13

Automatic Checking of Aggregation Abstractions Through State Enumeration

Seungjoon Park *Satyaki Das* *David L. Dill*
Computer Systems Laboratory, Stanford University
Gates {358, 312, 314}, Stanford University, Ca 94305, U. S. A.
{park@turnip, satyaki@turnip, dill@cs}.stanford.edu

Abstract

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 in the FLASH multiprocessor system.

Keywords

Automatic verification, cache coherence protocols, distributed systems, aggregation abstraction, formal methods

1 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 the other logical properties, or automata. However, the most appropriate specification for a protocol is often an abstract version of the protocol with coarser-grained atomicity. For example, most cache coherence protocols are intended to simulate atomic memory operations using non-atomic sequences of steps which execute in a distributed environment. Verification of such a protocol ultimately requires comparing the implementation protocol with the specification protocol with respect to some consistency criterion.

Previously, we developed a proof methodology called "aggregation" for relating a protocol to its abstract version by providing an abstraction function

Formal Description Techniques and Protocol Specification, Testing and Verification
T. Mizuno, N. Shiratori, T. Higashino & A. Togashi (Eds.)

which reassembles individual implementation steps into atomic transactions in a specification protocol [25, 24]. 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.

The aggregation method is applicable when the description attempts to simulate a set of atomic *transactions*, where each transaction has a *commit step*. The user provides an *aggregation function* which maps an implementation state to a specification state by completing any committed but incomplete transactions, and correspondence between implementation steps and specification steps. Given correct correspondence, an aggregation function, and a proper invariant, a theorem prover can finish the proofs automatically (or semi-automatically 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 invariant.

In this paper, we propose 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 invariant (an "AG property" for those familiar with 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 abstract relations between implementation and specification (for comparison, see the section on related work). The number of states searched during verification using our technique is same as checking propositional safety properties on a finite-state 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, find-

ing a simulation relation automatically, this may be 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 enumerator 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 BDD-based model checkers* [22]. Indeed, it may outperform other methods using abstraction in BDD-based model checkers, for some applications.

Background and related work

The use of abstraction functions and relations of various kinds (also called refinements [1, 19], homomorphisms [16], and simulations [23]) 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. [21, 18, 6]).

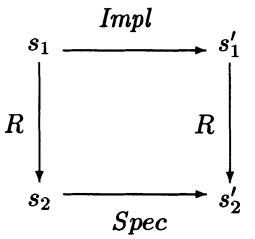

Figure 1 Abstraction relation

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 something like (as shown in Figure 1):

$$\forall s_1, s'_1, s_2 : \exists s'_2 : R(s_1, s_2) \wedge \mathit{Impl}(s_1, s'_1) \Rightarrow R(s'_1, s'_2) \wedge \mathit{Spec}(s_2, s'_2), \qquad (1)$$

*BDD-based model checkers use a binary decision diagram [2] to represent a Boolean function that describes a set of states symbolically.

where *Impl* is a step of the implementation, *Spec* is a corresponding step of the specification, and R is a binary relation from an implementation domain to a specification domain.

One approach using 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 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 [4, 5, 15, 14]. 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 [7]. However, this requires dealing with relations containing Boolean variables of both the implementation and specification state graphs, so the cost of verification using this method is also much greater than that of checking a simple property on an implementation state graph alone.

Another approach using abstraction with BDDs is found in [20, 10, 9]. 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 [13], such as all those described below.

A direct approach would appear to checking inclusion of the language of a finite automaton describing implementation behavior in the language of another automaton describing specification behavior, possibly using an on-the-fly algorithm. If the specification automaton is nondeterministic, this operation is generally exponential in the size of the specification automaton (some individuals have finessed this problem by requiring the specification automaton to be presented in a complemented form [16]). If the specification automaton

is deterministic, the algorithm for inclusion checking is basically identical to simulation preorder checking.

Recently, there has been proposed an approach to using a model checker for comparing a specification protocol and an implementation protocol [11]. However, the technique uses a model checker simply to run the two protocols in parallel without defining a precise abstraction relation between the two protocols. Moreover, the size of each state checked by the technique is increased by that of a specification protocol.

