A Simple Method for Parameterized Verification of Cache Coherence Protocols

Ching-Tsun Chou*, Phanindra K. Mannava, and Seungjoon Park

Intel Corporation
3600 Juliette Lane, SC12-322
Santa Clara, CA 95054, USA
ching-tsun.chou@intel.com

Abstract. We present a simple method for verifying the safety properties of cache coherence protocols with arbitrarily many nodes. Our presentation begins with two examples. The first example describes in intuitive terms how the German protocol with arbitrarily many nodes can be verified using a combination of Murphi model checking and apparently circular reasoning. The second example outlines a similar proof of the FLASH protocol. These are followed by a simple theory based on the classical notion of simulation proofs that justifies the apparently circular reasoning. We conclude the paper by discussing what remains to be done and by comparing our method with other approaches to the parameterized verification of cache coherence protocols, such as compositional model checking, machine-assisted theorem proving, predicate abstraction, invisible invariants, and cut-off theorems.

1 Introduction

The by-now standard method in industry for debugging a cache coherence protocol is to build a formal model of the protocol at the algorithmic level and then do an exhaustive reachability analysis of the model for a small configuration size (typically 3 or 4 nodes) using either explicit-state or symbolic model checking. While this method does offer a much higher degree of confidence in the correctness of the protocol than informal reasoning and simulation can, and protocol designers often have intuitions about why 3 or 4 nodes suffice to exercise all "interesting" scenarios, it is still very desirable to actually have a *proof* that the protocol model is correct for *any* number of nodes.

Proving a protocol correct for any number of nodes (or some other configuration parameters) is called *parameterized verification*. Unfortunately, parameterized verification is in general an undecidable problem [1]. While this result may not be directly applicable to a specific protocol or even a restricted class of protocols, it does suggest that parameterized verification of real-world protocols will likely require a certain amount of human intervention. So our goal here is to figure out how to minimize human intervention and maximize the work done by

* Main contact.

automatic tools (such as model checkers). Most importantly, we would like the automatic tools to extract the necessary information from the protocol that can apparently circular reasoning needed in our proofs. Section 5 discusses what remains to be done, in particular how the many tedious tasks performed by hand in the German and FLASH proofs can and should be mechanized. Section 6 compares our method with other approaches to the parameterized verification of cache coherence protocols.

2 Parameterized Verification of the German Protocol

The German protocol [7] is a simple cache coherence protocol devised by Steven German in 2000 as a challenge problem to the formal verification community[2]. Since then it has become a common example in papers on parameterized verification [2, 6, 9, 17]. The Murphi code of the German protocol we will use is shown in Figure 1 and should be self-explanatory. It is essentially the same as the one in [17] except that we have shortened identifier names and added data paths to make it more interesting. The state variable `AuxData` is an auxiliary variable for tracking the latest value of the cache line and does not affect the execution of the protocol in any way; its sole purpose is to allow us to state the property (`DataProp`) about the correct data values in memory and caches.

Now let us try to prove that the invariants `CtrlProp` and `DataProp` are true in the German protocol (abbreviated as German below) for an arbitrary number of caching nodes. The basic idea behind our method is as follows. Consider an instance of German with a large number of caching nodes. Choose any 2 of the caching nodes (the reason for the number 2 will become clear later) and observe the behaviors of them plus the home node (whose data structures are not indexed by `NODE`). Note that since all caching nodes are symmetric [8] with respect to one another, it does not matter which 2 nodes we choose. We will try to construct an abstract model AbsGerman containing the home node plus the 2 chosen nodes with the following properties:

P1. AbsGerman permits all possible behaviors that the home node plus the 2 chosen nodes can engage in, including what those nodes that are *not* chosen can do to them.

P2. The behaviors of AbsGerman are sufficiently constrained that interesting properties (including `CtrlProp` and `DataProp`) can be proved about them.

If we can achieve both P1 and P2, then we can deduce the truth of `CtrlProp` and `DataProp` in German from their truth in AbsGerman. But there is clearly a tension between P1 and P2 and it is not obvious how to meet them both. Our strategy is to start with an AbsGerman that obviously satisfies P1 but violates P2 and then refine AbsGerman over several steps until P2 is satisfied, while maintaining P1 all the time.

We begin with a naive abstraction of German shown in Figure 2 that is obtained by making the following changes to the model in Figure 1:

[2] German's challenge was to verify the protocol fully automatically, which is *not* our goal. But his protocol, being short, is a good medium for presenting our method.

