Reducing Verification Complexity of a Multicore Coherence Protocol Using Assume/Guarantee

Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan
School of Computing, University of Utah
{xiachen, yuyang, ganesh}@cs.utah.edu

Ching-Tsun Chou
Intel Corporation
ching-tsun.chou@intel.com

***Abstract*— We illustrate how to employ metacircular assume/guarantee reasoning to reduce the verification complexity of finite instances of protocols for safety, using nothing more than an explicit state model checker. The formal underpinnings of our method are based on establishing a simulation relation between the given protocol M, and several overapproximations thereof, $\tilde{M}_1, \ldots, \tilde{M}_k$. Each $\tilde{M}_i$ simulates M, and represents one "view" of it. The $\tilde{M}_i$s depend on each other both to define the abstractions as well as to justify them. We show that in case of our hierarchical coherence protocol, its designer could easily construct each of the $\tilde{M}_i$ in a counterexample guided manner. This approach is practical, considerably reduces the verification complexity, and has been successfully applied to a complex hierarchical multicore cache coherence protocol which could not be verified through traditional model checking.**

I. Introduction

The dream of parameterized infinite-state verification of protocols for assertions written in expressive property languages such as CTL* are commonly held, and even demonstrated on actual protocols. However, highly complex industrial cache coherence protocols are the other extreme: they are so complex that one would be lucky to model check even small instances consisting of only a few caching agents for safety properties. With the imminence of multicore chips, the situation is expected to become worse, as multiple clusters of caching agents will interact through a second level of protocols, which will also be complex. The combined state space of protocols at various caching levels is astronomical. While one may attempt to separately verify each level and somehow "glue" together the results, discovering suitable formal arguments supporting the gluing step can, in fact, be an equally complex task. This paper addresses the following problems, making the indicated contributions:

• *There is no public domain hierarchical cache coherence protocol of reasonable complexity to employ as a verification benchmark.* In response to this problem, we developed a complex hierarchical protocol under the supervision of an industrial expert to ensure that our assumptions are realistic. Our protocol adapts the FLASH [1] protocol with MESI [2] features to manage caching within a cluster (consisting of two cache agents and the associated local directory), and adapts the DASH [3] protocol also with MESI features to manage coherence among three clusters. Figure 1 shows our verification model. It consists of a home cluster, h, and two identical remote clusters, r_1 and r_2, namely $M = (h \parallel r_1 \parallel r_2)$. Written in Murphi [4], the model [5] has about $3,000$ lines of code, and has the kind of complex scenarios that industrial protocols being designed today have.

This work was supported in part by SRC Contract 2005-TJ-1318.

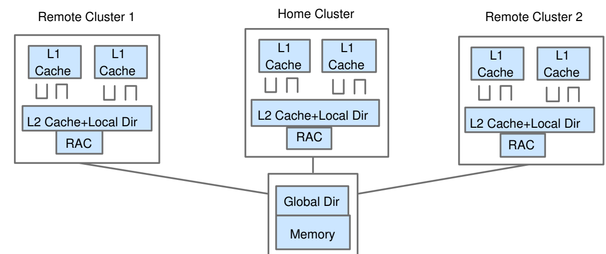

Fig. 1. A 2-level hierarchical cache coherence protocol.

• After removing many shallow bugs from M through non-exhaustive safety model checking, we failed to show the set of cache coherence properties ϕ_{coh} on M, due to state explosion, after $161,876,000$ of states. Our main contribution in this paper is a novel solution to this verification problem, adapting many ideas from a recently proposed assume/guarantee (A/G) approach. This A/G approach was first proposed in [6], but employed for *parameterized* verification in that work. Our method works as follows.

1. A set of abstracted protocols, $\tilde{M}_1, \ldots, \tilde{M}_k$, are constructed from M by simply projecting out (unconstraining) selected global variables, and correspondingly overapproximating the protocol rules.[1] Different variables are projected out for each M_i, and therefore, each $\tilde{M}_i$ presents a different overapproximated view of M.
2. Upon model checking each $\tilde{M}_i$ with respect to ϕ_{coh} and a set of non-interference lemmas (initially an empty set), there are three outcomes possible: *the verification succeeds*: M can then be claimed to satisfy ϕ_{coh}; *a genuine counterexample is generated:* the designer corrects M and iterates; *a counterexample infeasible in M is generated:* the designer strengthens some of the guards in $\tilde{M}_i$, and also adds a corresponding *verification obligation* (sometimes called *noninterference lemma*) to one of the $\tilde{M}_1, \ldots, \tilde{M}_k$, and the whole process is repeated.

Applying this procedure to our example protocol M, the above process generates three abstracted protocols, $\tilde{M}_1$, $\tilde{M}_2$ and $\tilde{M}_3$ ($\tilde{M}_2$ and $\tilde{M}_3$ are the same as explained in Section III). A genuine bug was found in M, and corrected. Thereafter,

[1] Murphi is a rule-based system, in which a transition relation is represented as: $rule$ "$name$" $guard \rightarrow action$;.

10 iterations were applied to both $\tilde{M}_1$ and $\tilde{M}_2$ to eliminate infeasible counterexamples. In the end, model checking was able to verify $\tilde{M}_1$ and $\tilde{M}_2$ with the same resources, over three and seven hours respectively[2].

- The refinement process discussed for $\tilde{M}_1$ and $\tilde{M}_2$ only requires modest manual effort, with each guard strengthening condition corresponding to a reasonably natural designer insight. Also, all noninterference lemmas can be obtained from the strengthening conditions, and hence the soundness of our method is assured (formally proved in Section IV).
- Our technique can be employed using *any* safety model checker.