2 THE AGGREGATION ABSTRACTION

This section describes the aggregation method in general. The verification method begins with logical descriptions of state graphs of the implementation and the specification. The implementation description contains a set of state variables; the set Q of states of the implementation is the set of assignments of values to the state variables. The specification description may contain a subset of the state variables of the implementation. Each description also specifies a transition relation between a state and its possible successors represented by a set of functions. An implementation step $Impl_i$ maps a given implementation state to its next state. Similarly, a specification step $Spec_i$ maps a given specification state to its next state. The specification contains *idle* transitions which map a state to itself.

The aggregation abstraction works when the computation can be thought of as implementing a set of transactions and each transaction has an identifiable commit step. Based on the reasoning about the commit steps, the user defines an aggregation function *aggr* which maps an implementation state to a specification state by first completing any committed but incomplete transactions, then hiding variables that do not appear in the specification. The commit steps in the implementation correspond to atomic transactions in the specification, and the other steps in the implementation correspond to an idle transition in the specification. For each pair of corresponding implementation step and specification step (for convenience, we assumed $Impl_i$ corresponds to $Spec_i$), the aggregation function should satisfy the following commutativity requirement,

$$\forall q \in Q : aggr(Impl_i(q)) = Spec_i(aggr(q)). \tag{2}$$

The number of proofs required is equal to the number of transition functions in the implementation.

The requirements (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*, the requirement can then be weakened to

$$\forall q \in Q : Inv(q) \Rightarrow aggr(Impl_i(q)) = Spec_i(aggr(q)). \quad (3)$$

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 a proper invariant that makes (3) satisfied, and second, we must prove 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 the required invariant (the reachable state space) automatically.

3 CHECKING AGGREGATION ABSTRACTIONS ON-THE-FLY

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 Boolean condition on the implementation state which consists of a set of conjuncts corresponding to each implementation step:

$$\bigwedge_i aggr(Impl_i(q)) = Spec_i(aggr(q)). \quad (4)$$

The aggregation abstraction holds if the Boolean condition 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 Murφ verifier [8] for this purpose, the technique could be used with other model checkers, including model checkers based on BDDs [22] or other symbolic representations.

3.1 Murφ description language and verifier system

Murφ is a high-level description language 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 Murφ program is an implicit description of a state graph, which consists of a collection of *state variables* and *transition rules*. The states of the graph are assignments to each global state variables 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:

`Rule` *condition* `==>` *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.

Murφ has an automatic verifier which generates all of the reachable states of the described system. Execution of a Murφ 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 Murφ verifier tries them *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.

3.2 Checking aggregation abstractions using Murφ

Normally, a description of a single protocol implementation is specified in Murφ, 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 Murφ. 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 Boolean condition (4) is using an *invariant* of Murφ. 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. Using in-line *assert* commands of Murφ, each conjunct of the Boolean 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 ==> Begin ACTION-STATEMENT; Endrule;`):

```
Rule "Transition relation for Impl_i"
  CONDITION
==>
Var i0, i1: ImplState;
Begin
  i0 := current_state;  ACTION-STATEMENT;  i1 := current_state;
  Assert Spec_i(aggr(i0)) = aggr(i1);
Endrule;
```

The two local variables i_0 and i_1 contain the implementation state before and after the execution of the rule respectively. The assert statement expresses the corresponding commutativity requirement using the functions defined for specification steps and the aggregation function.

The Murφ 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 only.

4 EXAMPLE: FLASH CACHE COHERENCE PROTOCOL

This section illustrates our technique on the cache coherence protocol used in the Stanford FLASH multiprocessor system [17, 12].

4.1 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. 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). 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 *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 *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.

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, 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. Then the home grants the request by sending a PUTX to the requesting node*. If there are no shared copies, the home simply 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.

*This 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.

During the read miss transaction, an operation called 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 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 a 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.

When a remote 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 The aggregation function

To define the aggregation function *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 with an exclusive copy, sends a PUT (or PUTX) reply, granting the request. A write-back transaction begins with invalidating an exclusive copy and sending a WB request to the home; and this is the commit step of the transaction because a part of the specification variables are already updated at this moment and the write-back request can not 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 it receives at the end of the transaction: putting the data in the message into its cache and setting the state to *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 and need to be processed to complete the committed transac-

tions: SWB 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 reseting 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 Murφ using a "for-loop" indexed on the network queue. Figure 2 shows the definition of the function. An implementation state is declared as a record consisting of an array *PNet* for network containing reply messages, an array *QNet* for network containing request messages, and an array *Procs* modeling processors with caches. Each message is also a record containing fields for its destination *dst*, source *src*, and data *Data*. From an implementation state *ist*, the function computes a specification state using a local variable *sst* to be returned. First, the specification variables of *ist* is copied into *sst*. Then, in the second for-loop, *sst* is modified by simulating to process each message in the network in the implementation state *ist* if it is one of such types that completes a committed transaction.

