Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking

K. L. McMillan

Cadence Berkeley Labs

Abstract. We consider the formal verification of the cache coherence protocol of the Stanford FLASH multiprocessor for N processors. The proof uses the SMV proof assistant, a proof system based on symbolic model checking. The proof process is described step by step. The protocol model is derived from an earlier proof of the FLASH protocol, using the PVS system, allowing a direct comparison between the two methods.

1 Introduction

The verification of cache coherence protocols was perhaps the earliest commercial application of model checking [MS91]. Later, more efficient model checking methods were developed for this application [CD93], and compositional methods were applied to show that a verified protocol was implemented correctly in hardware [Eir98]. However, these techniques were unsound, in the sense that they could be applied only to fixed number N of nodes in the network, whereas in fact N had no useful upper bound. This left open the possibility that a protocol error was missed, which only manifested itself for N greater than the size verifiable by model checking.

Verification for arbitrary N can be accomplished in a number of ways. For example, Park [PD96] applied the general purpose theorem prover PVS [ORS92] to the verify the cache protocol of the Stanford FLASH multiprocessor [KOH$^+$94] for arbitrary N. This is a laborious process, since inductive invariants must be devised, and the theorem prover must be manually guided. It was also shown that in some cases, a protocol could be verified automatically for arbitrary N, using finite state methods with a "symbolic state" abstraction [PD93,ID96]. This abstraction cannot be applied to all protocols, however, and is too coarse to prove liveness. In fact, the first error detected in a cache protocol by model checking was a liveness error.

Thus, there is some interest in finding methods of proof for arbitrary N that are at the same time general, capable of proving liveness, and not unduly time consuming. Here, we consider applying methods of compositional model checking to the problem. This approach has the advantage that parameterized systems can be proved correct without the need to state inductive invariants, since invariant information is obtained by model checking abstract systems. We use the SMV proof assistant [McM99] to verify both safety and liveness of the FLASH protocol. This protocol has the advantage that it has been approached previously using a variety of methods, and thus offers a good basis for comparison. In particular, we verify the same model used in Park's PVS proof (with one minor change). Relative to that proof, the proof we obtain is at least an order of magnitude more concise. We will begin in section 2 with some background on the SMV proof assistant and compositional model checking. Then in section 3, the protocol model is described. The next section covers the proof of correctness. Then in section 5, a comparison is made with previous approaches.

2 Background

The SMV proof assistant [McM99] is designed to support the reduction of correctness conditions for unbounded or infinite-state systems to lemmas that can be verified by model checking checking finite state systems. The reduction to finite-state is done by methods of *abstract interpretation*. In effect, we throw away information about the model that is not needed to prove a given property. To make this effective, however, we must first break the desired property into localized properties that rely on only part of the system state. SMV provides two alternatives for this purpose: compositional proof and temporal case splitting.

Compositional proof The correctness of an implementation model is specified by a collection of temporal logic properties, usually with respect to a reference model. In our case, the reference model defines the programmer's model of the memory. Temporal specifications relative to a reference model are called *refinement relations*. In the SMV system, these can be specified in one of two ways. We might, for example, write a linear time temporal logic property, such as the following:

p : **assert G** $(\mathit{cnd} \Rightarrow \mathit{impl\_data} = \mathit{ref\_data})$;

This says that, at all times, if condition *cnd* holds, then *impl_data* (some data value in the implementation) must be equal to *ref_data* (some corresponding value in the reference model). Typically, *ref_data* will refer to some data computed by the reference model in the past. That is, there is no need for the implementation and the reference model to operate in "lockstep". On the other hand, we might express the same property as a "layer":

layer p: **if**(cnd) $\mathit{impl\_data} := \mathit{ref\_data}$;

This has the same semantics as the property above, but it tells SMV that we wish to use p as an abstraction of variable *impl_data* when proving other properties.

The refinement relations are proved by mutual course-of-values induction over time [McM99]. Each refinement relation must be a temporal property of the form $G\phi$, meaning that ϕ is true for all times t. To prove that ϕ is true at time t, we can assume by induction that the other refinement relations hold for all times less than t. This is useful in a methodology based on model checking, because the notion that ψ up to time $t-1$ implies ϕ at time t can be expressed in temporal logic as $\neg(\psi \; \mathcal{U} \; \neg\phi)$. Hence, this proposition can be checked by a model checker. To tell SMV to use property p up to $t-1$ to prove property q at time t, we use the following notation:

using (p) **prove** q;

The parentheses indicate that p is to be used up to time $t-1$ and not t. If p is expressed as a layer, we don't need to use this directive. SMV will determine whether *impl_data* is relevant to property q, using a dependency analysis. If so it will use property p in place of the implementation definition of *impl_data*. In this way, the proof of q can be localized to a small part of the implementation.