The rest of the paper is organized as follows. Section II presents an overview and some features of the 2-level MESI protocol. Section III presents the generation of the $\tilde{M}_i$s, and Section IV describes the construction of the noninterference lemmas and guard-strengthening. Related work and concluding remarks follow.

II. Benchmark Hierarchical Coherence Protocols

Our benchmark hierarchical coherence protocol is derived by combining features from the FLASH and DASH protocols. Such a protocol is realistic, as DASH was originally developed for managing coherence across many clusters. It also renders our hierarchical protocol easy to understand for researchers who might want to attack this verification challenge problem. One address is modeled in our protocol, as is typical in model-checking based verification for coherence. As shown in Figure 1, the protocol is composed of three NUMA (Non-Uniform Memory Access) clusters: one home cluster and two identical remote clusters. Each cluster has two symmetric L1 caches, an L2 cache and a local directory. “RAC” is the communication controller with other clusters and the global directory. The main memory in reality is attached to every cluster. The fact there is only one memory is a consequence of the 1-address abstraction of our protocol. Finally, the global directory is to manage data copies on the three clusters.

In the 2-level hierarchy, the level-1 protocol is used within a cluster, i.e. the two L1 caches and the L2 cache in Figure 1. It tracks which line is cached in what state at which agent(s). The FLASH protocol is adapted to model this level and keep data copies within a cluster consistent. The level-2 protocol is used among clusters, tracking caching status in cluster level. The DASH protocol is adapted to keep clusters consistent. Also,

- For each cache line, if it is cached in an agent then it must also be cached in the local directory of the agent, i.e. the *inclusive* property;
- Both levels use MESI, supporting explicit write-back and silent-drop[3]
- Both levels use non-FIFO network ordering (this, according to our industrial expert, is a desirable feature of modern protocols).

In the process of developing our hierarchical protocol, we discovered that it was not simply a matter of adapting FLASH and DASH to support MESI states and silent-drops, as such combinations can easily lead to livelocks. One such scenario is the following. Agent-1 of cluster-1 first requests an exclusive copy and gets granted. As a result, a subsequent request from the rest of the system to the same line will be forwarded to agent-1. At this time, if agent-1 has silently dropped the cache line, it will NACK the forwarded request. Because the local directory of agent-1 has no information about the silent-drop happening in agent-1, and it is not safe to use the data copy in the local directory to reply (e.g. write-back from agent-1 is on the way), cluster-1 will keep forwarding following requests to agent-1 and agent-1 will keep NACKing. This results in a livelock. We solved this livelock problem by making write-back a blocking operation and adding another NACKing message indicating silent-drops, as detailed in [5].

```
ProcState: record
   -- B1 agents in the cluster
   Proc :   array [NODE] of NODE_STATE;

   -- B2 network channels used in the cluster
   UniMsg:  array [NODE_L2] of UNI_MSG;
   InvMsg:  array [NODE_L2] of INV_MSG;
   WbMsg:   WB_MSG;
   ShWbMsg: SHWB_MSG;
   NakcMsg: NAKC_MSG;

   --  local dir for the cluster
   L2: record
       -- B3.1 used only by level-1 protocol
       pending:    boolean;
       ShrSet:      array [NODE] of boolean ;
       InvCnt:     CacheCnt;
       HeadPtr:    NODE_L2;
       ReqId:      NODE;
       ReqCluster: Procss;
       ReqType:    boolean;
       isRetired:  boolean;
       ifHoldMsg:  boolean;
       -- B3.2 used by both levels
       State:     L2State;
       Data:      Datas;
       Dirty:     boolean;
       OnlyCopy:  boolean;
       Gblock_WB: boolean;
   end;

   -- B4 comm. controller with other clusters and global dir
   RAC: record
       State:    RACState;
       InvCnt:  ClusterCnt;
   end;
end;
```

```
ABSProcState: record
    L2: record
         -- B3.2
        State:     L2State;
        Data:      Datas;
        Dirty:     boolean;
        OnlyCopy:  boolean;
        Gblock_WB: boolean;
    end;

     -- B4
    RAC: record
        State:    RACState;
        InvCnt:  ClusterCnt;
    end;
end;
```

Fig. 2. The concrete and abstract structures in Murphi, representing a cluster in the hierarchical protocol.

[2] These large run-times are due to the relatively complex nature of the protocol compared to many traditional academic benchmarks. Also, no hash compaction was used, as the in-house 64-bit version of Murphi we employed has not been tested under hash compaction.

[3] Silent-drop in MESI protocols means a cache may discard a non-Modified line at any time, changing to the Invalid state without informing the local or global directory.

Our hierarchical coherence protocol coded in Murphi includes all the control logic and data coherence invariants that FLASH and DASH have. In Figure 2, we present the data structure representing a cluster in the hierarchical protocol, the record “ProcState”. It contains five blocks of information: (**B1**) number “NODE” of agents in a cluster, (**B2**) a set of network channels used inside a cluster, (**B3.1**) & (**B3.2**) the local directory, and (**B4**) a controller to communicate with other clusters and the global directory. In these five blocks, the level-1 protocol (used within a cluster) only involves the first four blocks, i.e. blocks $B1$, $B2$, $B3.1$ and $B3.2$, and the level-2 protocol (used among clusters) only involves blocks $B3.2$ and $B4$. We also present the data structure of an abstracted cluster in Figure 2, the record “ABSProcsState”, as it will be used quite often in the rest of the paper. An abstracted cluster only contains blocks $B3.2$ and $B4$.