http://www.doc88.com/p-079841781727.html

https://www.cnblogs.com/snow826520/p/8574824.html

```
const  ---- Configuration parameters ----

  NODE_NUM : 4;
  DATA_NUM : 2;

type   ---- Type declarations ----

  NODE : scalarset(NODE_NUM);
  DATA : scalarset(DATA_NUM);

  CACHE_STATE : enum {I, S, E};
  CACHE : record State : CACHE_STATE; Data : DATA; end;

  MSG_CMD : enum {Empty, ReqS, ReqE, Inv, InvAck, GntS, GntE};
  MSG : record Cmd : MSG_CMD; Data : DATA; end;

var   ---- State variables ----

  Cache : array [NODE] of CACHE;      -- Caches
  Chan1 : array [NODE] of MSG;        -- Channels for Req*
  Chan2 : array [NODE] of MSG;        -- Channels for Gnt* and Inv
  Chan3 : array [NODE] of MSG;        -- Channels for InvAck
  InvSet : array [NODE] of boolean;   -- Nodes to be invalidated
  ShrSet : array [NODE] of boolean;   -- Nodes having S or E copies
  ExGntd : boolean;                   -- E copy has been granted
  CurCmd : MSG_CMD;                   -- Current request command
  CurPtr : NODE;                      -- Current request node
  MemData : DATA;                     -- Memory data
  AuxData : DATA;                     -- Latest value of cache line

---- Initial states ----

ruleset d : DATA do startstate "Init"
  for i : NODE do
    Chan1[i].Cmd := Empty; Chan2[i].Cmd := Empty; Chan3[i].Cmd := Empty;
    Cache[i].State := I; InvSet[i] := false; ShrSet[i] := false;
  end;
  ExGntd := false; CurCmd := Empty; MemData := d; AuxData := d;
end end;

---- State transitions ----

ruleset i : NODE do rule "SendReqS"
  Chan1[i].Cmd = Empty & Cache[i].State = I
==>
  Chan1[i].Cmd := ReqS;
end end;

ruleset i : NODE do rule "SendReqE"
  Chan1[i].Cmd = Empty & (Cache[i].State = I | Cache[i].State = S)
==>
  Chan1[i].Cmd := ReqE;
end end;

ruleset i : NODE do rule "RecvReqS"
  CurCmd = Empty & Chan1[i].Cmd = ReqS
==>
  CurCmd := ReqS; CurPtr := i; Chan1[i].Cmd := Empty;
  for j : NODE do InvSet[j] := ShrSet[j] end;
end end;

ruleset i : NODE do rule "RecvReqE"
  CurCmd = Empty & Chan1[i].Cmd = ReqE
==>
  CurCmd := ReqE; CurPtr := i; Chan1[i].Cmd := Empty;
  for j : NODE do InvSet[j] := ShrSet[j] end;
end end;
```

```
ruleset i : NODE do rule "SendInv"
  Chan2[i].Cmd = Empty & InvSet[i] = true &
  ( CurCmd = ReqE | CurCmd = ReqS & ExGntd = true )
==>
  Chan2[i].Cmd := Inv; InvSet[i] := false;
end end;

ruleset i : NODE do rule "SendInvAck"
  Chan2[i].Cmd = Inv & Chan3[i].Cmd = Empty
==>
  Chan2[i].Cmd := Empty; Chan3[i].Cmd := InvAck;
  if (Cache[i].State = E) then Chan3[i].Data := Cache[i].Data end;
  Cache[i].State := I; undefine Cache[i].Data;
end end;

ruleset i : NODE do rule "RecvInvAck"
  Chan3[i].Cmd = InvAck & CurCmd != Empty
==>
  Chan3[i].Cmd := Empty; ShrSet[i] := false;
  if (ExGntd = true)
  then ExGntd := false; MemData := Chan3[i].Data; undefine Chan3[i].Data end;
end end;

ruleset i : NODE do rule "SendGntS"
  CurCmd = ReqS & CurPtr = i & Chan2[i].Cmd = Empty & ExGntd = false
==>
  Chan2[i].Cmd := GntS; Chan2[i].Data := MemData; ShrSet[i] := true;
  CurCmd := Empty; undefine CurPtr;
end end;

ruleset i : NODE do rule "SendGntE"
  CurCmd = ReqE & CurPtr = i & Chan2[i].Cmd = Empty & ExGntd = false &
  forall j : NODE do ShrSet[j] = false end
==>
  Chan2[i].Cmd := GntE; Chan2[i].Data := MemData; ShrSet[i] := true;
  ExGntd := true; CurCmd := Empty; undefine CurPtr;
end end;

ruleset i : NODE do rule "RecvGntS"
  Chan2[i].Cmd = GntS
==>
  Cache[i].State := S; Cache[i].Data := Chan2[i].Data;
  Chan2[i].Cmd := Empty; undefine Chan2[i].Data;
end end;

ruleset i : NODE do rule "RecvGntE"
  Chan2[i].Cmd = GntE
==>
  Cache[i].State := E; Cache[i].Data := Chan2[i].Data;
  Chan2[i].Cmd := Empty; undefine Chan2[i].Data;
end end;

ruleset i : NODE; d : DATA do rule "Store"
  Cache[i].State = E
==>
  Cache[i].Data := d; AuxData := d;
end end;

---- Invariant properties ----

invariant "CtrlProp"
  forall i : NODE do forall j : NODE do
    i != j -> (Cache[i].State = E -> Cache[j].State = I) &
              (Cache[i].State = S -> Cache[j].State = I | Cache[j].State = S)
  end end;

invariant "DataProp"
  ( ExGntd = false -> MemData = AuxData ) &
  forall i : NODE do Cache[i].State != I -> Cache[i].Data = AuxData end;
```

某node在share之前，先保证其他node都无效

Fig. 1. The German cache coherence protocol

1. Set NODE_NUM to 2, which has the effect of changing the type NODE to containing only the 2 nodes chosen for observation.
2. Add a new type declaration: "ABS_NODE : union {NODE, enum{Other}}", which contains the 2 chosen nodes plus a special value Other representing all those nodes that are *not* chosen.
3. If a state variable (including array entries) has type NODE, change it to ABS_NODE, because in the abstract model a node pointer can still point to a node that is not being observed (i.e., an Other). In GERMAN, there is only one variable whose type is so changed: "CurPtr : ABS_NODE".
4. But the occurrences of NODE as array index types are *not* changed, because we are observing only the nodes in NODE (plus the home node, which is not indexed) and have discarded the part of the state corresponding to the nodes represented by Other.

```
const
  NODE_NUM : 2

type
  NODE : scalarset(NODE_NUM);
  ABS_NODE : union {NODE, enum{Other}};

var
  CurPtr : ABS_NODE;

-- Include the original German protocol model
-- here, but with the above changes/additions.

rule "ABS_Skip" end;

rule "ABS_RecvReqS"
  CurCmd = Empty
==>
  CurCmd := ReqS; CurPtr := Other;
  for j : NODE do InvSet[j] := ShrSet[j] end;
end;

rule "ABS_RecvReqE"
  CurCmd = Empty
==>
  CurCmd := ReqE; CurPtr := Other;
  for j : NODE do InvSet[j] := ShrSet[j] end;
end;

rule "ABS_RecvInvAck"
  CurCmd != Empty & ExGntd = true
==>
  ExGntd := false; undefine MemData;
end;

rule "ABS_SendGntS"
  CurCmd = ReqS & CurPtr = Other & ExGntd = false
==>
  CurCmd := Empty; undefine CurPtr;
end;

rule "ABS_SendGntE"
  CurCmd = ReqE & CurPtr = Other & ExGntd = false &
  forall j : NODE do ShrSet[j] = false end
==>
  ExGntd := true; CurCmd := Empty; undefine CurPtr;
end;

ruleset d : DATA do rule "ABS_Store"
  true
==>
  AuxData := d;
end end;
```