This mutually inductive approach is important because it allows us to assume, for example, when proving correctness of data in one cache, that the data in other caches have been correct in the past. Note this technique is quite different from the method of proof by invariant, in which we show that some state property at time $t-1$ implies itself at t. In our case, the properties are temporal, and the inductive hypotheses are assumed for all times less than t, and not just at $t-1$. This is important, since it allows us to avoid writing inductive invariants.

Temporal case splitting An alternative approach to localization is to specialize the properties we wish to prove, so that they depend on only a finite part of the overall state. For example, suppose that we have a state variable v, which is read and written by a collection of processes $1 \ldots n$. We wish to prove a property p of v, of the form $G\phi$. We add an auxiliary state variable w to the model which records a pointer to the most recent process to write the variable v (this is supported by a definitional mechanism in the proof assistant). Now, suppose we can prove for *all* process indices i that $G((w = i) \Rightarrow \phi)$, that is, ϕ holds at all those times when the most recent writer is process i. We can then infer that $G\phi$ holds, since at all times w must have some value. We call this "splitting cases" on the given variable, since it generates a parameterized property with one instance for each value of the given variable. We can tell SMV to do this as follows:

forall(i **in** *PROCS*)
    subcase $p[i]$ **of** p **for** w=i;

where *PROCS* is the type of process indices. This generates, for every i, an instance $p[i]$ of p, of the form $G((w = i) \Rightarrow \phi)$. The importance of this approach is that for any given parameter value i, we may be able to verify the property by abstracting away all processes except process i, since the particular case $w = i$ depends directly only on process p_i.

Abstract interpretation SMV uses abstract interpretation to reduce the verification of each parameterized property to a tractable model checking problem. The difficulty is that there may be variables in the model with large or unbounded ranges (such as memory addresses) and arrays with a large or unbounded number of elements (such as memory arrays), and the number of instances of the property may be unbounded. We deal with this problem by using abstract interpretation to reduce each data type to a small number of abstract values. For example, suppose that we have a property with a parameter i that ranges over memory addresses. We can reduce the type T of memory addresses to a set containing two values: the parameter value i, and an abstract symbol $T \setminus i$ that represents all values other than i. In the abstract interpretation, accessing a memory array m at location i will produce the value $m[i]$ of that location, whereas accessing the array at $T \setminus i$ produces $\bot$, a symbol representing no information about the value. The net effect of this abstraction is that, for each time we create a parameter by "splitting cases" on a variable of type T, there is one value in the abstract type and one element in each abstracted array indexed by that type. Thus, case splits come at the cost of increased verification complexity.

Normally, a suitable abstract interpretation for each data type can be chosen automatically. However, it can also be specified manually, as in the following:

using $T \Rightarrow \{i,j\}$ **prove** q;

This tells SMV to distinguish the values i and j of type T, but to represent all other values by an abstract symbol. If there are two parameters i and j of type T, then the proof assistant may, for example, split the problem into two cases: one where $i = j$ and one where $i \neq j$. Alternatively, it may consider separately the cases $i < j$, $i = j$ and $i > j$, if information about the order of these values is important to the property. This behavior depends on the declaration of the data type: a *scalarset* type considers only equality, while an *ordset* type considers the relative order of parameter values as well.

The abstractions used by the proof assistant are sound, in the sense that validity of a formula in the abstract interpretation implies validity in the concrete model for *all valuations* of the parameters. Thus, we need not check all parameter valuations separately. Of course, it is possible that the abstraction used may be too coarse to verify the given property (*i.e.*, the truth value of the property in the abstract model may be $\bot$) even though the property is true. Note, however that the user does not need to verify the correctness of the abstractions used, since these are drawn from a fixed set that is built into the proof assistant.

The proof process thus proceeds in the following steps. First, the user specifies refinement relations (and other lemmas, as necessary), which are proved by mutual temporal induction. These properties are parameterized by "splitting cases" on appropriate variables, so that any particular case depends on only a finite part of the system state. Finally, the proof assistant abstracts the model relative to the parameter values, reducing the types with large or unbounded ranges to small finite sets. The resulting proof obligations are discharged by a model checker. In the sequel we will consider how this general methodology can be applied to Park's model of the FLASH cache coherence protocol.

3 The protocol model

A FLASH system consists of a collection of N processors, each with a local cache memory. We will refer to a processor/cache pair as a *node*. The nodes act asynchronously, exchanging messages via a network with arbitrary latency. The protocol ensures that multiple cached copies of a given address are consistent, in an appropriate sense. Each memory address has a *home* node. The home node contains the master copy of the address in main memory, and also a directory entry, indicating the set of caches that may hold the given address, and whether the cached data may be modified (*dirty*). In the actual system, this set is maintained in a list structure. In Park's model, there is a variable containing the head of the list, and a set representing the remainder of the list. The model has only one memory address. Generalization to an arbitrary number of memory addresses is trivial, however.

A cache may contain the given address in one of three states: *invalid* (not present), *shared* (readable) and *exclusive* (readable and writable). As the name implies, there can be only one cache in the exclusive state at any time.