III. Building abstracted protocols

Our hierarchical protocol proved to be very complex such that after $161,876,000$ states, the model checking failed due to state explosion. This is not surprising, considering the multiplicative effect of having three instances of coherence protocols running concurrently. We believe that all hierarchical (e.g., multicore) coherence protocols will state explode in this manner. This section describes how our initial abstracted models $\tilde{M}_1$, $\tilde{M}_2$ and $\tilde{M}_3$ were obtained. Due to the symmetry between the two remote clusters, $\tilde{M}_2$ is actually the same with $\tilde{M}_3$. So only $\tilde{M}_1$ and $\tilde{M}_2$ will be discussed in the rest of the paper.

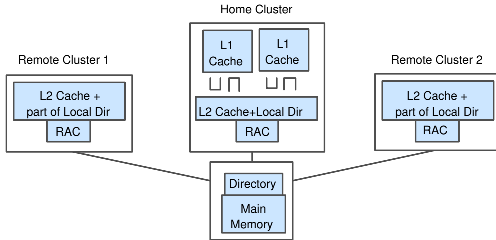

Fig. 3. The abstracted protocol $\tilde{M}_1$

A. Building the abstract models

Figure 3 intuitively depicts the abstracted protocol $\tilde{M}_1$, which retains the home cluster intact, while abstracting the two remote clusters (contrast with Figure 1). The abstracted protocol $\tilde{M}_2$ is very similar to $\tilde{M}_1$, except that it retains one remote cluster concretely and abstracts the other remote cluster as well as the home cluster. The derivation of $\tilde{M}_1$ and $\tilde{M}_2$ from M involves the variable abstraction and the corresponding transition relation and invariant abstraction described next. Specifically, in M, “Home” is the singleton set containing the home cluster and “Rmt” is the set containing the remote clusters [4]. “Procss” is their union, and the clusters are declared as “Procs:”

```
HomeCnt: 1;
Home:    scalarset(HomeCnt);
RmtCnt:  2;
Rmt:     scalarset(RmtCnt);
Procss:  union {Home, Rmt};

Procs: array [Procss] of ProcState
```

[4]We employ scalarsets to obtain the benefit of symmetry reduction.

Now, let us consider the state declarations of clusters in $\tilde{M}_1$ (first half of Figure 4) and $\tilde{M}_2$ (second half of Figure 4). In $\tilde{M}_1$, the home cluster will still be declared as “ProcState,” while the remote clusters will be declared as “ABSProcState,” the data structure only containing part of the local directory and the communication controller of a cluster. $\tilde{M}_2$ is similar.

```
-- clusters in  M1
HomeCnt: 1;
RmtCnt:  2;
Home:    scalarset(HomeCnt);
Rmt:     scalarset(RmtCnt);
Procss:  union {Home, Rmt};

Procs:    array [Home] of ProcState;
ABSProcs: array [Rmt] of ABSProcState;
```

```
-- clusters in  M2
HomeCnt:  1;
RmtCnt_1: 1;
RmtCnt_2: 1;
Home:     scalarset(HomeCnt);
Rmt_1:    scalarset(RmtCnt_1);
Rmt_2:    scalarset(RmtCnt_2);

Procs:   array [Rmt_1] of ProcState;
ABSHome: array [Home] of ABSProcState;
ABSRmt:  array [Rmt_2] of ABSProcState;
```

Fig. 4. Declaration of clusters in $\tilde{M}_1$ and $\tilde{M}_2$

For each transition (*TR*) in M, a *set* of corresponding TRs are constructed in $\tilde{M}_1$ and $\tilde{M}_2$. These could be automatically generated, and we have described the procedure for $\tilde{M}_1$ in the appendix (the procedure for $\tilde{M}_2$ is similar). We consider the rules one at a time in M. For every rule “$guard \rightarrow action$”, for any assignment of the form `v := E` in $action$, (i) if `v` is a variable that has been eliminated (abstracted away), then the whole assignment is eliminated, (ii) else if `E` contains even one variable that has been abstracted away, we replace `E` with a non-deterministic selection over the type of `E`. If a sub-expression in $guard$ contains any variable that has been abstracted away, the sub-expression turns into $true$.

IV. Verifying the hierarchical protocol

This section presents details of the refinement process and the soundness of our approach.

A. Counterexamples, lemmas, and guard-strengthening

Figure 5 shows the process of how the refinement is applied on $\tilde{M}_1$, $\tilde{M}_2$ and M (if it is buggy): When $\tilde{M}_1$ is model checked using Murphi with respect to ϕ_{coh}, if a genuine bug is detected, the designer corrects and reiterates. Assume that the error is a false alarm, and involves $rule_p$ “$g_p \rightarrow a_p$” of $\tilde{M}_1$. The corresponding rule in M is then located; assume it is “$G_p \rightarrow A_p$”. Then, based on the expression G_p, we *manually* derive another expression i_p which only contains the variables used

in the level-2 protocol, i.e. variables in blocks $B3.2$ and $B4$ in a cluster, the global directory, and the set of network channels used among clusters. We will discuss how such an expression i_p can be derived from G_p in a detailed example in the next section.

Now we add a *new verification obligation* (noninterference lemma) “$G_p \Rightarrow i_p$” to $\tilde{M}_2$, and at the same time strengthen $rule_p$ in $\tilde{M}_1$ to be “$g_p \wedge i_p \rightarrow a_p$”.

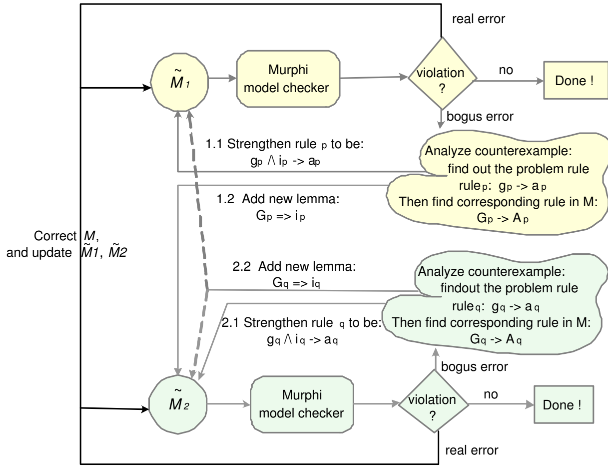

Fig. 5. Counterexample-guided metacircular refinement.

