the directory. If the line is \{it pending\}, \(H\) sends a NAK to \(R_3\). If the directory indicates there is an exclusive copy in a third node \(R_2\), then \(H\) forward the GETX to \(R_2\). If the directory indicates there are shared copies of the memory line in other nodes, \(H\) sends invalidations (INVs) to those nodes. Then \(H\) grants the request by sending a PUTX to \(R_4\). If there are no shared copies, \(H\) simply sends a PUTX to \(R_3\) and updates the directory to have \(R_4\) as the owner. When the requesting node receives a PUTX reply which contains an exclusive copy of the requested memory line, the processor sets its cache state to exclusive and proceeds to write.

2This is the case when the multiprocessor is running in EAGER mode. In DELAYED mode, this grant is deferred until all the invalidation acknowledgments are received by the home.
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 (for a previous read miss). 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 sending a replacement hint to the home, which removes the remote from its sharers list. An exclusive copy is written back to main memory by sending a WB request to the home. When the home receives the WB, it updates the line in main memory and the directory properly.

B. The Aggregation Function

To define the aggregation function \( \text{aggr} \), we first identify commit steps of each transaction in the protocol. For a transaction processing a read miss (or a write miss), the commit step occurs when the home or a remote node with an exclusive copy sends a PUT (or PUTX) reply granting the request; once this step takes place, any further requests will be processed by another node as if the ownership has already been transferred. A WB transaction begins with invalidating an exclusive copy and sending a WB request to the home. This is the commit step of the transaction because a part of the specification variables are already updated at this moment and the WB request cannot be denied by the home.

The aggregation function simulates completing all committed transactions in the current state. If there exists a PUT message destined to a node \( i \), the transaction for a read miss in node \( i \) must be completed by simulating the effect of node \( i \) processing the PUT message: Node \( i \) updates its cache with the data from the message and sets the state of the line to \( \text{shared} \). The transaction for a write miss is similarly completed by processing a PUTX message. There are two more kinds of messages possibly generated at commit steps that need to be processed to complete the committed transactions: SHWB and WB to the home. 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.

The aggregation function processing all the messages as described can be easily written in Murphi using a “for-loop” indexed on the network queue. Fig. 7 shows the definition of the function. An implementation state is declared as a record consisting of an array \( \text{PNet} \) for network containing reply messages, an array \( \text{QNet} \) for network containing request messages, and an array \( \text{Procs} \) modeling processors with caches. Each message is also a record containing fields for its destination \( \text{dst} \), source \( \text{src} \), and data \( \text{Data} \). From an implementation state \( \text{ist} \), the function computes a specification state using a local variable \( \text{sst} \) to be returned. First, the specification variables of \( \text{ist} \) are copied into \( \text{sst} \). Then, in the second for-loop, \( \text{sst} \) is updated by simulating the effect of processing each pending message in the network, if that message was sent after the commit step of the transaction.

C. Checking the Aggregation Abstraction using Murphi

We illustrate the details of checking the aggregation abstraction on one of the implementation steps (i.e., one of the require-
Fig. 8. The specification step to process a write miss in the FLASH protocol.

Fig. 9. A sample Murφ rule with an assert statement for checking the commutativity requirement. The lines marked with a star are additions to the original implementation description.

The abstraction has been checked automatically using Parallel Murφ on 32 ULTRA SPARC processors [11]. For the protocol with three processing nodes and request/reply message queues of size 5, the verifier explored 457 558 states in 126 s; for four processing nodes and queues of size 3, about 19 million states in 72 min. As expected, the number of states explored (also the usage of memory) is exactly the same as that found for exploring reachable state space of the implementation model.

V. BACKGROUND AND RELATED WORK

The use of abstraction functions and relations of various kinds (also called refinements [12], [13], homomorphisms [14], and simulations [15]) to compare two descriptions is a fundamental verification technique that can be applied to many different problems and representations in many different ways (e.g., [16]–[18]).

One approach which combines a notion of abstraction with finite-state methods is to prove that a simulation preorder holds between the implementation and specification state graphs. For simulation preorder checking, the user provides descriptions of the implementation and specification state graphs and an initial relation which must contain the desired simulation relation. For example, the initial relation could require that whenever an implementation state is paired with a specification state, the two states have the same label. Given this information, the existence

--- Parameterized rule for each message in the queue

Ruleset i : Queue Do

- Choose an arbitrary request in the queue,
- the destination of the request, and
- the node that initiated the request

Alias request: QNet.Message[i];

dst : request.dst;

SRC : request.SRC Do

Rule "NI Remote GetX (Commit)"

TopRequestTo{i}:

- The message is the oldest among those src -> dst
- & request.Mtype = GetX
- The message is a 'getx' request
- & request.dst != Home
- The receiving node is a remote node
- & Proc[dst].Cache.State = Exclusive
- The node dst has an exclusive copy
& Pspace(i) & Qspace(i)
- There are enough spaces in the network