The protocol works roughly as follows. A node requiring read access to a given address sends a *Get* message to the home node. If there are no exclusive copies in the system (the directory entry is not marked *dirty*), then the home fetches the value from main memory, and returns it in a *Put* message. The index of the requesting node is added to the sharing list in the directory, and the requesting cache enters the shared state. On the other hand, if the address is dirty, there must be an exclusive copy at some node (the *owner*) and the data in main memory are obsolete. The home thus *forwards* the *Get* request to the owner. The owner is then expected to send the data to the original requester and change to the shared state. Modified data are also returned to the home node via a sharing write-back (*ShWB*) message. This leaves the address clean, with two nodes on the sharing list. While this transaction is in progress, the directory remains in the *pending* state, and will not accept new requests.

When a node requires write access, it sends a *GetX* message to the home, to obtain an exclusive copy. If the address is not dirty, the home sends invalidation messages (*Inv*) to each node on the sharing list. It immediately returns the data in a *PutX* message without waiting for the invalidations to complete (that is, we model the "eager" mode [KOH$^+$94]). The directory then remains in the pending state until acknowledgments (*InvAck*) are obtained for all the invalidations sent. On the other hand, if the address is dirty, the *GetX* is forwarded to the owner, which is expected to send a *PutX* to the requester, enter the invalid state, and send an acknowledgment message *FAck* to the home. Again, the directory stays in the pending state until the *FAck* is received.

The owner may eject the address from its cache at any time, by sending modified data to the home in a write-back message (*WB*). This introduces a race condition, since a forwarded *Get* or *GetX* may be on its way to the owner. If this happens, the former owner will return a negative acknowledgment to the home, aborting the transaction.

Park's model encodes the network state in an unusual way. That is, instead of a uniform collection of message buffers, it has specialized sets of message buffers for each class of messages. For example, *unet* holds the *Put* and *Get* messages (with one entry per requester), while *invnet* holds the *Inv* and *InvAck* messages (one per node) and *wbnet* holds the write-back messages (one global entry). In particular, *unet*[i] holds *Get* messages for requester i, even if the destination of the message is some other node, due to forwarding. This encoding is in some ways convenient for model checking, although we will observe one case in which it is inconvenient.

This model was translated rather directly from PVS into the SMV notation. One notable difference is that the original uses a counter *real* to hold the number of pending invalidations. Since SMV has no reasonable way to deal with the cardinality of sets, this is replaced by the set of processor indices that have pending invalidations. As stated earlier, we would like to prove both coherence and liveness of this model.

4 Proof of coherence

We begin with the specification. Ideally, a cache coherence protocol would satisfy the condition of *sequential consistency* [Lam79]. That is, there should always exists a global sequence of reads and writes that agrees with the local observations of each processor, and in which every read gets the value of the most recent write to the given address. Our version of the FLASH protocol does not satisfy this condition. However, it does satisfy the weaker condition that reads and writes to a given cache line are sequentially consistent. We will refer to this condition as *coherence*.

In fact, we will prove a considerably stronger condition than coherence. That is, a read at a given node gets the value of the most recent write at the last time the address was *observable* to that node. We require that all values written by a given node be observable to that node. However, other than this, the definition of *observable* is up to the implementer. It is easy to show that any protocol implementing this model is coherent. We do not do that here, however, since this is a generic theorem, not related to the FLASH protocol *per se*. The intuition behind this definition is that if a node is holding a shared copy that is pending invalidation, then the address is not observable. Instead, the node sees the value of the address before the invalidating transaction began.

To express our specification, we add an auxiliary state variable m to the model, which holds the most recent value written to the address:

if(*store*) **next**(m) := *store_data*;

Here, *store* is an implementation variable indicating that a store is occurring, and *store_data* is the data value being stored. This definition says that when *store* is true, m is modified to hold the value of *store_data*. SMV allows such definitions of new variables to be added, so long as the implementation does not depend on them. We now specify correctness of data in the caches relative to this variable. It is most convenient to write the specification as a *layer* in the SMV system, which will cause it to be used by default in the proof of any property involving the cache data. We could, however, write the property as an ordinary temporal assertion. Here is the specification as a layer:

layer *L1*: **forall**(i **in** *Proc*)
    if(*readable*[i]) *cache*[i].*data* := m **whenlast** (*obs*[i] $\vee$ *last_writer* $= i$);

That is, the cache sees the value of m at the last time it was observable. Note that "x whenlast c" is a shorthand for the value of x at the most recent time that c was true (and is undefined if c has never been true). The "forall" prefix

```
scalarset Proc undefined;

/* ... Other type declarations go here ... */

module main(){

   /* The reference model */
      abstract m : Data;
      if(store) next(m) := store_data;

   /* ... The implementation model goes here ...*/

   /* coherence specification */
      abstract obs : array Proc of boolean;
      abstract last_writer : Proc;
      if(store) next(last_writer) := dst;

      layer L1: forall(i in Proc)
         if(readable[i]) cache[i].data := m whenlast (obs[i] ∨ last_writer = i);

   /* witness functions */
      obs[i] := ¬(dir.pending ∧ collecting) ∨ cache[i].state = exclusive;
      init(m) := mem;     /* make sure reference model has correct initial value */

   /* ... The proof goes here ... */
}
```