It is perhaps surprising that the noninterference lemma is added as a verification obligation in $\tilde{M}_2$, while the consequent of this lemma, i_p, is used in $\tilde{M}_1$. This is because according to the abstraction procedure in Section III, we know that the bogus error in $\tilde{M}_1$ is introduced by the overapproximation on the remote clusters. In particular, G_p contains some remote cluster details, however they are abstracted away in g_p. So the noninterference lemma $G_p \Rightarrow i_p$ can only be checked in $\tilde{M}_2$ where all the state elements present in G_p are available. A similar process can be applied while refining $\tilde{M}_2$, as also shown in Figure 5. Essentially, for each noninterference lemma L, it suffices to prove L in any of the abstracted models. The question of in which abstracted model to prove L is determined by which abstracted model has enough details for L to be proved.

B. A detailed example of refinement

One counterexample encountered in $\tilde{M}_1$ is as follows: after the startstate rule is fired (e.g. the state is just initialized), Rule “L2_Recv_NAKC_Nakc” in $\tilde{M}_1$ is fired and the first statement in the action – the assertion – is violated. This rule is shown in the second half of Figure 6, and the corresponding rule in M is also shown in the first half of the figure.

Consider the condition under which this rule can be enabled in the hierarchical protocol M. Figure 7 shows an example scenario: The remote cluster “p” initially holds an exclusive copy in the agent “src,” another remote cluster “aux” requesting for an exclusive copy (step 1 and 2) is forwarded by the global directory to the remote cluster “p” (step 3), and because the local directory of “p” indicates that the agent “src” holds the exclusive copy, “p” will forward the request to “src” (step

```
ruleset p: Rmt; src: NODE do
rule "L2_Recv_NAKC_Nakc"
    Procs[p].NakcMsg.Cmd = NAKC_Nakc &
    Procs[p].NakcMsg.Aux = L2 &
    Procs[p].L2.ReqType = false
==>
var aux: Rmt;
begin
    assert (Procs[p].L2.pending = true);
    Procs[p].L2.pending := false;

    assert(!isundefined(Procs[p].L2.ReqCluster));
    aux := Procs[p].L2.ReqCluster;

    assert (Procs[p].L2.ifHoldMsg = true);
    Procs[p].L2.ifHoldMsg := false;
    assert (GUniMsg[aux].Cmd = RDX_RAC &
            GUniMsg[aux].Cluster = p);
    undefine GUniMsg[aux];
    GUniMsg[aux].Cmd := NAK;

    GNakcMsg.Cmd := GNAKC_Nakc;
    GNakcMsg.Cluster := p;
    undefine Procs[p].L2.ReqCluster;
    undefine Procs[p].L2.ReqType;

    undefine Procs[p].NakcMsg;
    Procs[p].NakcMsg.Cmd := NAKC_None;
endrule;
endruleset;
```

Rule “L2_Recv_NAKC_Nakc” in M

```
ruleset p: Rmt; aux: Rmt do
rule "L2_Recv_NAKC_Nakc"
    true
==>
begin
    assert (GUniMsg[aux].Cmd = RDX_RAC&
            GUniMsg[aux].Cluster = p);
    undefine GUniMsg[aux];
    GUniMsg[aux].Cmd := NAK;

    GNakcMsg.Cmd := GNAKC_Nakc;
    GNakcMsg.Cluster := p;
endrule;
endruleset;
```

Rule “L2_Recv_NAKC_Nakc” in $\tilde{M}_1$

Fig. 6. Rule “L2_Recv_NAKC_Nakc” in M and $\tilde{M}_1$.

4.1); concurrently, “src” silently drops the copy (step 4.2). So in receiving the forwarded request from the local directory, “src” NACKs the request (step 5).

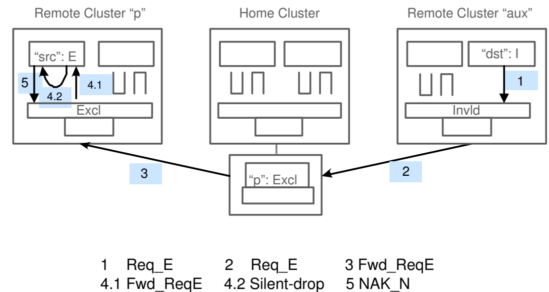

Fig. 7. A scenario which can enable “L2_Recv_NAKC_Nakc” in M.

Obviously, we can see that the violation in $\tilde{M}_1$ is a bogus error. The reason is because the guard condition of Rule “L2_Recv_NAKC_Nakc” is abstracted to “$true$”, which is overly approximated compared with that in M. Figure 8 shows the solution we used to overcome this violation: a new lemma “Lemma-7-A2” is added to $\tilde{M}_2$, and the consequent of this lemma is used to strengthen the guard of

"L2_Recv_NAKC_Nakc" in $\tilde{M}_1$.

```
ruleset p: Rmt; aux: Procss do
rule "L2_Recv_NAKC_Nakc"
    true &
    -- Lemma-7-A2
    ABSProcs[p].L2.State = Excl &
    ABSProcs[p].RAC.State = Inval &
    ABSProcs[p].L2.Gblock_WB = false &
    GUniMsg[aux].Cmd = RDX_RAC &
    GUniMsg[aux].Cluster = p
==>
begin
    ...
end;
```

