Noname manuscript No.
(will be inserted by the editor)

Efficient Methods for Formally Verifying Safety Properties of Hierarchical Cache Coherence Protocols

Xiaofang Chen · Yu Yang · Ganesh Gopalakrishnan · Ching-Tsun Chou

Abstract Multicore architectures are considered inevitable, given that sequential processing hardware has hit various limits. Unfortunately, the memory system of multicore processors is a huge bottleneck. To combat this problem, one needs to design aggressively optimized cache coherence protocols. This introduces the design correctness problem for advanced cache coherence protocols which will be hierarchically organized for scalable designs. Experiences show that monolithic formal verification will not scale to hierarchical designs. Hence, one needs to handle the complexity of several coherence protocols running concurrently, i.e. hierarchical protocols, using compositional techniques.

To solve the problem, we develop a family of compositional approaches all based on assume-guarantee reasoning to reducing the verification complexity. We show that for the three hierarchical protocols with certain realistic features that we developed for multiple chip-multiprocessors, more than a 20-fold improvement in terms of the number of states visited can be achieved. Also, to avoid false alarms wasting designer time, we have developed an error trace justification method to eliminate false alarms using heuristics that also capitalize on our assume-guarantee approaches. Our techniques need no special tool support. They can be carried out using the widely used Murphi model checker along with support tools for abstraction and error trace justification that we have built.

Keywords Explicit state model checking, hierarchical cache coherence protocols, abstraction/refinement, assume-guarantee reasoning

This work is supported in part by SRC TJ 1318.001, 1847.00, and NSF CCF-0811429.

X. Chen · Y. Yang · G. Gopalakrishnan
School of Computing, University of Utah
50 South Central Campus Dr. Room 3190
Salt Lake City, UT, 84112, USA
E-mail: {xiachen, yuyang, ganesh}@cs.utah.edu

C.-T. Chou
Intel Corporation
3600 Juliette Lane, SC12-322
Santa Clara, CA, 95054, USA
E-mail: ching-tsun.chou@intel.com

1 Introduction

Multicore architectures are considered inevitable, given that sequential processing hardware has hit various limits. While there will be many differences from one multicore CPU to another, the high level architecture of these chips will be similar, i.e. they will all have multiple cores connected by on-die interconnect with large caches or even addressable memories inside each set of CPUs. Two classes of protocols will be used in multicore processors: *cache coherence protocols* that manage the coherence of individual cache lines, and *shared memory consistency protocols* that manage the view across multiple addresses. These protocols will also have non-trivial relationships with other protocols, e.g. hardware transaction memory [36] and power management protocols [10]. In this paper, we consider the problem of verifying cache coherence protocols, assuming the existence of suitable techniques to decouple and verify related protocols.

As memory systems govern the achievable performance of multicore processors, future cache coherence protocols will be aggressively optimized. They will also be extremely complex due to the exploitation of multiple on-chip wiring options, and due to an overall energy-aware protocol design strategy adopted. In addition, they will be hierarchically organized, with the use of different protocols for different levels of the hierarchy. The complexity of these hierarchical protocols will mean that conventional testing will be ineffective as a debugging method.

This paper addresses verification of cache coherence protocols, more specifically, verification of design representations in terms of high level guard/action style rules. We call this type of design representations as *specifications*. For the verification of cache coherence protocol specifications, modern industrial practice consists of modeling small instances of the protocols, e.g. three CPUs handling two addresses and one bit of data, in terms of interleaving atomic steps in guard/action languages such as Murphi [16] or TLA+ [27], and exploring the reachable states through explicit state enumeration. While the soundness of data size reductions can often be easily justified, e.g. based on simple syntactic criteria, a reduction of the number of nodes usually does not provide a conservative abstraction.The problem is generally known as the Parameterized Model Checking Problem (PMCP). In [18], Emerson and Kahlon developed an approach which can soundly reduce the complexity of the PMCP for certain systems. Combining recently proposed approaches for parameterized verification by [40] and [7] with the approach presented here can avoid the unsoundness of reducing the number of nodes, and we consider such combination to be future work.

Accumulated experience, starting from the early work on UltraSparc-1 by [41] to more modern ones, e.g. [20], [11] and [6], shows that designs captured at the high level descriptions can be model checked to help eliminate high level concurrency bugs. *Monolithic* formal verification methods – methods that treat the protocol as a whole – have been used fairly routinely for verifying cache coherence protocols from the early 1990s, e.g. in [3] and [22]. However, these monolithic techniques will not be able to handle the very large state space of hierarchical protocols, as will be shown in Section 3. Compositional verification techniques are essential to scale up the verification capacity.

Our Contributions

This paper contributes to ensuring the correctness of hierarchical cache coherence protocols through scalable formal verification methods. A key feature of our work is that it requires only conventional model checking tools for supporting our compositional approach. The primary contributions include the following:

1. Developing several hierarchical cache coherence protocols with reasonable complexity and realistic features as hierarchical coherence protocol benchmarks.
2. Based on these protocols, developing a compositional approach to decompose hierarchical protocols into a set of abstract protocols with smaller verification complexity. Our approach is conservative for safety properties and shown to be practical.
3. Improving the compositional approach to formally verify hierarchical protocols one level at a time. This approach gives even more complexity reduction while putting only mild requirements on part of the protocol designer. We also extend the approach to handle hierarchical protocols with non-inclusive caches and snoopy protocols..
4. Developing a set of guided search methods to automatically identify whether an error trace from an abstract protocol corresponds to a genuine error in the original hierarchical protocol.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 describes the three multicore protocols that we developed as the benchmarks of our verification. Section 4 presents the compositional approach to verifying hierarchical protocols, and Section 5 improves the approach. Section 6 describes techniques to mechanize part of the approach including abstraction and error trace justification. Finally, Section 7 summarizes the paper and provides concluding remarks.