Fig. 1. Structure of SMV input file

creates one instance of the property for each node. The property requires that the address must be observable if the most recent writer was i (as indicated by the auxiliary variable *last_writer*). Otherwise, observability is determined by the predicate $obs[i]$. Note that *obs* is not defined in the specification. We need only show that the specification holds for *some* definition of *obs*. To prove *L1*, we use the following *witness function* for *obs*:

$$obs[i] := \neg(dir.pending \wedge collecting) \vee cache[i].state = exclusive;$$

That is, shared copies are "out of date" when there is a transaction pending in the directory that is collecting invalidation acknowledgments. Here, *collecting* is an auxiliary variable that records whether the current pending transaction sent invalidations. It is needed because the directory does not directly record this fact. Figure 1 shows the structure of the SMV input file at this point.[1]

The first thing we do is to use the model checker to check that the property is true for $N = 3$. To do this, we use the following declaration of the type *Proc*:

```
typedef Proc 0..2;
```

This check is done to give us confidence that the property is actually true before attempting to prove it for the general case. In fact, several errors in the model

[1] The complete file can be obtained at `http://www-cad.eecs.berkeley.edu/~kenmcmil`

were found in this way, resulting from incorrect translation from the PVS description. As a "sanity check", the $N = 3$ case was also checked for absence of deadlock using the model checker. Having some confidence in the correctness of the model, we now attempt to verify the property *L1* for arbitrary N. To do this, we change the declaration of type *Proc* to the following:

```
scalarset Proc undefined;
```

This tells SMV that type *Proc* is used in a symmetric way (which governs the abstraction used for data of type *Proc*) and that the range of the type is unknown.

Our general approach to the proof is to attempt to check the property using an abstraction of the data types. When a counterexample is produced, the likely source is a message produced by some processor that has been "abstracted out" of the model. There are two tactics we can apply at this point. Either we "split cases" on the producer of this message (which will, in effect, add this processor to the abstraction), or we add a lemma that rules out the interfering message. The first approach is simpler for the user, but adds to the complexity of the model checking.

When we first attempt to verify property *L1*, SMV uses an abstraction of type *Proc* that contains one fixed value i, the index of the cache we are verifying. An abstract symbol is used to represent all of the other processor indices. This abstraction is far too coarse to verify the property, as we can see from the counterexample we obtain. This shows a case in which i is the home node, and obtains a readable copy of the data. However, one of the processors that is abstracted away executes a store operation at this point. This is not a possible behavior of the system since only caches in the exclusive state can execute a store. However, the abstraction cannot rule it out, since the state of the abstracted processors is unknown in the abstraction. Here, there are two possible approaches. We can split cases on the most recent processor to execute a store. Or, we might write a lemma saying that only the owner of the cache line (the most recent to receive an exclusive copy) can write. Either will rule out our counterexample.

Since it is a bit simpler, we try the case splitting approach first. We add the following declaration:

```
forall(j in Proc)
    subcase L1[j] of cache[i].data//L1 for last_writer = j;
```

where *last_writer* is the auxiliary variable pointing to the most recent node to execute a *Store*. Since the parameter j becomes a distinguished value in the abstraction, this node is no longer abstracted out. Thus, we rule out the false counterexample in which an abstracted node corrupts the reference model data by executing a *Store*.

Because *Proc* is a scalarset, we now have two cases to consider: $i = j$ and $i \neq j$. In the latter case, we get a counterexample in which an abstracted processor sends a write-back, thus corrupting the state of main memory. This incorrect value is then loaded into the cache of processor i, violating our property. To fix this problem, we could split cases on the most recent processor to execute a write-back. However, we do not wish to include too many processors in the abstraction. Instead, let us split the problem into two parts by adding a lemma, stating that data in main memory are always correct when there is no exclusive copy in the system. This will rule out the false counterexample, although it increases our proof effort, since we have to prove this property separately. The lemma is as follows:

layer *L2*: **if**(¬*dir.dirty*)*mem* := *m*;

That is, when the directory is not in the dirty state, main memory must have the value of the most recent write. Because it is a layer, SMV will use this specification of *mem* by default when proving *L1* (and *vice versa*), using mutual temporal induction. With this added lemma, we get a counterexample in which the home node has the line in its cache. Node i requests a copy of the cache line, which is then forwarded directly from the home's cache (note the cache in the home node is not the same as the main memory *mem*). Unfortunately, the home node is abstracted in this counterexample. We can tell this because the value of *home* in the counterexample is the abstract symbol representing "other than i or j". At this point, we could add a lemma stating that the home node always sends correct data. This strategy seems unlikely to succeed in the long run, however, since a large part of the control of the protocol depends on whether a given node is the home node or not. Thus, we can expect many other false counterexamples to arise if the home node index is not present in the abstraction. Therefore, let us split cases on the value of *home*:

```
forall(k in Proc)
    subcase L1[j][k] of cache[i].data//L1[j] for home = k;
```