Guard strengthening in $\tilde{M}_1$ using lemma-7-A2

```
invariant "Lemma-7-A2"
forall p: Rmt do
    Procs[p].NakcMsg.Cmd = NAKC_Nakc &
    Procs[p].NakcMsg.Aux = L2 &
    Procs[p].L2.ReqType = false
->
    Procs[p].L2.State = Excl &
    Procs[p].RAC.State = Inval &
    Procs[p].L2.Gblock_WB = false &
    exists aux: Procss do
      GUniMsg[aux].Cmd = RDX_RAC &
      GUniMsg[aux].Cluster = p  end
end;
```

New invariant added to $\tilde{M}_2$

Fig. 8. New "Lemma-7-A2" of $\tilde{M}_2$ and guard-strengthening in $\tilde{M}_1$.

Intuitively, the newly added lemma tries to mimic the level-2 protocol state from the available level-1 protocol state. In the example of Figure 8, "Lemma-7-A2" says when the local directory of a remote cluster receives "NAKC_Nakc" and this negative reply is for a forwarded request from another remote cluster, i.e. "Procs[p].NakcMsg.Aux = L2", then part of the level-2 state must be as follows – the remote cluster "p" is in Exclusive state, its communication controller is free, it is not blocked on write-back, and there exists another remote cluster "aux" requesting an exclusive copy.

C. Refinement results

In the refinement process, we found a real bug in M. The scenario corresponding to the real bug in M is similar to that in Figure 7. Initially, a cluster called p holds an exclusive copy in the agent src; $(S1)$ p receives a forwarded request from another cluster aux, for an exclusive copy; $(S2)$ p then forwards the request to src, the owner agent of p. $(S3)$ Concurrently, src updates the cache line and $(S4)$ writes back the dirty line to p; $(S5)$ therefore, upon receiving the forwarded request, src will NACK it. $(S6)$ When p receives the write-back data and then $(S7)$ receives the NACKed reply from src, p will $(S8.1)$ reply positively to aux with a data, also $(S8.2)$ send a message to the global directory notifying that aux is now the exclusive owner.

In $(S8.1)$ of the above scenario, the cluster p should indicate whether the data supplied to aux is dirty or not, because this data is not sent to the global directory in $(S8.2)$. If there is no such indication, then aux, when it receives the data, can set the state of the data to Exclusive. This permits aux to later silently drop the cache line. This, in effect, throws away the only copy of the (most recently modified) data, violating the coherence protocol.

We eliminated this bug in M by attaching a dirty tag to the reply message in $(S8.1)$. Correspondingly, $\tilde{M}_1$ and $\tilde{M}_2$ are updated for this modification. After that, we also added 10 new noninterference lemmas to $\tilde{M}_1$ and $\tilde{M}_2$ individually.

The following describes the 10 new noninterference lemmas[5] added to $\tilde{M}_1$ and $\tilde{M}_2$ after the above bug was corrected.

Step 1: In $\tilde{M}_1$, a short counterexample is encountered, which violates the assertion that when a cluster receives a write-back from an agent, the cluster must be in Exclusive state. This violation is introduced in $\tilde{M}_1$ because the guard condition of receiving write-back in remote clusters is abstracted to "$true$". Correspondingly, two similar counterexamples are encountered in $\tilde{M}_2$, with the same violation on the abstracted home cluster and the abstracted remote cluster. One lemma is added to $\tilde{M}_2$, and its consequent is used in $\tilde{M}_1$ and $\tilde{M}_2$. Another lemma is added to $\tilde{M}_1$, but used in $\tilde{M}_2$, both asserting that when a cluster receives a write-back request, the cluster must be Exclusive.

Steps 2,3,6,7: Similar to step 7, which is already discussed in Section IV-B.

Step 4: Three short counterexamples are encountered in $\tilde{M}_1$ and $\tilde{M}_2$, violating an invariant that when a cluster is dirty or Exclusive, other clusters must be Invalid for the same cache line. The fix asserts that when a cluster receives a shared write-back request from an agent, then that agent must be Exclusive.

Step 5: Three counterexample are encountered in $\tilde{M}_1$ and $\tilde{M}_2$, violating the assertion of an impossible branch in the hierarchical protocol M ("assert false"). The fix involves asserting that: if there is a state s in which (i) a cluster receives a NACK reply indicating the silent dropping of a line by an agent, *and* (ii) when such NACK arrives, the exclusive owner pointer is pointing neither at the local directory nor at the agent sending the NACK, then s is impossible.

Step 8: Three rather long counterexamples are encountered in $\tilde{M}_1$ and $\tilde{M}_2$, violating the invariant that when a cluster is waiting for invalidation acknowledgments and another cluster is Shared for the line, then there must exist an invalidation request being sent to the second cluster. A similar fix is applied, asserting that when the owner pointer of a cluster is the local directory, the cluster must be either Exclusive or Shared.

Step 9: Three counterexamples are encountered in $\tilde{M}_1$ and $\tilde{M}_2$, violating the invariant that when the global directory is not Exclusive, the main memory data should be the same with the current value of the cache line in the system. A similar fix is applied, asserting that when a cluster receives a shared request from a second cluster, the first cluster must be Exclusive.

Step 10: Three counterexamples are encountered, violating the assertion that when an exclusive request is granted, the reply should also contain the data. A similar fix as in Step 9 is applied.

After Step 10, all counterexamples disappear. Model checking on $\tilde{M}_1$ results in $31,919,219$ states using three hours, and

[5]These 10 refinement steps are listed only to give the reader a detailed glimpse at the nature of the errors and their strengthening conditions. The details are not important to follow.