2 Related Work

Hierarchical cache coherence protocols have existed for a long time, e.g. in chip-multiprocessors (CMPs) [4] or non-CMPs [21]. However, we are not aware of any published work that has reported formal verification of a hierarchical coherence protocol with reasonable complexity. Most of the previous work considers only one level of the hierarchy at a time, manually abstracting away other levels. Such separable verification is not entirely sufficient because there is no guarantee underlying the separation to ensure that no bugs will be missed.

Before we began developing our verification approaches, the first problem we had to solve was the lack of a single publicly available hierarchical protocol benchmark with reasonable complexity. To fill this void, we developed several 2-level coherence protocols for multiple CMPs (M-CMPs) each with different features (to be presented in Section 3). To our knowledge, these are the first 2-level hierarchical protocols developed in the academic literature with snoopy or directory based protocols, with reasonable complexity. It is interesting that the DirectoryCMP protocol [30], which was developed independently and in parallel with us, is very similar to one of our protocols, except that they focus more on the architectural side for performance.

Other than the hierarchical protocols which employ snoopy or directory based protocols in each level, there is also work on employing token coherence protocols for M-CMPs [31]. This type of protocols are flat for correctness but hierarchical for performance. As far as we know, token coherence has not been used in any commercial processor and some of the underlying reasons are addressed in [31].

In terms of verification of hierarchical cache coherence protocols, one of the earliest works was on the protocol of the Gigamax distributed multiprocessor [34]. In the protocol, bus snooping is used in both levels. SMV [35] was used to verify this protocol with two clusters each having six processors. In terms of complexity, many realistic details are abstracted away so that the protocol does not have the typical complexity of a hierarchical one. Thus, it is not clear whether directly employing SMV can verify hierarchical protocols with realistic features.

The work of [39] proposed an adaptive protocol called `Cachet` for distributed shared memory systems. The correctness of the protocol is enforced by two aspects. First, Cachet implements the Commit-Reconcile & Fences (CRF) which separates how a cache provides correct values from how it maintains coherence. Second, voluntary rules were introduced to separate the correctness and liveness from performance in protocol design. Their approach guarantees that any coherence protocol following the CRF memory model is correct by construction. While their approach seems appealing, it is not clear how to verify hierarchical protocols of other memory models.

As said earlier, separable verification by considering only one level and manually abstracting other levels of a hierarchical protocol is not entirely sufficient. As far as we know, our work is the first formal verification approach which is able to handle hierarchical protocols with complexity similar to that in the real world. Given a hierarchical protocol, we first construct a set of abstract protocols where each abstract protocol overapproximates the original protocol. After that, verification consists of dealing with the abstract protocols in an assume guarantee manner, refining each abstract protocol guided by counterexamples (to be presented in Section 4).

Our approach of using abstraction, assume-guarantee reasoning and counterexample guided refinement [14], derives the basic ideas from [12] on parameterized verification of non-hierarchical cache coherence protocols. Their work is again attributed to [32] and [33], who added support for this style of reasoning into Cadence SMV. The idea is later formalized by [24] and [29]. Our approach of using abstraction is also similar to environment abstraction by [26].

In order to handle non-inclusive caches and snoopy protocols, we extended our compositional approach by introducing auxiliary state variables to the abstract protocols. The detail will be presented in Section 5. The value of each auxiliary variable is a function of those state variables already in the protocol. These variables have some similarity with history variables, e.g. in [15] and [37].

Any conservative verification approach (such as ours) must come with methods for automatically eliminating false alarms. Without this, conservative approaches will waste considerable expert designer time by requiring them to wade through false error reports. To solve this problem, we have developed a collection of heuristics supported by a tool that we have built. The main problem we had to solve was to tell whether error traces produced with respect to abstract verification models have concretizations that are feasible in the real model. Naively approached, this problem is as hard as applying a monolithic approach to verify hierarchical protocols, which is known to be practically impossible. Our contribution is to develop a scalable method that exploits our compositional approach. Our basic idea relies on guided search which has some similarity with directed model checking, e.g. in [17]. In predicate abstraction and counterexample guided refinement, there has been a large body of research which uses symbolic methods to solve this problem, e.g. converting it to a satisfiability problem. Our heuristics are believed to be helpful for complexity reduction regardless of the specific verification approach taken. These ideas will be presented in Section 6.

3 Development of Benchmark Hierarchical Protocols

We developed three benchmark protocols [9]: two of them employ a directory based protocol in both levels, and one employs a snoopy and a directory based protocols in the two levels. In more detail, the first benchmark protocol models an inclusive cache hierarchy and the second models a non-inclusive cache hierarchy. The term *inclusive* means that the content

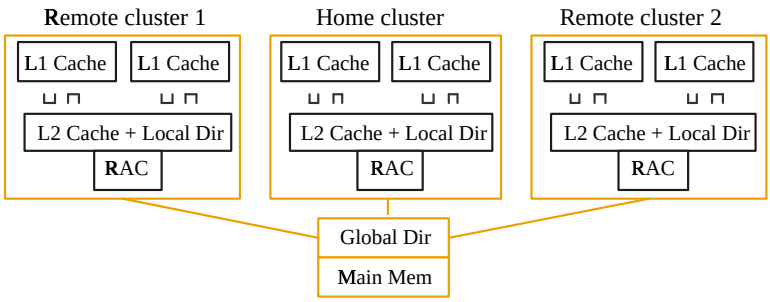

Fig. 1 A 2-level hierarchical cache coherence protocol.

of the L1 caches is a subset of that of the L2 cache in the same chip. *Non-inclusive* means that for any cache line in the L1 cache, it does not need to be in the L2 cache of the same chip. The cache state of MSI [5] or MESI [38] is used in each level. Also, features including silent dropping on non-modified cache lines and unordered network channels are modeled.