We now have an abstract model with three processors represented: the node i, whose cache we are verifying, the most recent writer j, and the home k. With this model, we get a counterexample in which data is forwarded from an abstracted processor to processor i. Thus, processor i receives incorrect data. Note that in principle, the forwarded value should be correct, since we are allowed to assume that *L1* holds for all nodes up to time $t-1$. However, by default *L1* is only instantiated for the process indices that are in the abstract model, that is, i, j and k. We need this property to hold for "the processor currently forwarding data", which in the model is indicated by the implementation variable *dst*. This is the index of the processor that is current receiving a request to forward data. Since SMV does not know enough to instantiate property *L1* for $i = dst$, we do it manually, adding the following lemma:

L1a : **assert G** (*cache*[*dst*].*state* = *exclusive* ⇒ *cache*[*dst*].*data* = *m*);

Note, we have weakened *L1a* a bit, omitting the case when the state is *shared*, since we don't need it. This lemma can be trivially proved as a special case of *L1*, so we omit the proof here. We now use this instantiation of *L1* up to time $t-1$, with the following declaration:

using (*L1a*) **prove** *cache*[*i*].*data*//*L1*[*j*][*k*];

This ensures that good data are always forwarded from other processors. However, it is still possible that a *Store* will occur while the data are in transit. This happens in the next counterexample. Processor j gets an exclusive copy of the data. However, an abstracted processor sends a write-back message to the home, causing it to think that the line is no longer owned by j. As a result, it sends a shared copy to i. Now processor j, which still thinks it has an exclusive copy, writes the line, causing k's data to be incorrect. The problem here is that it is not possible for another processor to have an exclusive copy while j is holding one (hence the write-back is impossible). However, this information is lost by the abstraction. To solve this problem, we might try case splitting on the sender of the write-back, but at this point the abstract model is already nearly too large to handle. Instead, we write a lemma stating when it is possible for a processor to be in the exclusive state. That is, it is possible when the directory thinks that the line is "dirty", and when no other processor holds an exclusive copy. Note that an exclusive copy can be held in a cache, or it can exist in a *PutX* message *en route* to a node. Thus, we write the following lemma:

$$
\begin{array}{l}
\mathit{L3}[i]\text{: }\mathbf{assert\ G}\ (\mathit{dst\_dirty} \Rightarrow \mathit{dir.dirty} \wedge \\
\qquad ((\mathit{cache}[i].\mathit{state} = \mathit{exclusive} \vee \mathit{unet}[i].\mathit{mtype} = \mathit{PutX}) \Rightarrow \mathit{dst} = i));
\end{array}
$$

That is, if the current "destination" node is in the exclusive state (*dst_dirty*), then the directory must be in the dirty state, and no other processor i may be in the exclusive state, or have a *PutX* message in transit to it. Note, that rather than writing this property for all pairs of nodes, we have written it only relative to the current *dst*, since we only need to rule out bad write-back messages. If we wrote the lemma for all pairs (i, j) we would then have to instantiate it manually for $\mathit{dst} = j$, as above. To prove this lemma, we clearly must split cases on *dst*, so that the state of node *dst* will be present in the abstraction. In addition, we will split cases on *home*, for the reasons mentioned above. As usual with such "non-interference" lemmas, we will assume that the general lemma holds up to time $t-1$ to prove a specific case at time t. Here is the SMV declaration we use:

```
forall(j in Proc) forall(k in Proc){
    subcase L3[i][j][k] of L3[i] for home = j ∧ dst = k;
    using (L3) prove L3[i][j][k];
}
```

From the ensuing counterexamples we discover that in fact there are three other types of messages that represent an "exclusive" copy: a write-back, returning ownership to the directory, a "sharing write-back" message, that is performing the same function but leaving shared copies in the system, and a *Put* message from the owner to home, which has the same effect as a write-back, but is elicited when the home cache needs a copy of the cache line. Thus, we add the following requirements to *L3*, in case *dst* is in the exclusive state:

$$\mathit{wbnet.mtype}{=}\mathit{Empty} \wedge \mathit{shwbnet.mtype}{\neq}\mathit{ShWB} \wedge \mathit{unet}[\mathit{home}].\mathit{mtype} \neq \mathit{Put}$$

At this point, in trying to verify *L3*, we run into a problem: the verification does not complete within our limit of 2GB of memory. We now have two choices. We could further decompose the proof into cases or lemmas, or we could help the model checker by adjusting the abstraction manually. Since the verification seems close to finishing, we choose the latter course. Adjustments to the abstraction can be made by "freeing" some variables that we think are irrelevant (*i.e.*, we ignore their definitions and leave them unconstrained). The topic of choosing an appropriate abstraction for model checking is beyond the scope of this article. Suffice it to say for our purposes that making a few educated guesses allows us to reduce the memory consumption of the model checker to about 270 MB, which is well within the capacity of our server, and that *L3* is verified.