$78,689,678$ states for $\tilde{M}_2$ using seven hours, both with the same resources.

D. A theorem justifying metacircular reasoning

The intuitive reason for the soundness of this approach is the following inductive argument. The verification obligations such as $G_p \Rightarrow i_p$ that are added to each of the abstract models $\tilde{M}_i$ are, of course, checked at the initial state. This implies that when G_p is true (the corresponding rule is enabled), i_p is implied. In other words, we *never* be underapproximating the state space if i_p is used to strengthen G_p. Since the valuations of the variables in $\tilde{M}_1$ and $\tilde{M}_2$ are the same as that in M, strengthening an *abstracted version* of G_p with i_p (which is what we do in our refinement method) is also sound. Now, since this check is done at every step, we ensure that the entire reachable state space is an overapproximation.

More formally, in [6], a theory was developed based on the classical notion of simulation proofs [7]. This theory was used to justify metacircular reasoning in *parameterized* verification of cache coherence protocols. However, the theory itself does not depend on any parameterized verification features. We summarize the main theorem of [6] as follows (we retain the equation and theorem numbers used in that paper):

Theorem 3. Suppose M is a state transition system (STS): $M = (S, I, T)$, where S is the set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the set of transition relations, and $\mathcal{R}(M)$ is the set of reachable states of M. Also suppose V is a set of "views" on M, $\{f_v : v \in V\}$ is a set of functions that create each view abstraction over the states in M, and $\{\tilde{M}_v : v \in V\}$ is the corresponding set of abstracted STS. Let f (respectively f^{-1}) be obtained by lifting f_v (respectively f_v^{-1}) over tuples of abstract states. If there exists an abstract system which is a product STS, $\tilde{M} = \Pi_{v \in V} \tilde{M}_v$, where $\tilde{M}_v = (\tilde{S}_v, \tilde{I}_v, \tilde{T}_v)$ for $v \in V$, then if for each $v \in V$:

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

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

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

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

While space does not permit a detailed explanation, we can actually explain these results quite well using our hierarchical protocol M. It is easy to see that each state s in M can be represented as a state vector

$$s = [\langle h.1, h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle]$$

where $\langle h.1, h.2\rangle$, $\langle r1.1, r1.2\rangle$ and $\langle r2.1, r2.2\rangle$ are the state components over the home cluster, remote-1 and remote-2 clusters. Moreover, $h.1$, $r1.1$, $r2.1$ correspond to the blocks $B1$, $B2$ and $B3.1$ in $ProcState$ of Figure 2, and $h.2$, $r1.2$, $r2.2$ corresponds to the blocks $B3.2$ and $B4$ in $ProcState$. gs is the rest of information in s, including the global directory and the network channels used among clusters. Now we discuss *three* abstractions in this section, $\tilde{M}_1, \tilde{M}_2$ and $\tilde{M}_3$, as was the case in our discussions in Section III. We can have the following theorem and equations in our framework. All numberings now carry a prime.

Theorem 3'. If f_1, f_2, f_3 are three abstract functions s.t.

(7')

$\forall s \in I$ and
$s = [\langle h.1, h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle]$

$\Rightarrow$

$f_i(s) \in \tilde{I}_i,\ i \in [1..3]$, where
$f_1(s) = [\langle h.1, h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle]$
$f_2(s) = [\langle h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle]$
$f_3(s) = [\langle h.2\rangle, \langle r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle]$

(8')

$\forall (s, s') \in T : f_i(s) \in \mathcal{R}(\tilde{M}_i),\ i \in [1..3]$ and
$s = [\langle h.1, h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle]$
$s' = [\langle h.1', h.2'\rangle, \langle r1.1', r1.2'\rangle, \langle r2.1', r2.2'\rangle, \langle gs'\rangle]$

$\Rightarrow$

$f_i(s, s') \in \tilde{T}_i,\ i \in [1..3]$, where
$f_1(s') = [\langle h.1', h.2'\rangle, \langle r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle]$
$f_2(s') = [\langle h.2'\rangle, \langle r1.1', r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle]$
$f_3(s') = [\langle h.2'\rangle, \langle r1.2'\rangle, \langle r2.1', r2.2'\rangle, \langle gs'\rangle]$

then all the invariants in $\tilde{M}_1$, $\tilde{M}_2$ and $\tilde{M}_3$ are valid in M, including the coherence properties. □

The fact that *Theorem 3'* is indeed a theorem follows from the fact that the antecedents of *Theorem 3'* are stronger than the antecedents used in *Theorem 3*.

1) Applying the theorem: Now we prove that the abstracted protocols $\tilde{M}_1, \tilde{M}_2$ and $\tilde{M}_3$ ($\tilde{M}_3$ is the same with $\tilde{M}_2$ for our hierarchical protocol M) indeed satisfy the conditions (7') and (8'). This will then allow us to conclude that the cache coherence properties in the abstracted protocols also hold in the original hierarchical protocol. We will, therefore, focus only on (8'), as the proof of satisfaction of (7') is much simpler.

Proof:

From a simple induction, we have

(9)'

$\forall s \in \mathcal{R}(M)$,
$s = [\langle h.1, h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle]$

$\Rightarrow$

$f_1(s) = [\langle h.1, h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle]$
$f_2(s) = [\langle h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle]$
$f_3(s) = [\langle h.2\rangle, \langle r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle]$

Now, consider any state transition (s, s') in the hierarchical protocol M. Assume that s' is obtained by firing a rule in M, "$R : g \rightarrow a$". From Section III-A, we know there exists one rule corresponding to R in each of $\tilde{M}_1$, $\tilde{M}_2$, and $\tilde{M}_3$. These are: in $\tilde{M}_1$ "$R1 :\ g1 \rightarrow a1''$; in $\tilde{M}_2$ $''R2 :\ g2 \rightarrow a2''$, and in $\tilde{M}_3$ $''R3 :\ g3 \rightarrow a3''$.