=>

* Var i0,i1: ImplState; -- Murphi local variables

s0,s1: SpecState; -- Murphi local variables

Begin

- i0 := ThisImplState(); -- copy the current state
- s0 := Faggr(i0);
- Cache.State := Invalid;
- Send.Reply(dst, SRC, PutX, Cache.Value);
- if SRC != Home then
  Send_Request(dst, Home, Fack, SRC, void); end;
- Consume_Request(i1);
- i1 := ThisImplState(); -- copy the current state
- s1 := Faggr(i1);
- s0 := Atomic_GetX(s0, i0,QNet.Message[i].dst,

- i0,QNet.Message[i].SRC);
- assert Eqv(s0,s1) "NI Remote GetX (Commit)"

Endrule;

Endalias;

Endruleset;

D. Experiments

By the aggregation abstraction, the protocol consisting of more than 100 different implementation steps has been reduced to a specification with only six kinds of atomic transactions. It is much easier to prove important properties of the reduced model, such as the consistency of data at the user level, than the original protocol description.

The abstraction has been checked automatically using Parallel Murφ on 32 ULTRA SPARC processors [11]. For the protocol with three processing nodes and request/reply message queues of size 5, the verifier explored 457 558 states in 126 s; for four processing nodes and queues of size 3, about 19 million states in 72 min. As expected, the number of states explored (also the usage of memory) is exactly the same as that found for exploring reachable state space of the implementation model.
of a simulation relation (and the most general relation) can be computed by iterative elimination of states that cannot satisfy property (1).

Simulation preorder checking can be computed “on-the-fly” if the state graphs are given implicitly as a set of rules or finite-state programs [19], [20]. Unfortunately, in the worst-case, this computation is linear in the size of the product of the implementation and specification graphs. In both theory and practice, checking simulation preorder between implementation and specification graphs is much more expensive than our technique, which costs the same as checking a simple safety property on the implementation graph alone.

Simulation preorder checking can also be performed on graphs represented by BDDs by using a symbolic fixed-point algorithm [21]. However, this requires dealing with relations containing Boolean variables of both the implementation and specification state graphs. Although BDD size is not necessarily determined by the number of variables in a description, the cost of BDD-based checking of a simulation preorder is typically much larger than the cost of checking a simple property on an implementation state graph alone.

Another approach using abstraction with BDDs is found in [22]–[24]. The method claims that a concrete program satisfies a property specified with CTL formulas if the abstracted program satisfies the corresponding property by an abstraction relation. To apply this method, the user must find an abstraction relation that preserves the property given in CTL formulas. There are two problems with this method from our perspective. First, we are interested in using a protocol as the specification. It is difficult or impossible to specify a protocol completely using CTL. Second, this method uses BDDs or some similar symbolic representation. Yet, we have found that explicit state enumeration greatly outperforms straightforward BDD-based verification for some classes of descriptions [25], such as the directory-based cache coherence protocols above.

A recent approach to verifying the “bounded retransmission protocol” is similar to ours [26]. It compares an implementation and specification protocol by augmenting a Murϕ description of the implementation with specification state variables, actions that update the specification variables during implementation steps, and assertions to check the consistency of the implementation and specification states. However, this work does not address our main problem, which is how to deal with partially completed transactions in abstraction functions. Moreover, if the implementation and specification do not correspond, their method may explore the product of the implementation and specification states, while our method is guaranteed never to explore more than the implementation states.

Finite-state methods (e.g., model checking) are widely used for validating cache coherence protocols [27]–[30]. Some of them use abstract state representation to reduce the state explosion [31], [32]. However, the properties checked are usually specified in CTL or as invariants. An example of such a property might be: “there is only one dirty copy of any cache line.” None of this work addresses the problem of comparing with an abstract version of the same protocol. As we pointed out in Section I, it is often much easier to give a complete specification in the form of an abstract protocol. In fact, none of this other work claimed to provide a complete specification.

VI. CONCLUSION

By limiting the verification problems to finite-state systems, we proposed an efficient technique for checking aggregation abstractions that does not require users to find inductive invariants. The verification requires checking only the same number of states as in the implementation model, by exploiting the correspondence information provided by the user.

The technique can be used alone for verification of finite-state protocols, since abstract protocols are sometimes the best properties to check. The technique can also be applied before theorem proving of aggregation abstractions to debug a purported aggregation function in early stage.

The FLASH protocol example has been verified before by applying aggregation abstraction using a general-purpose theorem-prover, which took two months [33]. However, the proofs would have been much easier had we thought of this finite-state method before completing them. Use of the automatic checking can reveal any human errors in finding aggregation functions and specification models, and helps to debug them in early stage of proofs.

There remains some work to be done. The examples to which we have applied the aggregation method so far are mainly cache coherence protocols in distributed shared memory or similar applications. We are currently working on generalizing the method to apply to different kinds of problems like communications protocols.

The aggregation abstraction is for checking safety properties but is not able to deal with liveness properties of protocols. We have proved liveness of some examples, but a general theory for proving liveness would be an important contribution.

ACKNOWLEDGMENT

The authors would like to thank U. Stern for helping run Parallel Murϕ and D. Culler at Berkeley for letting them use their NOW.

REFERENCES