Now, having proved our "non-interference lemma", let us return to the proof of *L1* (data correctness in the cache), adding *L3* to the list of properties used to prove *L1*. Now we get a counterexample showing another kind of "interference". That is, node i gets a shared copy, after which node j requests and obtains an exclusive copy. This causes an invalidation message to be sent to node i, and the directory state to be set to "pending", *i.e.*, awaiting invalidation acknowledgments. A new transaction cannot be started until all invalidations are complete. Oddly, then, node j (the owner) now receives a forwarded *Get* request for some abstracted node. How does this happen? It is an artifact of the way the network is coded in the model. Even though the forwarded *Get* message for a node n can only be sent from the home, it is stored in the message buffer $\mathit{unet}[n]$. This encoding, chosen by Park for his PVS proof [PD96], helps us in some ways, but hurts us here, since $\mathit{unet}[n]$ in this case is abstracted. Thus, the abstract model allows this incorrect message to arrive at the owner node while invalidations are pending. The owner sends a *FAck* message back to the home, acknowledging this bogus request. The home, receiving the bogus *FAck*, clears its pending bit. Now node i is holding stale data, but the pending bit is not set, violating *L1*.

At this point, we might choose to recode the model of the network in such way that a special buffer is reserved for forwarded *Get* messages. One such buffer would suffice, since only one such message can be in the system at a time. This buffer would always be present in the abstraction, so *Get* messages forwarded from abstracted nodes would not trouble us. However, for comparison purposes, we wish to use the same model that was used in the PVS proof. In any event, we can get the same effect by introducing the proposed message buffer as an auxiliary variable. We then prove that forwarded *Get* messages can only occur when they are found in this imaginary message buffer. This technique of replacing an inconvenient encoding of state information by a more convenient encoding is useful in many contexts. Here is the property we write:

$$\begin{aligned} &\mathit{L4}[i]\text{: }\mathbf{assert\ G}\ (\mathit{unet}[i].\mathit{mtype}\ \mathbf{in}\ \{\mathit{Get}, \mathit{GetX}\} \wedge \mathit{unet}[i].\mathit{proc} \neq \mathit{home} \\ &\qquad\qquad\qquad\qquad \Rightarrow (\mathit{unet}[i].\mathit{mtype} = \mathit{fwd\_get}));\end{aligned}$$

Here, *fwd_get* is our imaginary message buffer for forward *Get* messages. We say that if there is such a message in any message buffer i, and if the destination is not home (*i.e.*, it is being forwarded to some other node), then it must match our imaginary buffer.

Now we attempt to prove this. Initially, we try splitting cases on just the home node. This produces a counterexample in which a *GetX* is forwarded for node i and entered in *fwd_get*. However, this imaginary message doesn't specify that i was the original sender of the *GetX*. Some abstracted node's message buffer issues the *GetX*, clearing *fwd_get*. Now node i's message buffer violates the property, since it has a forwarded *GetX*, but *fwd_get* is empty. We decide to split cases on the message buffer that issued the bogus *GetX*, so that it will not be abstract in the model. This seems reasonable, since presently there are only two parameters in the property:

subcase $L4[i][j][k]$ **of** $L4[i]$ **for** $fwd\_src = j \wedge home = k$;

Here, *fwd_src* is an auxiliary variable that remembers which message buffer most recently issued a forwarded *Get*. This should solve the problem, since both message buffers i and j cannot hold a forwarded *Get* at the same time. Unfortunately, we now get a counterexample in which two such forwarded messages *do* exist at the same time. In this counterexample, one node requests an exclusive copy, and its *GetX* is forwarded to the owner (some abstracted node). In principle, a new transaction cannot be started until the directory knows that this one has terminated. However, at this point the directory receives in invalidation acknowledgement message (*InvAck*) from some abstracted node. Such messages should never arrive during an ownership transfer (they only occur when changing from shared to exclusive). The directory doesn't know this, however, and responds to the bogus *InvAck* by terminating the transaction. This allows a new transaction to begin, which results in two forwarded *GetX* messages being in the system at the same time. At his point, to avoid another case split, we introduce another non-interference lemma. This one says that *InvAck* messages cannot arrive during an ownership transfer:

$L5$: **assert G** $(src\_invack \Rightarrow dir.pending \wedge collecting)$;

Here *src_invack* says that an *InvAck* is arriving (recall that *collecting* means that invalidations are currently being collected). We assume this lemma up to time $t-1$, to prove $L4$ at time t. With this assumption, we obtain one more counterexample, in which an abstracted message buffer violates lemma $L4$. This leads to a later violation of $L4$ by message buffer i. This should be ruled out, because we can assume the general lemma up to $t-1$ when proving case i. However, to do this, we are forced to instantiate $L4$ manually for $src = i$, just as we did above for $L1$. With this addition lemma $L4$ is verified.