We modeled the three protocols in the language of Murphi, and performed all the experiments using the Murphi model checker. Murphi is one of the most widely used tools for the verification of cache coherence protocols in industry, e.g. the work of [12], [20] and [22] all mentioned the usages of Murphi. The language of Murphi supports the data types of booleans, enumerations, finite subranges of integers, arrays, records, etc. Each transition of Murphi is a *rule* in the form of *guard* $==>$ *action*, where *guard* specifies the condition under which the *action* can execute, and *action* specifies the updates to be performed. The modeling philosophy of Murphi follows the approach pioneered in Unity [8], a well-known guarded command language.

3.1 The First Benchmark: A Directory Based Inclusive Protocol

Our first benchmark protocol is derived by combining features from the FLASH [25] and DASH [28] 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 tackle this verification challenge problem. One address is modeled in the protocol as is typical in model checking based verification for coherence, because the behavior of most cache coherence protocols does not depend on the number of cache lines. Figure 1 intuitively depicts the protocol.

In more detail, the protocol is composed of three non-uniform memory access (NUMA) clusters: one home and two identical remote clusters. Here, we use *cluster* to refer to a chip-multiprocessor. Each cluster contains two symmetric L1 caches, an L2 cache and a local directory. “RAC” in the figure stands for remote access controller. It is the controller used to communicate with other clusters and the global directory. The main memory in reality is attached to every cluster. The fact there is only one memory in our protocol model is a consequence of employing our 1-address abstraction.

In the 2-level hierarchy, the level-1 protocol is used within a cluster. It tracks which line is cached in what state at which cache(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 the cluster level. The DASH protocol is adapted to keep the clusters consistent. Other features of the protocol include: (i) the L1 and L2 caches

are modeled as inclusive in a cluster, (ii) both levels use the MESI protocol supporting explicit writeback and silent dropping on non-modified cache lines, and (iii) both levels use unordered network channels, i.e. messages can arrive out of order.

In the process of developing the hierarchical protocol, we discovered that a straightforward adaption of the FLASH and DASH protocols to support an MESI cache state management scheme can result in a protocol that livelocks. For example, suppose an L1 cache silently drops an exclusive line, and the local directory receives a request from another cluster. The local directory will forward the request to the L1 cache and the latter will send a negative acknowledgement (NACK). After receiving the NACK, the local directory cannot use the L2 cache copy to reply the request as it is possible that the L1 cache has issued a writeback request. Now the local directory can only send a NACK to the requester and this can result in livelock.

There can be many solutions to the livelock problem. Here we just describe one which may not be very efficient in performance. We prevent the livelock problem by making writeback a blocking operation, i.e. after a cache sends a writeback request, it cannot issue new requests or process forwarded requests until an acknowledgment is received. Also, a new type of NACK messages is introduced to notify a directory that silent drop may have happened. This type of NACK messages can only be used by a cache when its cache line is invalid and it is not waiting for any reply of a former request. Upon receiving the new type of NACK messages, the directory will reset the exclusive ownership if necessary.

In summary, our first hierarchical benchmark protocol coded in Murphi occupies about 2500 lines of code (LOC). It contains the safety properties for control logic and datapath that a coherence protocol typically satisfies. Our protocol and that of [30] are some of the largest publicly available protocol benchmarks. To the best of our knowledge, the protocol of [30] has not been subject to formal verification.

3.2 The Second Benchmark: A Directory Based Non-inclusive Protocol

Our second benchmark protocol has the same configuration as the first one: it has three clusters each with two symmetric CPUs. The intra- and inter-cluster protocols all use a directory based MSI protocol. Here we choose MSI instead of MESI because the non-inclusive protocol is much more complex than the inclusive one, as to be shown in the experimental results in Section 5.3.

We will still use Figure 1 to represent the non-inclusive protocol. According to the definition, with non-inclusive caches, for any cache line in an L1 cache, the line does not need to be in the L2 cache of the same cluster. This characteristic is modeled in our protocol by allowing a cache line in the L2 cache to be silently dropped nondeterministically, if it is not the current owner. Also, any shared cache line is allowed to be silently dropped. Moreover, when an L2 cache line is evicted, the local directory entry of the cluster can also be evicted. Compared against the inclusive benchmark protocol, this protocol has several different features as follows.

First, for the network channels used inside a cluster, other than those in the inclusive protocol, there also exists a set of broadcast channels. These channels will be used when there is a cache miss in the L2 cache and the local directory entry has been evicted. At this time, the request for the cache miss will be broadcast to all the L1 caches in the cluster. With these broadcast channels, the L1 caches can send coherence requests to the local directory without contending with the broadcast requests.

Second, the use of non-inclusive caches can result in situations where the local directory has an imprecise record of a cache line. For example, suppose initially an L1 cache (agent-1) of a cluster has a shared copy, and this information is recorded in the local directory. After that, the L2 cache line and the local directory of the line are evicted. Later, another L1 cache (agent-2) of the cluster requests a shared copy. At this time, because the local directory of the line has been evicted, the request will be broadcast inside the cluster. The request will be negatively acknowledged by agent-1 as it is not safe for agent-1 to supply its data: the broadcast request could be interleaved with an invalidate request coming from outside the cluster. Thus, the request will be forwarded to the global directory. Suppose the request is finally granted, and the local directory of the line records it. Now the local directory has lost the information that agent-1 also has a shared copy. This type of imprecision can lead to coherence violations in certain cases. For this problem, there can be many ways to fix it and we omit our solution here.

In summary, our second benchmark protocol coded in Murphi occupies about 2500 LOC. It models a 2-level directory based MSI protocol with the features including: shared cache lines can be silently dropped, the L2 cache is non-inclusive of the L1 caches of the same cluster, messages can be delivered out of order, broadcast network channels are modeled, etc. Overall, it is similar with the protocol used in Piranha [4] except the latter makes various optimizations to improve performance.