Also, the guard expression g of R can belong to only one of the following four cases. This is because any level-1 protocol details can only be visible to the level-2 protocol in the *same* cluster:

1. g only involves variables in $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$
2. g only involves $\langle h.1, h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$
3. g only involves $\langle h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$
4. g only involves $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle$

We now analyze each case to see that (8') is indeed satisfied.

Case 1. g only involves $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$ (i.e. does not refer to any "inner" state components).

From Section III-A, we know that $g1 = g2 = g3 = g$, also because

$f_1(s), f_2(s), f_3(s)$ and s hold the same values over variables in $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$

So $f_1(s), f_2(s), f_3(s)$ are enabled at $R1, R2, R3$ individually; from Section III-A we also know that the action ai's in rule Ri's, have the exact updates over variables in $f_i(s),\ i \in [1..3]$, as the action of the hierarchical protocol, a in R does. So

- $a1(f_1(s)) = [\langle h.1', h.2'\rangle, \langle r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle]$
  $= f_1(s')$
- $a2(f_2(s)) = [\langle h.2'\rangle, \langle r1.1', r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle]$
  $= f_2(s')$
- $a3(f_3(s)) = [\langle h.2'\rangle, \langle r1.2'\rangle, \langle r2.1', r2.2'\rangle, \langle gs'\rangle]$
  $= f_3(s')$

Case 2. g only involves $\langle h.1, h.2\rangle$,$\langle r1.2\rangle$, $\langle r2.2\rangle$,$\langle gs\rangle$ (i.e. g refers to some "inner" state components of the home cluster)

On one side, for $R1$ in $\tilde{M}_1$, based on Section III-A,

- $g1 = g$
- $a1$, the action of $R1$ in $\tilde{M}_1$, has the exact updates over the variables in $\langle h.1, h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$ as a, the action of R in M does;

So $f_1(s)$ is enabled at $R1$, and

- $a1(f_1(s)) = [\langle h.1', h.2'\rangle, \langle r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle]$
  $= f_1(s')$

On the other side, for $R2$ in $\tilde{M}_2$, based on Section III-A,

$g2$ only involves variables in $\langle h.2\rangle$,$\langle r1.2\rangle$, $\langle r2.2\rangle$,$\langle gs\rangle$, and all the sub-expressions in $g2$ involving any of these variables are exactly the same as the sub-expressions in g, so $g \Rightarrow g2$; or a new lemma is added and guard-strengthening is applied such that

* $g2 = g2_0 \wedge G_{uard}S_{strength}$
* We do know that $g \Rightarrow g2_0$
* $G_{uard}S_{strength}$ only involves variables in $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$
* $g \Rightarrow G_{uard}S_{strength}$ (this follows from the noninterference lemma being established)

Please note that the above reasoning does not depend on whether the new lemma was added into $\tilde{M}_1$, $\tilde{M}_2$ or $\tilde{M}_3$. The only property that matters is that for each variable in $\langle h.1, h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle$, it has the same value (if exists) in $s, f_1(s), f_2(s)$ and $f_3(s)$, which has already been stated in (9').

Again, because the variables in $f_2(s)$ have the same values over $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$ with s, so

* $f_2(s) \Rightarrow g2_0$
* $f_2(s) \Rightarrow G_{uard}S_{strength}$

That is, $f_2(s)$ is enabled at $R2$. Based on Section III-A, we know that the action $a2$ in rule $R2$ of $\tilde{M}_2$, has the exact updates over state components $\langle h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$ as a, the action of R in M does. So

- $a2(f_2(s)) = [\langle h.2'\rangle, \langle r1.1', r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle]$
  $= f_2(s')$

Similarly, the result holds for f_3: $a3(f_3(s)) = f_3(s')$.

Case 3& 4. These two cases can be proved similarly. □

The above proof justifies that the coherence properties in the abstracted protocols $\tilde{M}_1$, $\tilde{M}_2$ and $\tilde{M}_3$ also hold in the hierarchical protocol M. Moreover, according to Section III-A, we know that every invariant in M is covered by invariants in $\tilde{M}_1$ and $\tilde{M}_2$. So, it follows that once $\tilde{M}_1$ and $\tilde{M}_2$ are model checked to be coherent, the hierarchical protocol is also coherent.[6]

V. Related Work

This paper owes most of its intellectual debts to the work in [6] on verifying parameterized cache coherence protocols, and McMillan's work on compositional model checking [8]. The abstractions we used, the reliance on circular reasoning, and the counterexample-guided discovery of noninterference lemmas are all deeply influenced by their work. In addition to all our contributions pointed out earlier, we have two methodological contributions: Firstly, we find a way to naturally abstract a 2-level hierarchical cache coherence protocol into a few far simpler protocols. Secondly, the refinement process on the abstracted protocols is straightforward to conduct, including the noninterference lemmas. Finally, based on a theorem in [6], we have formally verified that our abstraction method is sound and complete.

McMillan [9] also modeled a 2-level MSI coherence protocol, based on the Gigamax distributed multiprocessor. In his protocol, bus-snooping is used in both levels. SMV [9] was used to check two clusters each having six processors with safety and liveness properties. Compared to our hierarchical protocol, the Gigamax model is much simpler.

Lahiri and Bryant in [10] use predicate abstraction to automatically construct quantified invariants. Their abstraction is similar with ours: for each concrete state, the abstraction maps it into multiple abstract states each of which corresponds to subranges of a set of universally quantified variables. Overapproximation ensures that the soundness of properties in the abstracted system guarantee the soundness in the concrete system.