This leaves lemma $L5$ to prove, that is, that no unexpected *InvAck* messages arrive. To check this, as usual, we split cases on the sender of the bogus messages, and also on the home location, as follows:

subcase $L5[i][k]$ **of** $L5$ **for** $src = i \wedge home = k$;

This is sufficient to prove the lemma. Note that when proving the non-interference lemmas, we always use all the lemmas up to time $t-1$. If nothing else, this practice cuts down on the size of the reachable state space (since the lemmas constrain the possible transitions) and may prevent us from looking at bogus counterexamples. Also, we use the same abstraction adjustments for all of the lemmas, except that for *L5*, we cannot abstract out the part of the network model that handles invalidations.

Now we return to property *L1*, data correctness in the caches. Recall that a counterexample to this property caused us to add *L4*, and in turn *L5*. Now we assume these lemmas to prove *L1*. This time the verification succeeds. All that remains is property *L2*, which specifies correctness of the contents of main memory. We use exactly the same strategy as for *L1*, that is, we split cases on the most recent node to execute a *Store*, and on the index of the home node:

subcase $L2[j][k]$ **of** $mem//L2$ **for** $last\_writer{=}j \wedge home = k$;

Using our non-interference lemmas, this property is also verified (layer *L1* is automatically assumed as the definition of the cache contents).

At this point, we have verified that contents in the caches satisfy our specification. This proof was completed in two days. In fact, an earlier proof of the same model required less time, about 12 hours. However in the present proof, a greater burden was placed on the model checker in order to make the proof simpler for presentation. Thus, most of the two days was spent waiting for the model checker to terminate. The proof of liveness is omitted here for space reasons. It is actually substantially simpler than the coherence proof and required four hours to accomplish.

5 Related work

As stated earlier, the model of the FLASH protocol that is verified here is adapted from the work of Park [PD96], who verified using the PVS theorem prover that it implements an abstract model. In this model, the cache states are updated globally in a single atomic action. This proof was done using an abstraction function that computes the result on the system state of executing all the "committed" transactions that are currently in the system. This has the effect of emptying the network of messages, producing an abstract state. It is then proved that each action of the implementation model implements some (possibly null) atomic action of the abstract model, modulo the abstraction function. This proof also requires an inductive invariant of the system to be stated and proved, since the abstraction function does not hold for unreachable states.

In fact, such invariants can be quite complex and time-consuming to produce. The lemmas in Park's refinement proof in PVS requires 776 lines (21KB) to state, perhaps half of which is taken up by invariants. This does not include the proof script required to prove the lemmas, which can be quite large. Nor does this include a proof that the atomic model implements some memory model.[2] Park does not report the time required to complete the proof, though we can infer from the 111 lemmas and theorems that it must have been substantial. The safety proof presented here takes 30 lines (1.7KB) in SMV (including the four auxiliary lemmas, case splits, instantiations and abstraction adjustments) plus

[2] For the eager model, a lemma is proved for each low level action, stating that it commutes with some high level action, but two models are not defined *per se*, and coherence is not specified.

two auxiliary variable definitions scattered in the model code, and required two days to complete. We should note there is one simplification in the model – a counter is replaced by a set. Since liveness was not proved in Park's work, a comparison of liveness proofs is not possible.

The relative conciseness of the SMV proof appears to stem from two factors. First, when using model checking of abstract models, it is not necessary to have an inductive invariant. Rather, the model checker generates the strongest invariant of the abstract model. Strengthening is required when the abstract model is too coarse (hence, the non-interference lemmas L3, L4 and L5). Note, however, that these lemmas are proved by course-of-values induction over time, rather than by simple induction. The net effect is that the non-interference lemmas are considerably simpler than a global inductive invariant. The second possible factor in the simplicity of the proof is that no intermediate model is used, and hence no abstraction function is required.

Another approach to generating an inductive invariant is to use a technique called *predicate abstraction* [SG97]. In this method, the user provides a collection of predicates on the system state, and a program generates the strongest invariant of the system that can be expressed as a boolean combination of these predicates. This method has been used by Das, Dill and Park [DDP99] to generate some invariants of the FLASH protocol (for example, that there are never two exclusive copies in the system). Invariants generated in this way were subsequently transferred to Park's proof of refinement in PVS.

It appears possible that invariants generated by predicate abstraction could be used to aid the proof presented here. For example, lemma *L3*, stating that an exclusive copy cannot coexist with another exclusive copy, write-back, *etc.*, appears to be the kind of invariant that was generated in [DDP99] (although few details are given). On the other hand, it appears that the manual effort required to arrive at the necessary predicates (a reported five days) was greater than the manual effort of the entire proof presented here, so it is not clear that using predicate abstraction to obtain the non-interference conditions would be worth the effort. On the other hand, the method of [DDP99] can in some cases guess necessary instantiations of universal quantifiers. There were two cases in the above proof in which such instantiations were introduced manually. Possibly by applying the methods of [DDP99], these instantiations could have been found automatically. That is an interesting possibility, since the abstractions generated by SMV can in fact be cast in terms of predicate abstraction. This points to a possible integration of the two methods.