3.3 The Third Benchmark: A Multicore Coherence Protocol With Snooping

Our third benchmark protocol is much simpler than the first two. It models two symmetric clusters where a snoopy protocol is used inside a cluster, and a directory based MSI protocol is used among the clusters. Figure 2 shows the protocol.

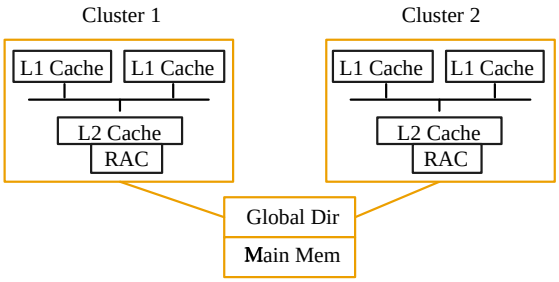

Fig. 2 A hierarchical cache coherence protocol with snooping.

In more detail, our snoopy protocol does not employ split transaction buses. Instead, when a cache puts a request on the bus, it will not release the bus until the response is received. Split transaction protocols are usually employed to improve the bus efficiency. Here, we choose this simple protocol just to check whether our approaches for reducing the verification complexity can be applied to snoopy protocols or not. For future work, we plan to develop and verify snoopy protocols with split transaction buses as well.

Finally, the L1 and L2 caches on a chip are modeled as non-inclusive caches, and write-backs on exclusive cache lines are modeled in both levels. Overall, this protocol coded in Murphi occupies about 1000 LOC.

Our first two benchmark protocols are in fact very complex: after all the shallow bugs are fixed, the model checking failed due to state explosion after more than 0.4 billion of states explored individually. The experiments were done on a machine using 18GB of memory, symmetry reduction, and with 40-bit of hash compaction provided by Murphi (there is no partial order reduction in standard Murphi). This is not surprising, considering the multiplicative effect of having four instances of coherence protocols running concurrently, i.e. one inter-cluster and three intra-cluster protocols, We believe all hierarchical coherence protocols will state explode in this manner. This section describes our compositional approach to scaling up the verification capacity.

Our compositional approach consists of two steps: abstraction and counterexample guided refinement. Given a hierarchical protocol, we first use abstraction to decompose the protocol into a set of abstract protocols each with smaller verification complexity. We then apply counterexample guided refinement on the abstract protocols using assume-guarantee reasoning. Our compositional approach is conservative, in the sense that if the abstract protocols can be verified correct, the original hierarchical protocol must also be correct with respect to its verification obligations.

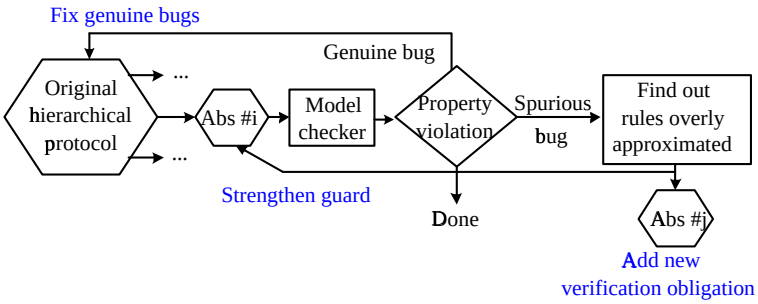

Fig. 3 The workflow of our approach.

The workflow of our approach is shown in Figure 3. That is, given a hierarchical protocol, we first construct a set of abstract protocols (`Abs #i`'s) where each abstract protocol overapproximates the original protocol by construction. We will formally define overapproximation in Section 4.5. We then model check each abstract protocol individually. If a genuine bug is found in an abstract protocol, we fix the bug in the original protocol and regenerate all the abstract protocols. If a spurious bug is reported in `Abs #i`, we constrain `Abs #i` and at the same time, add a new verification obligation to one of the abstract protocols `Abs #j` to guarantee that the constraining on `Abs #i` is sound. The question of how to identify whether an error corresponds to a genuine or a spurious bug will be discussed in Section 6.

In the following subsections, we will first present an overview of our compositional approach, and then illustrate the abstraction and refinement in detail. We will take the first benchmark protocol as the driving example to illustrate our compositional approach. In Section 5, we will describe how to extend the approach to verify the other two benchmarks.

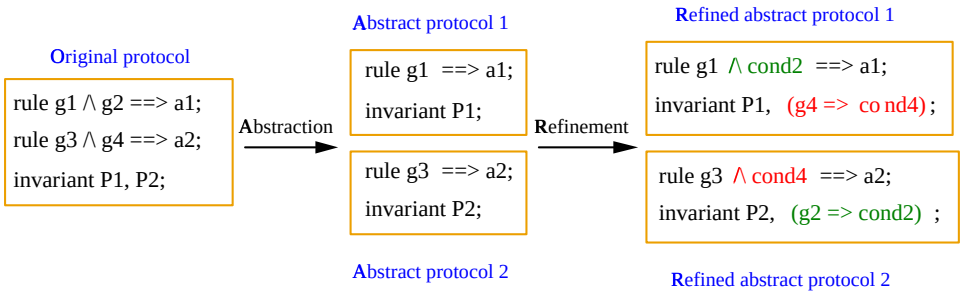

Fig. 4 An overview of our abstraction and refinement.

4.1 An Overview of Our Compositional Approach

We will illustrate the basic idea of our compositional approach using a simple example in the context of Murphi. Suppose there is a hierarchical protocol which contains two rules as shown in the left part of Figure 4, where the symbol $\wedge$ represents conjunction. Two properties $P1, P2$ are to be verified in the protocol. Let us decompose the hierarchical protocol into two abstract protocols each with one abstracted rule shown in the middle of Figure 4. As we can see, the guard condition of each abstracted rule is weaker than that in the original protocol. Thus, the abstract protocols may have more behavior than those in the original protocol.