VI. Conclusions and Future Work

Hierarchical cache coherence protocol verification is a challenging problem, as these protocols have many more protocol corner cases than non-hierarchical protocols, and have too many reachable states. In this paper, we propose a method to abstract a 2-level hierarchical coherence protocol into a few far simpler and tractable protocols. By verifying these simpler protocols, the correctness of the hierarchical protocol follows through the refinement theorem, also presented in this paper. Our success with the complex 2-level MESI hierarchical protocol developed in consultation with an industrial coherence protocol designer leads us to believe that other hierarchical coherence protocols – including snoopy, directory-based, ring interconnect based, as well as token-coherence based protocols – can be similarly verified.z Protocols with more than two

[6]The process of refinement will, of course, terminate because in the extreme case, we may end up describing the original protocol M through the noninterference lemmas! This extreme is not expected in practice.

levels (again common in large shared memory machines) also appear entirely amenable to our approach.

We plan to mechanize our method as much as possible. After the designer picks the variables to be projected out, constructing the initial abstract protocols could be automated. Identifying the noninterference lemmas will, in general, force the designer to understand their protocol - albeit in a localized manner in response to the counterexamples. It would also be interesting to exploit the fact that the two abstract protocols have a common subset of transitions, e.g. the level-1 protocol which only involves the agents and the local directory within a cluster. Extending our approach for non-inclusive cache coherence protocols is also in progress.

Acknowledgments We would thank Ritwik Bhattacharya, Igor Melatti and Liqun Cheng for their help on this work.

REFERENCES

[1] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. G. J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The stanford flash multiprocessor," in *ISCA*, 1994.

[2] M. Papamarcos and J. Patel, "A low overhead coherence solution for multiprocessors with private cache memories," in *ISCA*, 1984.

[3] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *ISCA*, 1990.

[4] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[5] Http://www.cs.utah.edu/formal_verification/fmcad06models.tar.gz.

[6] C.-T. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *FMCAD*, 2004.

[7] R. Milner, "An algebraic definition of simulation between programs," in *IJCAI*, 1971.

[8] K. McMillan, "Verification of infinite state systems by compositional model checking," in *CHARME*, 1999.

[9] K. L. McMillan, "Symbolic model checking," Ph.D. dissertation, Carnegie Mellon University, 1992.

[10] S. K. Lahiri and R. E. Bryant, "Constructing quantified invariants via predicate abstraction," in *VMCAI*, 2004.

APPENDIX
PROCEDURE FOR ABSTRACTING THE TRANSITION RELATION

We only present the procedure to construct the transition relations for $\tilde{M}_1$ in the following, as the procedure for $\tilde{M}_2$ is very similar. Our procedure is described in the context of Murphi, but the ideas are broadly applicable. We denote the set of variables which are abstracted away in $\tilde{M}_1$ by D.

1) Pre-processing:
   a) If there exists a $ruleset$ parameter whose range is over all the clusters in M, i.e. "Procss", divide the rule into two rules with the same guard and action: one rule with the parameter only over the home cluster ("Home"), and the other with the parameter over the remote clusters ("Rmt").
   b) If there exists any conditional statements, i.e. ifstmt or switchstmt, in the action of a rule, divide the rule into several sub-rules such that each sub-rule covers one branch of the conditional statement and the sub-rule does not contain conditional statements anymore. The guard of the sub-rule is the logic "$\wedge$" of the original rule guard and the conditional expr(s) for that branch. The action is composed by the statements inside the branch and the rest of statements in the original rule action.
2) For each rule with $ruleset$ parameters only over the home cluster, do nothing for the rule.
3) For each rule with $ruleset$ parameters over the remote clusters, or without parameters
   a) If there exists any $ruleset$ parameters over D, remove such parameters.
   b) For the rule guard, replace "Procs[]" over remote clusters with "ABSProcs[]", i.e. replacing concrete clusters with abstract clusters; if there exists any boolean sub-expr involving "$forall$" or "$exists$" with parameters over all the clusters, replace the sub-expr with two "$\wedge$" sub-exprs: one with the parameter over the home cluster, and the other with the parameter over the remote clusters; if there exists any boolean sub-expr involving variables in D, replace the sub-expr with "$true$".
   c) For each statement in the rule action, replace "Procs[]" over remote clusters with "ABSProcs[]",
      i) assignment: if the designator involves any variable in D, remove the assignment; otherwise, for the expr to be assigned to the designator, if there exists any sub-expr involving variables in D, replace the sub-expr with a nondeterministic value in the type of the sub-expr.
      ii) forstmt: if the quantifier in the forstmt ranges over all the clusters, divide the forstmt into two forstmts: one with the quantifier on the home cluster, and the other on the remote clusters; if the quantifier involves any variable in D, remove the forstmt; for each statement inside the forstmt, goto 3(c).
      iii) whilestmt: for the condition expr in the whilestmt, if there exists any sub-expr involving variables in D, replace the sub-expr with "$true$"; for each statement inside the whilestmt, goto 3(c).
      iv) aliasstmt: if there exists any variable declaration involving variables in D, remove the declaration.
      v) proccall: if there exists any parameter in the proccall involving variables in D, replace the parameter with a nondeterministic value; for each statement in the procedure, goto 3(c).
      vi) clearstmt, putstmt, errorstmt: if the stmt involves variables in D, remove the stmt.
      vii) assertstmt: for the expr inside assertstmt, if there exists any sub-expr involving variables in D, replace the sub-expr with "$true$".
      viii) returnstmt: for the expr in the returnstmt, if there exists any sub-expr involving variables in D, replace the sub-expr with a nondeterministic value in the type of the sub-expr.