Fig. 2. Abstract German protocol: First version

5. There is nothing to be done about abstracting the state initialization routine `Init`, since no node pointer is initialized to a specific node (`CurPtr` is initialized to undefined).
6. We now consider how to abstract the state transition rulesets, each of which has a node parameter `i`. There are two cases to consider:
   (a) When `i` is one of the 2 chosen nodes: This is taken care of by keeping a copy of the original ruleset, since now the type `NODE` contains precisely those 2 nodes. There is one subtlety, though: note that the precondition of ruleset `SendGntE` contains a universal quantification over `NODE`, which is weakened by the "shrinking" of `NODE` to only the 2 chosen nodes. But since the universal quantification occurs positively, this weakening only makes the rule more permissive, which is what we want (recall P1 above).
   (b) When `i` is an `Other`: We create an abstract version of each ruleset for this case, as shown in Figure 2. The goal is to satisfy P1 without making the abstract rulesets too permissive. Our method is summarized below:
      i. All occurrences of `i` in the original rulesets are replaced by `Other`.
      ii. All references to the part of the state indexed by `Other` that occur positively in the preconditions are replaced by `true`, since the abstract model does not track that part of the state and hence cannot know the truth values of such references. By substituting `true` for such references, the rules are made more permissive (indeed, often *too* permissive, which we will fix later).
      iii. Similarly, all changes to the part of the state indexed by `Other` are discarded, since they have no effects on the part of the state that the abstract model keeps. As a consequence, the rulesets `SendReq*`,

SendInv, SendInvAck, and RecvGnt* are all abstracted by a single no-op rule ABS_Skip. Furthermore, since the statement part of ruleset ABS_RecvInvAck now contains only a single "**if** c **then** s **end**", we move the condition c into the guard of the ruleset.

iv. If a part of the state indexed by Other is assigned to a state variable in the abstract model, we undefine that variable to represent the fact that the value being assigned is unknown. This happens to MemData in ruleset ABS_RecvInvAck.

v. The argument about the universal quantification in the precondition of the ruleset ABS_SendGntE is the same as before.

Clearly, the above abstraction steps are *conservative* in the sense that the AbsGerman thus obtained permit all possible behaviors of the home node plus any 2 caching nodes in German. It is, however, *too* conservative: if we model-check the abstract model in Figure 2, we will get a counterexample. In the rest of this section we explain how AbsGerman can be "fixed" to remove all counterexamples. But before we do that, let us comment out the property DataProp, because for cache coherence protocols it is generally a good idea to prove all control logic properties before working on any data path properties, as the latter depends on the former but not vice versa.

We now do the model checking, which produces the following counterexample to CtrlProp: node n_1 sends a ReqE to home; home receives the ReqE and sends a GntE to node n_1; node n_1 receives the GntE and changes its cache state to E; node n_2 sends a ReqS to home; home receives the ReqS and is about to send an Inv to node n_1; but suddenly home receives a bogus InvAck from Other (via ABS_RecvInvAck), which causes home to reset ExGntd and send a GntS to node n_2; node n_2 receives the GntS and changes its cache state to S, which violates CtrlProp because node n_1 is still in E. The bogus InvAck from Other is clearly where things start to go wrong: if there is a node in E, home should not receive InvAck from any other node. We can capture this desired property as a ***noninterference lemma***:

```
invariant "Lemma_1"
  forall i : NODE do
    Chan3[i].Cmd = InvAck & CurCmd != Empty & ExGntd = true ->
    forall j : NODE do
      j != i -> Cache[j].State != E & Chan2[j].Cmd != GntE
  end end;
```

which says that if home is ready to receive an InvAck from node i (note that the antecedent is simply the precondition of RecvInvAck plus the condition ExGntd = true, which is the only case when the InvAck is to have any effect in ABS_RecvInvAck), then every other node j must not have cache state E or a GntE in transit to it. (We are looking ahead a bit here: if the part about GntE is omitted from Lemma_1, the next counterexample will compel us to add it.) If Lemma_1 is indeed true in German, then we will be justified to ***refine*** the offending abstract ruleset ABS_RecvInvAck as follows:

```
rule "ABS_RecvInvAck"
  CurCmd != Empty & ExGntd = true &
  forall j : NODE do
    Cache[j].State != E & Chan2[j].Cmd != GntE
  end
==> ... end;
```

where we have strengthened the precondition by instantiating Lemma_1 with i = Other. (Note that since Other is distinct from any j in NODE in the abstract model, there is no need to test for the inequality.) Why is this strengthening justified? Because Lemma_1 says that when RecvInvAck with i = Other is enabled, the conjunct we add to the precondition of ABS_RecvInvAck is true anyway, so adding that conjunct does *not* make ABS_RecvInvAck any less permissive.

But how do we prove that Lemma_1 is true in German? The surprising answer is that we can prove it in the same abstract model where we have used it to refine one of the abstract ruleset! Is there any circularity in our argument? The answer is *no*, and we will develop a theory in Section 4 to justify this claim.

So we can refine ABS_RecvInvAck as shown above and add Lemma_1 as an additional invariant to prove in the abstract model. But this is not yet sufficient for removing all counterexamples. More noninterference lemmas and ruleset refinements are needed for that and the model checker will guide us to discovering them via the counterexamples. The final result of this process is shown in Figure 3, where the step numbers refer to the following sequence of steps:

Step 1: This is the discussion above.