Note that the predicate abstraction method by itself cannot prove liveness of the protocol. In fact, no single finite abstraction can do this, because a finite system, if it terminates, must terminate in a bounded number of steps. However, there is no fixed bound on the number of steps required for a transaction to terminate in the FLASH protocol. Rather, this number increases with N.

Verification of cache coherence protocols with an arbitrary number of processors has also been done using so-called *symbolic state* abstractions [PD93]. This is a finite-state approach to systems of many identical processes, in which information about the exact number of processes in each state is abstracted, so that only the cases 0, 1 and "more than one" are distinguished. Pong used this method to verify cache coherence protocols. However, the method required that the protocol be described in a specialized abstract form. This left open the problem of verifying that this model in fact is refined by the more operational description that one actually implements. This problem was solved by Ip [ID96], who showed how such a model could be extracted from a description in the Murϕ language [ID96], under certain restrictions. Not all protocols can be handled in this way. In particular, protocols that pass pointers to nodes can only be handled if pointer chains are bounded. However, in the cases when it does apply, the method is certainly more automated than the one presented here (it is unclear whether it could be applied effectively to FLASH). Note, however, that the abstraction it uses is too coarse to prove liveness, so other methods would have to be used for this purpose.

Finally, an advantage of using a system such as SMV for the verification of cache protocols is that SMV is also capable of verifying that the actual hardware implementation of the protocol conforms to the verified model. This kind of refinement verification was done, for example, in [Eir98], although in that case the protocol was only verified for a fixed number of nodes. A general purpose prover such as PVS is not well suited to this task. On the other hand, the Murϕ system provides no means to verify that a hardware system refines a Murϕ model (this, of course, suggests a possible link between the tools).

6 Conclusion

We have seen an example of formal verification of a cache coherence protocol model, for N processors, using a compositional model checking tool. The protocol was specified with respect to a very simple reference model. The proof that the protocol model implements this reference model was accomplished with four auxiliary lemmas. The first is a "refinement relation", specifying the contents of main memory in terms of the reference model. The other three are "non-interference" lemmas. These were required to rule out false counterexamples that occurred because some nodes in the system were abstracted away.

Our general approach was driven by counterexamples. That is, we attempt naïvely to prove a property, and when a counterexample arises, we diagnose the cause of the error. We then attempt to rule out this cause by one of two strategies. If the cause is reception of an impossible message from an abstracted node, we can "split cases" on the sender of this message, causing the sender to be included in the abstraction. This case splitting can only be done a few times, however, since it leads to state space explosion. The other possibility is to write a "non-interference" lemma, ruling out the bad message. This lemma must be proved separately. It is often useful to pose these lemmas as refinement relations, by which we recode the model state in a more convenient way. The lemmas are then proved by course-of-values induction over time. That is, we can assume all the lemmas up to $t - 1$ when proving a case of any given lemma at time t.

This approach appears to be the most effective one to date for verifying both safety and liveness of cache coherence protocols with an arbitrary number of processes. For example, the proof effort for the FLASH protocol using model checking was clearly much less than the effort for the same model using PVS. The method of predicate abstraction requires considerable user assistance to guess the necessary predicates and cannot prove liveness of the FLASH protocol. Symbolic state methods give a fully automated verification of safety. However, they do not apply to all protocols, and cannot prove liveness. Finally, having proved properties of a protocol model in SMV, it is then possible to refine that model to a formally verified implementation in hardware.

References

[CD93] C.N. Ip and D.L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100. Elsevier, 1993.

[DDP99] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification (CAV'99)*, pages 160–171, 1999.

[Eir98] A. Eiriksson. Formal design of 1M-gate ASICs. In *FMCAD '98*, number 1522 in LNCS, pages 49–63. Springer, 1998.

[ID96] C.N. Ip and D.L. Dill. Verifying systems with replicated components in murphi. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV'96)*, volume 1102, pages 147–158. Springer Verlag, 1996.

[KOH^{+}94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The stanford FLASH multiprocessor. In *Proc. of the 21th Annual Int'l Symp. on Comp. Arch. (ISCA'94)*, pages 302–313, 1994.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[McM99] K. L. McMillan. Verification of an infinite state systems by compositional model checking. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 219–233, 1999.

[MS91] K.L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In N. Suzuki, editor, *Proceedings of the International Symposium on Shared Memory Multiprocessors.* MIT Press, 1991.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In E. Kapur, editor, *Conf. on Automated Deduction (CADE'92)*, number 607 in LNCS. Springer-Verlag, 1992.

[PD93] F. Pong and M. Dubois. The verification of cache coherence protocols. In *Proc. of the 5th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'93)*, pages 11–20, 1993.

[PD96] S. Park and D.L. Dill. Verification of the FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, Padua, Italy, 1996.

[SG97] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254, pages 72–83, Haifa, Israel, 1997. Springer-Verlag.