4.3 Checking the aggregation abstraction using Murφ

We illustrate the details on one of the implementation step (i.e., one of the requirements) of the protocol: a commit step of the transaction processing a write miss. Figure 3 shows the Murφ function for the specification step which corresponds to the transaction. As before, a specification state is declared as a record consisting of an array of cache states and data for each processor, and main memory. The function returns a specification state which is obtained by processing a write miss transaction atomically: if *oldproc* owns an exclusive copy, the exclusive data is transferred to processor *newproc*; otherwise if there is no exclusive copy in any processors, an exclusive copy is granted to processor *newproc* by copying the data in main memory to its cache.

To check the aggregation abstraction on-the-fly, we make sure that the rules in Murφ description correspond exactly to the implementation steps $Impl_i$ of the protocol. Figure 4 presents the detailed rule for the implementation step where a remote node having an exclusive copy grants the ownership transfer by sending a PUTX reply to the requesting node. The guard condition of the rule checks if there is a GETX request on the head of the request queue from *src* to *dst* (which is a remote) and the node *dst* contains an exclusive copy of the memory line. In the action statement, the processor in the node *dst* invalidates its own copy and sends out a PUTX reply to the requesting node.

Without changing the original description, we simply add a few lines of commands to check the commutativity requirement (additions to the original

```
Function Faggr(ist:ImplState): SpecState;
Var  sst: SpecState;      -- specification state to be returned
Begin
 For i:Proc Do                                    --
   sst.State[i] := ist.Procs[i].Cache.State;  -- Copy the specification
   sst.Data[i]  := ist.Procs[i].Cache.Value;  -- variables from the current
 EndFor;                                          -- state of implementation
 sst.Memory   := ist.Memory;                      --

 For i:Queue do          -- Check each message in the network
   If i < ist.PNet.Count then    -- for the reply queue
     If ist.PNet.Message[i].Mtype=Putt then
       -- Simulate processing a 'put' reply
       If ist.PNet.Message[i].dst=Home then
         sst.Memory := ist.PNet.Message[i].Data;
       Endif;
       If ! ist.Procs[ist.PNet.Message[i].dst].Cache.InvMarked then
         sst.Data[ist.PNet.Message[i].dst] := ist.PNet.Message[i].Data;
         sst.State[ist.PNet.Message[i].dst] := Shared;
       Else
         sst.State[ist.PNet.Message[i].dst] := Invalid;
       EndIf;
     Elsif ist.PNet.Message[i].Mtype=PutX then
       -- Simulate processing a 'putx' reply
       sst.Data[ist.PNet.Message[i].dst] := ist.PNet.Message[i].Data;
       sst.State[ist.PNet.Message[i].dst] := Exclusive;
     endif;
   Endif;
   If i < ist.QNet.Count then    -- for the request queue
     -- Simulate processing a 'WB' or a 'ShWB' sent to the home
     if ist.QNet.Message[i].Mtype=WB | ist.QNet.Message[i].Mtype=ShWB
     then sst.Memory := ist.QNet.Message[i].Data;
     endif;
   Endif;
 EndFor;
 Return sst;
End;
```

Figure 2 The aggregation function written in Murφ

implementation description are marked with stars in the figure). First, local variables i_0, i_1, s_0, and s_1 are declared to be used to copy the implementation states and specification states, respectively, before and after the execution of the rule. Because this implementation step is the commit step of a write-miss transaction, the corresponding specification step is "Atomic_GetX" in Figure 3. Using the declaration of the specification step and the aggregation function, the assert statement explicitly specifies the corresponding commutativity requirement.

```
Function Atomic_GetX(st:SpecState;  oldproc,newproc:Proc): SpecState;
Begin
  If st.State[oldproc] = Exclusive & oldproc != newproc then
    st.State[oldproc] := Invalid;
    st.State[newproc] := Exclusive;
    st.Data[newproc] := st.Data[oldproc];
    Return st;
  Elsif Forall i:Proc Do st.State[i] != Exclusive EndForall then
    st.State[Home] := Invalid;
    st.State[newproc] := Exclusive;
    st.Data[newproc] := st.Memory;
    Return st;
  Else Return st;
  Endif;
End;
```