Step 2: A rather long counterexample to Lemma_1 shows the following. Node n_1 acquires an E copy, which is invalidated by a ReqS from Other. But before the InvAck reaches home, home receives a bogus InvAck from Other, which makes home think that there is no E copy outstanding and hence sends GntS to Other. Now node n_2 sends ReqE to home, which receives the stale InvAck from node n_1 and sends GntE to node n_2. But the Inv that home sends to n_1 on behalf of n_2 is still in the network, which now reaches node n_1 and generates a InvAck. So now we have both a InvAck the8he4(fi5 TL x00388 01ia 1)]

```
-- Everything up to rule "ABS_SendGntE"
-- is exactly the same as in Figure 2.

rule "ABS_RecvInvAck"
  CurCmd != Empty & ExGntd = true &
  forall j : NODE do
    Cache[j].State != E &     -- Step 1
    Chan2[j].Cmd != GntE &    -- Step 1
    Chan3[j].Cmd != InvAck    -- Step 2
  end
==>
  ExGntd := false;
  MemData := AuxData;         -- Step 4
end;

ruleset d : DATA do rule "ABS_Store"
  ExGntd = true &             -- Step 3
  forall j : NODE do
    Cache[j].State = I &      -- Step 5
    Chan2[j].Cmd != GntS &    -- Step 5
    Chan2[j].Cmd != GntE &    -- Step 5
    Chan3[j].Cmd != InvAck    -- Step 6
  end
==>
  AuxData := d;
end end;
```

```
-- Noninterference lemmas:

invariant "Lemma_1"
  forall i : NODE do
    Chan3[i].Cmd = InvAck & CurCmd != Empty &
    ExGntd = true ->
    Chan3[i].Data = AuxData &             -- Step 4
    forall j : NODE do
      j != i -> Cache[j].State != E &     -- Step 1
                Chan2[j].Cmd != GntE &    -- Step 1
                Chan3[j].Cmd != InvAck    -- Step 2
  end end;

invariant "Lemma_2"
  forall i : NODE do
    Cache[i].State = E ->
    ExGntd = true &                       -- Step 3
    forall j : NODE do
      j != i -> Cache[j].State = I &      -- Step 5
                Chan2[j].Cmd != GntS &    -- Step 5
                Chan2[j].Cmd != GntE &    -- Step 5
                Chan3[j].Cmd != InvAck    -- Step 6
  end end;
```

Fig. 3. Abstract German protocol: Final version

Step 4: A short counterexample to the first clause of DataProp shows that Other acquires an E and then messes up MemData by writing back undefined data. The fix is to refine ABS_RecvInvAck using a strengthened Lemma_1 asserting that the written back data must be AuxData.

Step 5: A short counterexample to the second clause of DataProp shows that a node acquires an E copy and then suddenly Other does a store that changes AuxData. The fix is to refine ABS_Store using a strengthened Lemma_2 asserting that if any node i is in state E, then any other node j cannot be in E as well. Looking ahead, j should also be required not to be in S or about to become E or S, for similar counterexamples can arise without these additional properties. Again, the model checker will lead us to these additional requirements even if we have not thought of them.

Step 6: A counterexample to Lemma_1 shows the following. Node n_1 acquires an E copy, which is invalidated by a ReqE from Other. But before the InvAck reaches home, Other does a store that changes AuxData to violate its equality to the data carried by the InvAck (which is added to Lemma_1 in Step 4). The fix is to refine ABS_Store using a strengthened Lemma_2 asserting that if any node has cache state E, then any other node cannot have an InvAck in transit to home.

After Step 6, all counterexamples disappear. According to the theory developed in Section 4, this means that CtrlProp and DataProp (plus the noninterference lemmas) have been proved for German with arbitrarily many nodes.

Now we come to the question of why the abstract model is set up to have 2 nodes. This is because none of the universally quantified formulas in the desired properties (CtrlProp and DataProp), the noninterference lemmas (Lemma_1 and

`Lemma_2`), and the rulesets has more than 2 nested quantifiers over nodes. So a 2-node abstract model suffices to give a "maximally diverse" interpretation to the quantified formulas (i.e., an interpretation in which different quantified variables are not forced to take on the same values due to the small size of the universe of the interpretation).

A related question is why, unlikely McMillan in [10, 12], we have been able to dispense with a three-valued logic in the abstraction and reasoning process. The reason for this is that we have imposed the following syntactic constraints on our models and properties: (1) a state variable (or array entry) of type `NODE` is never compared with another state variable (or array entry) of the same type, and (2) an array over `NODE` is never indexed into directly by a state variable (or array entry). For instance, in the rulesets `SendGnt*` in Figure 1, instead of using `CurPtr` to index into arrays directly (as is done in [17]), we introduce a bound variable `i` ranging over `NODE`, test `CurPtr` = `i` in the precondition, and use `i` to index into arrays. Under these syntactic constraints, the abstraction process outlined above becomes feasible[4] and every logic formula is either true or false in the abstract model, where `Other` is a possible value of a node-valued state variable (or array entry). Our experience suggests that all practical cache coherence protocols can be modeled and their properties stated under these syntactic constraints. Also note that these constraints make it impossible for a formula to implicitly say that "there are K nodes" (e.g., by stating that K node-valued state variables are pairwise unequal) without a corresponding number of quantifiers over nodes[5].

3 Parameterized Verification of the FLASH Protocol

The Murphi code of FLASH we use is translated from McMillan's SMV code [12], which in turn is translated from Park's PVS code [16]. To be precise, this is a model of the "eager mode" of the FLASH protocol[6].

FLASH is much more complex and realistic than German. Their numbers of reachable states (after symmetry reduction) are an indication of this: German has 852, 5235, 28088 states and FLASH has 6336, 1083603, 67540392 states at 2, 3, 4 nodes, respectively. So, with brute-force model checking, FLASH is at best barely verifiable at 5 nodes and definitely not verifiable at 6 nodes. (SMV does not perform any better than Murphi on FLASH.) FLASH is a good test for any proposed method of parameterized verification: if the method works on FLASH, then there is a good chance that it will also work on many real-world cache coherence protocols.

[4] Existentially quantified formulas that occur positively in rule preconditions will still cause problems, but in practice they rarely occur and can always be replaced by auxiliary variables that supply explicit witnesses.

[5] We are grateful to Steven German for pointing out this issue to us.