We then model check each abstract protocol individually. Suppose a counterexample is found in the first abstract protocol which violates $P1$, and this violation corresponds to a spurious error. Now we perform refinement for the abstract protocol: (i) we strengthen the guard $g1$ of the abstracted rule into $g1 \wedge cond2$, and (ii) we add a new verification obligation to the second abstract protocol asserting that $cond2$ is weaker than $g2$. Intuitively, step (ii) means that $g1 \wedge cond2$ is weaker than $g1 \wedge g2$, i.e. the abstract protocol still overapproximates the original hierarchical protocol. Similarly, suppose a violation to the property $P2$ is encountered in the model checking of the second abstract protocol. We can perform refinement in a similar way by strengthening the guard of the abstracted rule, from $g3$ to $g3 \wedge cond4$, and also adding a new property $g4 \Rightarrow cond4$ to be verified in the first abstract protocol. These steps are shown in the right part of Figure 4.

After the above refinement, suppose both the abstract protocols are verified correct. At this time, we can claim the correctness of the original hierarchical protocol, i.e. $P1, P2$ both hold. Section 4.5 will present the soundness proof of our compositional approach. In the following, we will illustrate the abstraction, the assume-guarantee reasoning, and the experimental results in detail.

4.2 Abstraction

For the hierarchical protocol shown in Figure 1, our abstraction will decompose it into three abstract protocols. Figure 5 shows one of the abstract protocols. Contrast with Figure 1, we can see that it retains all the details of the home cluster while abstracts away some of the remote clusters. The other two abstract protocols are similar, except that in each abstract protocol, one remote cluster is retained and the remaining clusters are abstracted. Due to the symmetry which is declared in the protocol through scalarsets [23] between

the two remote clusters, the two abstract protocols obtained by retaining one of the remote clusters, respectively, are symmetric. As a result we will only consider the two distinct (non-symmetric) abstract protocols.

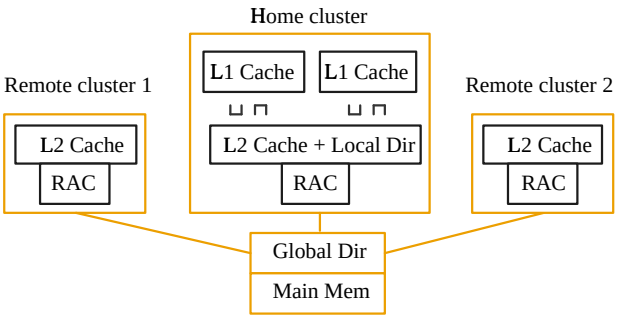

Fig. 5 An abstract protocol.

The derivation of the abstract protocols from the hierarchical protocol involves the abstraction of the state variables, the transition relation, and the verification obligations.

Abstraction of State Variables

Our abstraction of state variables is simply a *projection.* That is, we remove certain state variables from the original protocol to obtain the set of state variables of an abstract protocol. In general, the set of state variables of each abstract protocol is a proper subset of that in the original protocol. For example, for the abstract protocol in Figure 5, its state variables can be obtained by removing the following variables from the remote clusters of the protocol in Figure 1: (i) the L1 caches, (ii) the set of network channels used inside a cluster, and (iii) the local directory. The minimal requirement of our approach on state variables is that the union of the state variables from all the abstract protocols be the same as that of the original protocol, as will be formally presented in Section 4.5.

Abstraction of The Transition Relation

After we remove certain state variables from the original protocol, we need to adjust the transition relation accordingly, to obtain an abstract protocol. Our abstraction of the transition relation is an overapproximation. In this paper, we consider rule based systems in which each transition is a rule in the form of $guard \longrightarrow action$, where $guard$ is a predicate on states and $action$ is a function to update states. Basically our abstraction of the transition relation works as follows. For every rule $guard \longrightarrow action$ in the original protocol, (1) if a literal, i.e. a propositional variable or its negation, in $guard$ contains any variable that has been eliminated, we replace the literal with $true$; (2) for any assignment of the form v := E in $action$, if v is a state variable that is eliminated, then the whole assignment will be eliminated in the abstract protocol; otherwise if E contains even one variable that has been eliminated, we replace E with a non-deterministic selection over the type of E. Here, we assume that the transition relation is already in the negation normal form, i.e. negation is only applied to propositional variables. Otherwise, the transition relation can be converted into the form with preprocessing.

In the above, for every rule in the original protocol, our abstraction generates a new rule in an abstract protocol. The abstraction ensures that the guard of the new rule is either the

same or more permissive than the original one. Similarly, the action of the new rule allows the same or more updates to happen.

Now let's look at one example of the abstraction of the transition relation. Consider the scenario when a writeback request from an L1 cache line is received by the local directory of a remote cluster (*r1*) in the hierarchical protocol. As shown on the left-hand side of Figure 6, when this request is received, the following updates will be performed: (i) the L2 cache line copy is updated, (ii) the local directory sets the L2 cache line to be the current owner (*HeadPtr*), (iii) the L2 cache line state is set to modified, (iv) the writeback acknowledgment (*UniMsg*) is sent, and (v) the writeback request (*WbMsg*) is absorbed.