Figure 3 The specification step processing a write miss in the FLASH protocol

```
Ruleset i : Queue Do        -- Parameterized rule for each message in the queue
Alias request: QNet.Message[i];    -- Choose an arbitrary request in the queue
      dst    : request.dst;        -- the destination of the request
      SRC    : request.SRC Do      -- the node that initiated the request
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
  & Procs[dst].Cache.State = Exclusive  -- The node dst has an exclusive copy
  & Pspace(1) & Qspace(1)   -- There are enough spaces in the network
  ==>
* Var  i0,i1: ImplState;  s0,s1: SpecState;  -- local variables
  Begin
*   i0 := ThisImplState();
*   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(i);
*   i1 := ThisImplState();
*   s1 := Faggr(i1);
*   s0 := Atomic_GetX(s0, i0.QNet.Message[i].dst, i0.QNet.Message[i].SRC);
*   Assert Equiv(s0,s1) "NI Remote GetX (Commit)";
Endrule;
Endalias;
Endruleset;
```

Figure 4 A sample Murφ rule with an assert statement for checking the corresponding commutativity requirement. The lines marked with a star are additions to the original implementation description.

4.4 Experiments

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

To check the abstraction automatically, we have run Parallel Murφ on 32 ULTRA SPARC processors [27]. For the protocol with 3 processing nodes and request/reply message queues of size 5, the verifier explored 457,558 states in 126 seconds; for 4 processing nodes and queues of size 3, about 19 million states in 72 minutes. As expected, the number of states explored (also the usage of memory) is exactly same as that found for exploring reachable state space of the implementation model.

5 CONCLUSION

By limiting the verification problems to finite-state systems, we proposed an efficient technique for checking aggregation abstractions without reasoning about invariants of the system. 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 [26]. However, the proof 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.

Acknowledgment

This research was supported by the Advanced Research Projects Agency through NASA grant NAG-2-891. We thank Ulrich Stern for helping run Parallel Murφ and David Culler at Berkeley for letting us use their NOW.

REFERENCES

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings.

Theoretical Computer Science, 82:253–284, 1991.

[2] Randal Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[3] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), April 1986.

[4] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.

[5] Rance Cleaveland and Steve Sims. The NCSU Concurrency Workbench. In *Computer Aided Verification, 8th International Conference, CAV'96*, pages 394–397. Springer-Verlag, 1996.

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

[7] 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.

[8] David L. Dill. The Murφ verification system. In *Computer Aided Verification, 8th International Conference, CAV'96*, pages 390–393. Springer-Verlag, July 1996.

[9] Susanne Graf. Verification of a distributed cache memory by using abstractions. In *6th International Conference on Computer-Aided Verification*, pages 207–219, 1994.

[10] Susanne Graf and Claire Loiseaux. A tool for symbolic program verification and abstraction. In *5th International Conference on Computer-Aided Verification*, pages 71–84, 1993.

[11] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, pages 662–681, March 1996.

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

[13] Alan John Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*, chapter 4 on 'BDD Blow-Up Representing Sets of States', pages 41–49. Stanford University, December 1995. Ph.D. Thesis.

[14] C. Norris Ip and David L. Dill. Verifying systems with replicated components in Murφ. In *8th International Conference on Computer-Aided Verification*, pages 147–158, 1996.

[15] Chung-Wah Norris Ip. *State Reduction Methods for Automatic Formal Verification.* PhD thesis, Stanford University, December 1996.

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

[17] 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.

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

[19] Leslie Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[20] David Long. *Model Checking, Abstraction and Compositional Verification.* PhD thesis, Carnegie Mellon University, July 1993.

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

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

[23] R. Milner. An algebraic definition of simulation between programs. In *Proc. of the 2nd International Joint Conference on Artificial Intellegence*, pages 481–489, 1971.

[24] Seungjoon Park. *Computer Assisted Analysis of Multiprocessor Memory Systems.* PhD thesis, Stanford University, June 1996.

[25] Seungjoon Park and David L. Dill. Protocol verification by aggregation of distributed transactions. In *Computer Aided Verification, 8th International Conference, CAV'96*, pages 300–310. Springer-Verlag, July 1996.

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

[27] Ulrich Stern and David L. Dill. Parallelizing the Murφ verifier. In *Computer Aided Verification, 9th International Conference, CAV'97.* Springer-Verlag, June 1997.