[6] In the *eager mode* of FLASH, the home is allowed to grant an exclusive copy before all shared copies have been invalidated. In contrast, in the *delayed mode* of FLASH, the home must invalidate all shared copies before granting an exclusive copy.

```
invariant "Lemma_1"
  forall h : NODE do forall i : NODE do
    h = Home & Proc[i].CacheState = CACHE_E ->
    Dir.Dirty & WbMsg.Cmd != WB_Wb & ShWbMsg.Cmd != SHWB_ShWb & UniMsg[h].Cmd != UNI_Put &
    forall j : NODE do UniMsg[j].Cmd != UNI_PutX end &
    forall j : NODE do j != i -> Proc[j].CacheState != CACHE_E end
  end end;

invariant "Lemma_2"
  forall h : NODE do forall i : NODE do forall j : NODE do
    h = Home & i != j & j != h & UniMsg[i].Cmd = UNI_Get & UniMsg[i].Proc = j ->
    Dir.Pending & !Dir.Local & PendReqSrc = i & FwdCmd = UNI_Get
  end end end;

invariant "Lemma_3"
  forall h : NODE do forall i : NODE do forall j : NODE do
    h = Home & i != j & j != h & UniMsg[i].Cmd = UNI_GetX & UniMsg[i].Proc = j ->
    Dir.Pending & !Dir.Local & PendReqSrc = i & FwdCmd = UNI_GetX
  end end end;

invariant "Lemma_4"
  forall h : NODE do forall i : NODE do
    h = Home & i != h & InvMsg[i].Cmd = INV_InvAck ->
    Dir.Pending & Collecting & NakcMsg.Cmd = NAKC_None & ShWbMsg.Cmd = SHWB_None &
    forall j : NODE do
      ( UniMsg[j].Cmd = UNI_Get | UniMsg[j].Cmd = UNI_GetX -> UniMsg[j].Proc = h ) &
      ( UniMsg[j].Cmd = UNI_PutX -> UniMsg[j].Proc = h & PendReqSrc = j )
  end end end;

invariant "Lemma_5"
  forall i : NODE do Proc[i].CacheState = CACHE_E -> Proc[i].CacheData = CurrData end;
```

Fig. 4. Noninterference lemmas for FLASH

We have done a proof of the safety properties of FLASH for any number of nodes (which is available upon request) using the same method as described in Section 2. Due to space limitations, we cannot give full details here. Below we only list the main differences between this proof and the proof of German:

1. The number of nodes in the abstract model is 3 (instead of 2). For, in FLASH, the request processing flow is such that it is convenient to make the home node data structures also indexed by `NODE`. This has the effect of making some noninterference lemmas contain 3 nested quantifers over nodes.
2. In FLASH there are node-indexed arrays whose entries are node-valued, which is a type of data structures that German does not have. In the abstract model those node-valued array entries must be allowed to have the value `Other`, just like node-valued state variables.
3. In FLASH a ruleset may have up to 2 node parameters. A typical example is a node n_1 with the exclusive copy receiving a forwarded request from the home, in which case n_1 sends the copy directly to the requesting node n_2 without going through the home. To abstract such a ruleset, we have to consider four cases: when n_1 and n_2 are both in the abstract model, when n_1 is in but n_2 is `Other`, when n_2 is in but n_1 is `Other`, and when n_1 and n_2 are both `Other`. So a single ruleset in FLASH may be split into up to four rulesets in the abstract model.

Despite these differences and the complexity of FLASH, we find that we need to introduce only five noninterference lemmas (shown in Figure 4) to get the proof to work for both control logic and data path properties. As can be seen, the conjunction of these lemmas fall far short of an inductive invariant. Perhaps more importantly, the total amount of efforts required for the proof is modest: 1 day to translate the SMV code into Murphi code, 0.5 day to flush out translation errors using conventional model checking (up to 4 nodes), 0.5 day to manually abstract the model, and 1 day to iteratively find the noninterference lemmas from counterexamples and to finish the proof. One interesting observation is that the abstract FLASH model has 21411411 reachable states (after symmetry reduction), so its complexity is roughly between 3-node and 4-node FLASH, which makes perfect sense.

4 A Theory Justifying Apparently Circular Reasoning

When the proof in Section 2 or 3 is completed, a small (2- or 3-node) abstract model has been constructed and the model checker has proved several invariants (desired properties and noninterference lemmas) about the abstract model. Why are we then justified in concluding that the desired properties are in fact true for the original parameterized model with any number of nodes? We have made some informal arguments, but they appear alarmingly circular. In particular, why is it sound to prove the noninterference lemmas in the abstract model which have been argued to be more permissive than the original model *using the very same lemmas*? In this section we first develop a theory based on the classical notion of simulation proofs [15] that justifies such apparently circular reasoning, and then shows how it is applied in the German and FLASH proofs.

4.1 Simulation Proofs

We will use standard set-theoretic notations. For any function $f \in A \to B$ and $C \subseteq A$ and $D \subseteq B$, the ***image*** of C under f is $f(C) = \{f(a) \in B : a \in C\}$ and the ***inverse image*** of D under f is $f^{-1}(D) = \{a \in A : f(a) \in D\}$. A useful fact to know is that $f(C) \subseteq D \Leftrightarrow C \subseteq f^{-1}(D) \Leftrightarrow \forall a \in C : f(a) \in D$. We also generalize f to operate on $A \times A$: $f(a, a') = (f(a), f(a'))$. Let V be a set of indices. If B is the (cartesian) ***product*** of an indexed family of sets, $B = \prod_{v \in V} B_v$, then f naturally induces a family of functions, $f_v \in A \to B_v$ for $v \in V$, such that $f(a) = \langle f_v(a) : v \in V \rangle$.

We will model protocols and their abstractions as state transition systems. Formally, a ***state transition system*** (STS) $M = (S, I, T)$ consists of a set S of states, a set $I \subseteq S$ of initial states, and a transition relation $T \subseteq S \times S$. An ***execution*** $(s_0, s_1, \ldots)$ of M is a finite or infinite sequence of states of M such that $s_0 \in I$ and $(s_i, s_{i+1}) \in T$ for all $i \geq 0$. A state s of M is ***reachable*** iff s is the last state of a finite execution of M; the set of reachable states of M is denoted by $\mathcal{R}(M)$. For an indexed family of STSs, $M_v = (S_v, I_v, T_v)$ for $v \in V$, the ***product*** STS is $\prod_{v \in V} M_v = (\prod_{v \in V} S_v, \prod_{v \in V} I_v, \prod_{v \in V} T_v)$. Clearly, we have $\mathcal{R}(\prod_{v \in V} M_v) = \prod_{v \in V} \mathcal{R}(M_v)$.