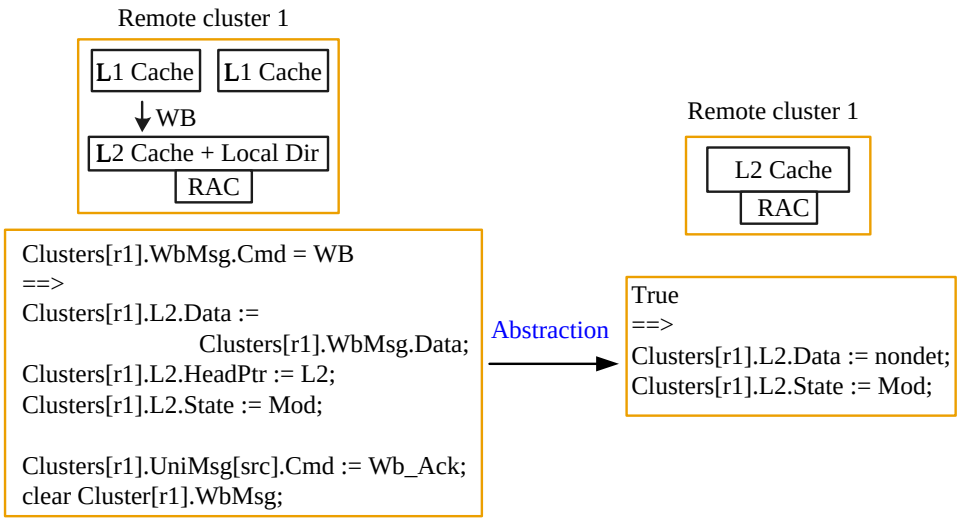

Fig. 6 An example of abstraction of the transition relation.

For the abstract protocol shown in Figure 5, the state variables *WbMsg*, *HeadPtr* and *UniMsg* of *Clusters[r1]* will all be eliminated from the protocol in Figure 1. So applying our abstraction, the guard of the writeback rule will be converted to $true$, and the updates (ii), (iv) and (v) will simply be eliminated. The resulting rule is shown on the right-hand side of Figure 6. It states that at any time, the L2 cache line of the remote cluster can be updated, clearly an overapproximation of the original rule.

Abstraction of Verification Obligations

For each verification obligation in the original hierarchical protocol, it will be abstracted similarly to a rule's guard. That is, if the verification obligation contains any state variable that is eliminated in the abstract protocol, we replace the verification obligation with $true$.

Now for every verification obligation p in the original protocol, the abstraction will generate a verification obligation p_i in each abstract protocol. To guarantee that our compositional approach is conservative – i.e. when all the abstract protocols are verified correct the original protocol must be correct – we require that $\wedge p_i \Rightarrow p$ must hold. Section 4.5 will describe this more formally, in the soundness proof of our compositional approach.

4.3 Assume-guarantee Reasoning

From Section 4.2, it is clear that each abstract protocol, by construction, overapproximates the original hierarchical protocol on those state variables that are retained. As described in the workflow of our compositional approach in Figure 3, after the abstract protocols are obtained, we model check them individually with respect to the abstracted verification obligations. When a property violation is encountered in an abstract protocol, there are two possibilities: (i) it is a genuine bug in the original protocol, or (ii) the bug is spurious due to the abstraction process. The question of how to identify whether it is a genuine bug will be discussed in Section 6.

For genuine bugs, we fix them in the original protocol, regenerate all the abstract protocols and iterate the process. For a spurious bug, we constrain the abstract protocol and at the same time, add a new verification obligation to one of the abstract protocols. Essentially, it suffices to prove every such new verification obligation in any of the abstract protocols. Since each abstract protocol retains a different subset of the original protocol, the abstract protocols in fact depend on each other in the refinement. The above refinement process is iterated until all the abstract protocols are verified correct.

In more detail, when a spurious bug is found in an abstract protocol, we first find out from the counterexample, which rule is overly approximated that causes the bug. Let this rule be $g \longrightarrow a$. We then find out the corresponding rule in the original protocol from which the abstracted rule is obtained; let it be $G \longrightarrow A$. By construction we have $G \Rightarrow g$. Now we strengthen the abstracted rule into $g \wedge p \longrightarrow a$ where p is a formula. At the same time, we add a new verification obligation to one of the abstract protocols, in the form of $g' \Rightarrow p$ where $G \Rightarrow g'$ is valid. Intuitively, this means that p is weaker than g', and also G. Thus the strengthened guard $g \wedge p$ is weaker than G, i.e. $G \Rightarrow g \wedge p$.

Mh

```
True &
(Clusters[r1].L2.State = Excl |
 Clusters[r1].L2.State = Mod)
==>
Clusters[r1].L2.Data := nondet;
Clusters[r1].L2.State := Mod;
...
```

Mr1

```
Clusters[r1].WbMsg.Cmd = WB
==>
Clusters[r1].L2.Data := Clusters[r1].WbMsg.Data;
Clusters[r1].L2.HeadPtr := L2;
Clusters[r1].L2.State := Mod;
Clusters[r1].UniMsg[src].Cmd := Wb_Ack;
clear Cluster[r1].WbMsg;
…
Invariant forall c: ClusterSet do
  Clusters[c].WbMsg.Cmd = WB
  ->
  (Clusters[c].L2.State = Excl |
   Clusters[c].L2.State = Mod)
end;
```

Fig. 7 Refine the writeback example.

An Example of Refinement

Now let's look at one example to see how the refinement is performed. In this example, let M denote our first benchmark protocol shown in Figure 1. Let M_h, M_{r1} denote the

abstract protocols in which the home cluster, and the remote cluster 1 is retained respectively. Consider the writeback rule shown on the left-hand side of Figure 6. In M_h, this rule will be abstracted to the one shown on the right-hand side of Figure 6, because the remote cluster 1 is abstracted. The abstracted rule simply states that, at any time, the L2 cache line of the remote cluster can be updated.

Clearly the abstracted rule in M_h can easily introduce violations to coherence as the L2 cache line can be updated randomly, and such a violation is a spurious bug. To refine M_h, we do two things: (i) constrain the guard of the abstracted rule such that only when the L2 cache line is exclusive or modified, can its copy be updated; (ii) add a new verification obligation (to be checked) in M_{r1}, asserting that whenever a writeback request is received from an L1 cache line, the L2 cache line must be exclusive or modified, i.e. the characteristics of inclusive caches. In Section 5, we will discuss how to perform refinement for non-inclusive caches. Figure 7 shows the refinement process.

Refinement Results After Abstraction

For the first benchmark protocol, a genuine bug was found in the refinement process. Basically, this bug relates to the following scenario: (1) a remote cluster (*r1*) is in a state where only the L2 cache line holds a modified copy, (2) a request for an exclusive copy from the other remote cluster (*r2*) is forwarded to *r1*, and (3) *r1* replies *r2*'s request and sets itself to invalid. In step (3), *r1* should have indicated whether the data supplied to *r2* is dirty or not; otherwise *r2* could silently drop the cache copy later. This in effect throws away the only dirty copy in the system, violating the coherence property. The bug is not hard to fix so we skip it here.

After the genuine bug was fixed in the original protocol, 18 spurious bugs were found in each abstract protocol. These spurious bugs can all be fixed in a similar way as we do for the writeback example, and the abstract protocols are refined. After that, all the verification obligations are shown to hold in the abstract protocols.

4.4 Experimental results

Table 1 shows the experimental results on the first benchmark protocol, using the monolithic approach and our compositional approach. The experiments were performed on an Intel IA-64 machine, using the 64-bit version of Murphi and the 40-bit hash compaction.

Table 1 Verification complexity of the first benchmark protocol.

Approaches	Protocols	# of states	Runtime (sec)	Verified?
Monolithic	Original	>438,120,000	>125,410	Non conclusive
Our compositional	Abs. 1	284,088,425	44,978	Yes
	Abs. 2	636,613,051	66,249	Yes

In the table, model checking on the original protocol using the traditional monolithic approach failed after more than 0.4 billion of states, due to state explosion. Our compositional approach was able to verify the protocol. The model checking took about 12 and 18 hours individually on the two abstract protocols. Other than this, it took about two days of human efforts to manually abstract the protocol, fix the bug and refine the abstract protocols. We will present how to mechanize our compositional approach in Section 6.

4.5 Soundness Proof

We now prove that our compositional approach is sound. That is, if all the abstract protocols can be verified correct, the original hierarchical protocol must be correct with respect to its properties as well. We first introduce several notations for the proof.

Some Preliminaries: States, Transitions, Models, Executions

Let V be a set of variables and D be a set of values which are both non-empty. A *state* s is a function from variables to values, $s : V \rightarrow D$. Let S be the set of states. For a state s and a set of variables X, $s \mid X$ denotes the restriction of s to the variables in X. A *transition* t consists of a *guard* g and an *action* a, where g is a predicate on states, and a is a function $a : S \rightarrow S$. We write $g \rightarrow a$ to denote the transition with guard g and action a.

A *model* has the form (V, I, T, A), where V is a set of variables, I is a set of initial states over V, T is a set of transitions, and A is a set of verification obligations (state formulas over V) which can be empty.

An *execution* of a model is based on steps in which a single enabled transition fires. For a transition $t = g \rightarrow a$, we write $s \xrightarrow{t} s'$ to denote that $g(s)$ holds and $s' = a(s)$. An execution of (V, I, T, A) is a sequence of states $s_0, s_1, \ldots$, such that $s_0 \in I$ and for all $i \geq 0$, there is a transition $t_i \in T$ such that $s_i \xrightarrow{t_i} s_{i+1}$. Finally, if a state s satisfies a formula p, we denote it by $p(s)$. If for any state s of any execution of a model $M = (V, I, T, A)$ s satisfies a verification obligation $p \in A$, then we say M satisfies p. If M satisfies all the verification obligations of A then we denote it by $\models M$.

A Theory Justifying Apparently Circular Reasoning

In the following, we will present two theorems to justify our compositional approach. Theorem 1 provides a formal proof to ensure that our abstraction is conservative. That is, given a hierarchical protocol, if all the abstract protocols which are obtained using our abstraction can be proved correct, the original hierarchical must be correct with respect to its verification obligations. Theorem 2 ensures that our refinement together with the abstraction is conservative. Thus, our compositional approach is sound, i.e. the correctness of the refined abstract protocols implies the correctness of the original protocol.

Theorem 1. Let $M = (V, I, T, A)$ be a model. If there exists a set of models $M_i = (V_i, I_i, T_i, A_i), i \in [1..n]$, $n \geq 0$, such that

1. $V_i \subseteq V$,
2. $I_i \supseteq I \mid V_i$,
3. For every t of the form $g \rightarrow a$ in T, $\exists t_i = g_i \rightarrow a_i \in T_i$ such that for every state s, s' where $s \xrightarrow{t} s'$ holds, $s \mid V_i \xrightarrow{t_i} s' \mid V_i$ holds, and
4. $\forall p \in A$ and $\forall i \in [1..n]$, $\exists p_i \in A_i$ where p_i is either $true$ or p, and $\wedge_{\{i \in [1..n]\}} p_i \Rightarrow p$ is valid.

Then $\wedge_{i \in [1..n]} \models M_i$ implies $\models M$.

Proof: We will prove this theorem by contradiction. Suppose $\models M$ does not hold while $\wedge_{i \in [1..n]} \models M_i$ holds. Then there exists a counterexample to a verification obligation $q \in A$ of M: $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{m-1}} s_m$ where $m \geq 0$, and $t_j = g_j \rightarrow a_j$ for all $j \in [0..m-1]$, such that $q(s_m)$ does not hold. We will prove by induction that for every $i \in [1..n]$, there exists a valid execution of M_i: $s_0 \mid V_i,\ s_1 \mid V_i,\ \ldots,\ s_m \mid V_i$. Here we will prove it for M_1 only; other M_i's can be proved similarly.