A set of states $P \subseteq S$ is an ***invariant*** of M iff $\mathcal{R}(M) \subseteq P$. A set of states $Q \subseteq S$ is ***inductive*** in M iff $\forall s \in I : s \in Q$ and $\forall (s, s') \in T : s \in Q \Rightarrow s' \in Q$. Clearly, an inductive set of states of M is always an invariant of M, and the set of reachable states of M, $\mathcal{R}(M)$, is the strongest invariant of M and is always inductive. All safety properties of M can be reduced to invariant properties provided that a sufficient amount of history information is recorded in the state, which can always be achieved by adding auxiliary variables.

Milner [15] introduced the notion of *simulation.* The following definition is not the most general possible, but rather is tailored to our needs. Let $M = (S, I, T)$ be a *concrete* STS and $\widetilde{M} = (\widetilde{S}, \widetilde{I}, \widetilde{T})$ an *abstract* STS.

Definition 1. *A* **simulation** (P, f) *from* M *to* $\widetilde{M}$ *consists of an* **inductive invariant** $P \subseteq S$ *and an* **abstraction function** $f \in S \rightarrow \widetilde{S}$ *such that:*

(1) $\quad \forall s \in I : s \in P \wedge f(s) \in \widetilde{I}$

(2) $\quad \forall (s, s') \in T : s \in P \Rightarrow s' \in P \wedge f(s, s') \in \widetilde{T}$

The notion of simulation is useful because it allows one to infer invariant properties of the concrete system from those of the abstract system.

Theorem 1. *If* (P, f) *is a simulation from* M *to* $\widetilde{M}$*, then:*

(3) $\quad \forall s \in \mathcal{R}(M) : s \in P \wedge f(s) \in \mathcal{R}(\widetilde{M})$

Proof. By induction over the lengths of executions of M. □

Formula (3) says that each reachable state s of M not only satisfies the inductive invariant P, but also inherits all invariant properties of $\widetilde{M}$ via f^{-1}.

In practice, the main difficulty in using simulation to infer properties of the concrete system from those of the abstract system lies in coming up with a suitable inductive invariant, which is very hard for any nontrivial system. But, fortunately, the following theorem says that there is at least one invariant that always works:

Theorem 2. *For any function* $f \in S \rightarrow \widetilde{S}$*, if:*

(4) $\quad \forall s \in I : f(s) \in \widetilde{I}$

(5) $\quad \forall (s, s') \in T : f(s) \in \mathcal{R}(\widetilde{M}) \Rightarrow f(s, s') \in \widetilde{T}$

then $(f^{-1}(\mathcal{R}(\widetilde{M})), f)$ *is a simulation from* M *to* $\widetilde{M}$ *and:*

(6) $\quad \forall s \in \mathcal{R}(M) : f(s) \in \mathcal{R}(\widetilde{M})$

Proof. Let $P = f^{-1}(\mathcal{R}(\widetilde{M}))$. Since $\mathcal{R}(\widetilde{M})$ is inductive in $\widetilde{M}$, (4) and (5) imply (1) and (2), respectively. So (P, f) is a simulation from M to $\widetilde{M}$. Furthermore, (6) and (3) are equivalent in this case. □

Theorem 2 is the ultimate source of apparent circularity in our proof method, in the following sense. On the one hand, (6) says that the invariant property that

M inherits from $\widetilde{M}$ is $f^{-1}(\mathcal{R}(\widetilde{M}))$. On the other hand, (5) says that $f^{-1}(\mathcal{R}(\widetilde{M}))$ can also be used as an *assumption* in the inductive step of the simulation proof.

For reasoning about a parameterized concrete system, the abstract system we will use is a product of many small systems, each of which captures a partial *view* of the concrete system. What views are will become clear shortly; here we just want to point out that a view is *not* a node in a cache coherence protocol.

Theorem 3. *Suppose the abstract system is a product STS,* $\widetilde{M} = \prod_{v \in V} \widetilde{M}_v$, *where* $\widetilde{M}_v = (\widetilde{S}_v, \widetilde{I}_v, \widetilde{T}_v)$ *for* $v \in V$. *If for each* $v \in V$:

(7) $\forall s \in I : f_v(s) \in \widetilde{I}_v$

(8) $\forall (s, s') \in T : (\forall u \in V : f_u(s) \in \mathcal{R}(\widetilde{M}_u)\,) \Rightarrow f_v(s, s') \in \widetilde{T}_v$

then $(f^{-1}(\mathcal{R}(\widetilde{M})), f)$ *is a simulation from* M *to* $\widetilde{M}$ *and:*

(9) $\forall s \in \mathcal{R}(M) : (\forall v \in V : f_v(s) \in \mathcal{R}(\widetilde{M}_v)\,)$

Proof. Theorem 3 is simply a re-statement of Theorem 2 using the following facts: $\widetilde{I} = \prod_{v \in V} \widetilde{I}_v$, $\widetilde{T} = \prod_{v \in V} \widetilde{T}_v$, and $\mathcal{R}(\widetilde{M}) = \prod_{v \in V} \mathcal{R}(\widetilde{M}_v)$. □

Theorem 3 enables one to break a simulation proof into small subproofs, one for each view v as represented by the abstract system $\widetilde{M}_v$ (see the antecedent of the theorem). Furthermore, (8) says that the conjunction of *all* inherited invariants can be used as an inductive hypothesis in *every* subproof.

4.2 Applying the Theory

We now show how Theorem 3 is used. Let M be German or FLASH with a large number of nodes.

First, note that the states of M are valuations of a finite number of ***state variables*** each of which is in one of the following forms:

- A boolean variable, `x` : B.
- A (node) pointer variable, `y` : N.
- An array of booleans, `z` : `array` $[N]$ `of` B.
- An array of (node) pointers, `w` : `array` $[N]$ `of` N.

where N is the set of nodes (or rather, node names) and B is the set of booleans. (There is no loss of generality in considering only booleans because enumerated types can be encoded using booleans.)

Second, there is a fixed m such that any subset of m nodes determines a ***view*** $v = \{n_1, \ldots, n_m\}$ of M. In other words, V is the set of m-element subsets of N. (For example, $m = 2$ for German and $m = 3$ for FLASH[7].) For each $v \in V$, the abstraction function f_v retains all boolean and pointer variables, discards all array entries except those indexed by a node in v, and sets any pointer-valued

[7] There is a slight complication here: in the case of FLASH, one of the 3 nodes in a view must be the home node.

variable or array entry to a special value `Other` $\notin N$ if its value is $\notin v$. More precisely, we define f_v as follows:

$$\begin{cases} f_v(s)(\mathtt{x}) = s(\mathtt{x}) & \text{for } \mathtt{x} : B \\ f_v(s)(\mathtt{y}) = s(\mathtt{y}) \downarrow v & \text{for } \mathtt{y} : N \\ f_v(s)(\mathtt{z}) = \lambda i \in v : s(\mathtt{z})(i) & \text{for } \mathtt{z} : \mathtt{array}\ [N]\ \mathtt{of}\ B \\ f_v(s)(\mathtt{w}) = \lambda i \in v : s(\mathtt{w})(i) \downarrow v & \text{for } \mathtt{w} : \mathtt{array}\ [N]\ \mathtt{of}\ N \end{cases}$$

where $j \downarrow v =$ **if** $j \in v$ **then** j **else** `Other`.

Third, since all nodes are symmetric with respect to each other in M (i.e., the set N is a *scalarset* [8]), we can take all $\widetilde{M_v}$'s to be isomorphic copies of the same abstract system $\widetilde{M_r}$ ("r" for "representative"). For instance, for German, we can take $\widetilde{M_r}$ to be the STS corresponding to AbsGerman (i.e., Figure 3). For each $v \in V$, $\widetilde{M_v}$ is obtained from $\widetilde{M_r}$ by renaming the nodes using any 1-1 mapping from r to v, where r also denotes the set of (non-`Other`) nodes in $\widetilde{M_r}$.

Now let us see what Theorem 3 says, given the above. Its conclusion (9) says that the property $P = \forall v \in V : f_v^{-1}(\mathcal{R}(\widetilde{M_v}))$ is an invariant of M. But what does P say? P is true of a state s of M iff for any $v \in V$, any invariant of $\widetilde{M_v}$ is true of s when projected via f_v^{-1} onto the view v (remember that the set of reachable states is also the strongest invariant). For example, the property `CtrlProp` has been proved (by model checking) to be an invariant of AbsGerman and hence also an invariant of any isomorphic copy $\widetilde{M_v}$ of AbsGerman. Since `CtrlProp` contains 2 node quantifiers and $\widetilde{M_v}$ contains 2 nodes, (9) allows us to conclude that `CtrlProp` is an invariant of M. The same reasoning applies to all desired properties and noninterference lemmas in the German and FLASH proofs. Note again that the reasoning depends on the fact that there are at least as many nodes in $\widetilde{M_v}$ as there are nested node quantifiers in the invariant.

But, in order to invoke the conclusion (9) of Theorem 3, we must discharge its antecedents (7) and (8) for each view $v \in V$. Since $\widetilde{M_v}$ is a renamed copy of $\widetilde{M_r}$ and all nodes are symmetric in M, there is no loss of generality in considering only the case when $v = r$. We will discuss only (8), since (7) is similar but simpler. Consider any step (s, s') of M. Note that the transition relation T can be decomposed as follows:

$$T = \bigcup_{r_1 \in R_1} \bigcup_{i \in N} T_i^{r_1} \cup \bigcup_{r_2 \in R_2} \bigcup_{(i,j) \in N \times N} T_{i,j}^{r_2}$$

where each $T_i^{r_1}$ (respectively, $T_{i,j}^{r_2}$) corresponds to an instance of a ruleset with name r_1 (r_2) and node parameters i (i and j). So the proof of (8) entails a case split into which ruleset instance, $T_i^{r_1}$ or $T_{i,j}^{r_2}$, the step (s, s') belongs to. The former case is split further into subcases $i \in r$ or $i \notin r$; the latter into subcases ($i \in r$ and $j \in r$) or ($i \notin r$ and $j \in r$) or ($i \in r$ and $j \notin r$) or ($i \notin r$ and $j \notin r$). We will do one subcase for German as an example of the apparently circular reasoning; all other cases in the German and FLASH proofs are similar.

Consider the ruleset `RecvInvAck` in German when $\mathtt{i} \notin r$. Suppose it fires. If we can prove that the state change it effects in M is (via f_r) permitted by ABS_RecvInvAck or ABS_Skip in $\widetilde{M}_r$, then we have discharged (8) for this subcase. Since RecvInvAck fires, its precondition:

```
Chan3[i].Cmd = InvAck & CurCmd != Empty
```

must be true in the current state s. Now comes the crucial point: (8) allows us to assume that the aforementioned property $P = \forall v \in V : f_v^{-1}(\mathcal{R}(\widetilde{M}_v))$ is true at s. So we can project the noninterference lemma Lemma_1 on i and any node j in r. There are two further cases to consider: if ExGntd is true, the projected Lemma_1 implies that the state change is permitted by ABS_RecvInvAck (in particular, the precondition of ABS_RecvInvAck is true); otherwise, if ExGntd is false, RecvInvAck has no effect whatever after f_r (because i $\notin r$) and hence is trivially permitted by ABS_Skip. QED.

5 What Remains to Be Done

The first priority is clearly *mechanization*. We have carried out by hand the reasoning steps in Section 4.2 (i.e., the discharging of (7) and (8) and the application of (9)). Though they are quite simple, it would be much better if they are checked by a theorem prover. Another task that should be completely automatable is the construction of the initial abstract models as described in Sections 2 and 3. Such abstraction is very tedious and may allow errors to creep in when the protocol description is long. Ideally, we want to formalize not only the reasoning steps in Section 4.2 but also the theory developed in Section 4.1 in a theorem prover, so that we can have a completely formal proof.

It is also desirable to be able to reason about liveness, which we cannot do now. We have put some thoughts into this and believe that it is doable, but of course the devil will be in the details. Since Theorem 3 is quite general and does not depend on any intrinsic property of the index set V, it should be possible to use it to reason about parameterized systems where the parameter sets are not scalarsets but have additional structures (such as successor and ordering) [14].

6 Comparison with Other Works

This paper owes most of its intellectual debts to McMillan's work on compositional model checking [10] and its application to FLASH [12]. The abstractions we used, the reliance on apparently circular reasoning, and the counterexample-guided discovery of noninterference lemmas are all deeply influenced by McMillan's work. His framework is also more general than ours by encompassing liveness properties [11], though we believe that our framework can be generalized to handle liveness as well. Relative to his work, we think we make two main contributions. First, we show that practical parameterized verification can be done using any model checker (not just Cadence SMV) plus some simple reasoning. The freedom to choose model checkers is important in practice, as experience shows that for cache coherence protocols explicit-state model checkers are often superior to symbolic model checkers. Second, we develop a simple theory based on the classical notion of simulation proofs to justify the apparently circular reasoning. We believe this de-mystifies compositional model checking and opens the way to formalizing the theory and its application in a theorem prover.

Park and Dill [16] proved the safety properties of FLASH using machine-assisted theorem proving in PVS. Their proof is also based on the notion of simulation proofs, but uses the formulation of simulation in Definition 1, which requires an inductive invariant. Not surprisingly, they spent a significant amount (perhaps most) of their efforts on formulating and proving the inductive invariant. In contrast, the conjunction of the noninterference lemmas in Figure 4 falls far short of an inductive invariant. Also, it took them roughly two weeks to come up with the inductive invariant and to do the proof, which is a lot longer than the one day we spent.

Predicate abstraction has been used to verify German (without data paths) by Baukus, Lakhnech, and Stahl [2] and FLASH by Das, Dill, and Park [3]; the former also handles liveness. There are two main problems to be solved when applying predicate abstraction to parameterized verification: how to discover a suitable set of predicates, and how to map a finite set of predicates onto an unbounded set of state variables. To solve the second problem, the above two papers use complex predicates containing quantifiers, some of which are almost as complex as an invariant. This makes the discovery of such predicates non-obvious and probably as hard as the formulation of noninterference lemmas. More recently, a conceptual breakthrough was made by Lahiri and Bryant [9], who developed a theory of and the associated symbolic algorithms for *indexed predicates*, where the indices are implicitly universally quantified over. They used their techniques to verify a version of German with unbounded FIFO queues. We believe that there are close connections between their work and this paper, which we want to explore in the future.

Pnueli, Ruah, and Zuck [17] proposed an automatic (though incomplete) technique for parameterized verification called *invisible invariants*, which uses a small instance with N_0 nodes to generate an inductive invariant that works for instances of any size, where the bound N_0 depends on the forms of protocol and property descriptions. For German (without data paths), $N_0 = 4$. Although their technique is very attractive for being automatic, there are reasons to believe that it would not work for FLASH. First, their theory does not seem to allow the protocol to use node pointer arrays indexed by nodes, which FLASH has. Second, even if the theory can be made to work, the bound N_0 for FLASH is likely to be much greater than 4. Given the remarks in Section 3, this makes it very doubtful that FLASH can be verified using their method. We believe that the large bound results from the automatic nature of their method, which forces them to use general arguments that depend only on the form of the protocol and property descriptions. In our framework, human insights about specific protocols can limit the number of nodes needed by means of noninterference lemmas.

Emerson and Kahlon [6] verified German (without data paths) by first reducing it to a snoopy bus protocol and then invoking a theorem of theirs asserting that if a snoopy bus protocol of a certain form is correct for 7 nodes then it is correct for any number of nodes. Unfortunately, no such cut-off results are known for protocols as complex as FLASH (or, for that matter, for German directly), nor is it clear how FLASH can be reduced to protocols for which cut-off results are known.

Acknowledgements

We are grateful to Steven German for his comments, which greatly helped to improve this paper.

References

1. Apt, K.R., Kozen, D.: Limits for automatic program verification of finite-state concurrent systems. *Information Processing Letters* (1986) 22(6):307–309.
2. Baukus, K., Lakhnech, Y., Stahl, K.: Parameterized verification of a cache coherence protocol: safety and liveness. VMCAI (2002) 317–330.
3. Das, S., Dill, D.L., Park, S.: Experience with predicate abstract. CAV (1999) 160–171.
4. Della Penna, G., Intrigila, B., Tronci, E., Zilli, M.V.: Exploiting transition locality in the disk based Murphi verifier. FMCAD (2002) 202–219.
5. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors (1992) 522-525.
6. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. CHARME (2003) 247–262.
7. German, S.M.: Personal communications (2000).
8. Ip, C.N., Dill, D.L.: Better verification through symmetry. CHDL (1993) 87–100.
9. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. VMCAI (2004) 267–281.
10. McMillan, K.L.: Verification of infinite state systems by compositional model checking. CHARME (1999) 219–234.
11. McMillan, K.L.: Circular compositional reasoning about liveness. CHARME (1999) 342–345.
12. McMillan, K.L.: Parameterized verification of FLASH cache coherence protocol by compositional model checking. CHARME (2001) 179–195.
13. McMillan, K.L.: Exploiting SAT solvers in unbounded model checking. CAV tutorial (2003) (`http://www-cad.eecs.berkeley.edu/~kenmcmil/cav03tut.ppt`).
14. McMillan, K.L., Qadeer, S., Saxe, J.B.: Induction in compositional model checking. CAV (2000) 312–327.
15. Milner, R.: An algebraic definition of simulation between programs. IJCAI (1971) 481–489.
16. Park, S., Dill, D.L.: Verification of the FLASH cache coherence protocol by aggregation of distributed transactions. SPAA (1996) 288–296.
17. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. TACAS (2001) 82–97.