- Base Case: $j = 0$. From 2 in the theorem, it follows that $s_0 \mid V_1 \in I_1$.
- Induction Hypothesis: Suppose for $j \leq k$, $0 \leq k < m$, $s_0 \mid V_1, \ldots, s_k \mid V_1$ is a valid execution of M_1. Let $s_{j,1} = s_j \mid V_1$ for $j \in [0..k]$.
- Induction Step: We consider $j = k+1$. Because of $s_k \xrightarrow{t_k} s_{k+1}$, from 3 in the theorem, it follows that $\exists t_{j,1} = g_{j,1} \rightarrow a_{j,1}$ such that $g_j \Rightarrow g_{j,1}$ and $s_{j+1} \mid V_1 = a_j(s_j) \mid V_1 = a_{j,1}(s_j \mid V_1) = a_{j,1}(s_{j,1})$. Let $s_{j+1,1} = s_{j+1} \mid V_1$. Then $s_{j,1} \xrightarrow{t_{j,1}} s_{j+1,1}$ is a valid execution of M_1. That is, the induction holds for $j = k+1$.

Finally, consider q and its corresponding verification obligation in each M_i. From 4 in the theorem, it follows that for every $i \in [1..n]$, there must exist an $r \in [1..n]$ such that $q_r = q$. Let $s_{m,r} = s_m \mid V_r$. Since $\models M_r$ holds, we have $q_r(s_{m,r})$ holds, i.e. $q(s_{m,r})$ holds. So q must hold at s_m. Contradiction. □

Theorem 2. Let $M = (V, I, T, A)$ be a model. If there exists a set of models $M_i = (V_i, I_i, T_i, A_i), i \in [1..n]$, $n \geq 0$, such that

1. $V_i \subseteq V$,
2. $I_i \supseteq I \mid V_i$,
3. For every t of the form $g \rightarrow a \in T$, $\exists t_i = g_i \wedge e_i \rightarrow a_i \in T_i$ such that $g \Rightarrow g_i$, and if $e_i \neq \mathit{true}$ then $\exists k \in [1..n]$, $g' \Rightarrow e_i \in A_k$ where $g \Rightarrow g'$ is valid. Also, $a(s) \mid V_i = a_i(s \mid V_i)$ holds for every state s, and
4. $\forall p \in A$ and $\forall i \in [1..n]$, $\exists p_i \in A_i$ where p_i is either true or p, and $\wedge_{\{i \in [1..n]\}} p_i \Rightarrow p$ is valid.

Then $\wedge_{i \in [1..n]} \models M_i$ implies $\models M$.

Proof: We prove this by contradiction. Suppose $\models M$ does not hold while $\wedge_{i \in [1..n]} \models M_i$ holds. Then there must exist a counterexample to a verification obligation $q \in A$ of M: $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{m-1}} s_m$ where $m \geq 0$, and $t_j = g_j \rightarrow a_j$ for all $j \in [0..m-1]$, such that $q(s_m)$ does not hold. We will use induction to prove that for every $i \in [1..n]$, there exists a valid execution of M_i: $s_0 \mid V_i,\ s_1 \mid V_i,\ \ldots,\ s_m \mid V_i$.

- Base Case: $j = 0$. From 2 in the theorem, it follows that $s_0 \mid V_i \in I_i$.
- Induction Hypothesis: Suppose for $j \leq k$, $0 \leq k < m$, $s_0 \mid V_i, \ldots, s_k \mid V_i$ is a valid execution of M_i for every $i \in [1..n]$. Let $s_{j,i} = s_j \mid V_i$ for $j \in [0..k]$.
- Induction Step: We consider $j = k+1$. From 3 in the theorem, for $s_k \xrightarrow{t_k} s_{k+1}$ in M, for every $i \in [1..n]$, there exists $t_{k,i} = g_{k,i} \wedge e_{k,i} \rightarrow a_{k,i} \in T_i$, such that $g_k \Rightarrow g_{k,i}$, if $e_{k,i} \neq \mathit{true}$ then $\exists g'_k \Rightarrow e_{k,i} \in A_r$ for some $r \in [1..n]$ where $g_k \Rightarrow g'_k$ is valid. Also $a_{k,i}(s_k \mid V_i) = s_{k+1} \mid V_i$. Because $\models M_r$ holds, $g'_k \Rightarrow e_{k,i}$ must hold at state $s_k \mid V_r$, also at s_k. This, together with the fact that $g_k \Rightarrow g_{k,i}$, implies that $g_k \Rightarrow (g_{k,i} \wedge e_{k,i})$. So $t_{k,i}$ is enabled at $s_{k,i}$. Moreover, we have $a_{k,i}(s_{k,i}) = s_{k+1} \mid V_i$. Let $s_{k+1,i} = s_{k+1} \mid V_i$. Then $s_{k,i} \xrightarrow{t_{k,i}} s_{k+1,i}$ is a valid execution of each M_i, i.e. the induction holds for $j = k+1$.

Finally, we consider q and its corresponding verification obligation in each M_i. From 4 in the theorem, it follows that for every $i \in [1..n]$, there must exist an $r \in [1..n]$ such that $q_r = q$. Let $s_{m,r} = s_m \mid V_r$. Since $\models M_r$ holds, we have $q_r(s_{m,r})$ holds, i.e. $q(s_{m,r})$ holds. So q must hold at s_m. Contradiction. □

The above two theorems are different in that for every rule $g \rightarrow a$ in the original protocol, it is possible that the corresponding rule in an abstract protocol is in the form of $g_i \wedge e_i \rightarrow a_i$ in Theorem 2, instead of $g_i \rightarrow a_i$ in Theorem 1. Moreover, if the e_i is not equal to true, a

new verification obligation in the form of $g' \Rightarrow e_i$ must be provable in at least one abstract protocol, where $g \Rightarrow g'$ is valid.

5 Verification of Hierarchical Protocols One Level At A Time

In the previous section, we have presented a compositional approach to verifying hierarchical cache coherence protocols. Although this approach has been successfully applied to the first benchmark protocol which cannot be verified through traditional monolithic model checking, several problems still exist. These limitations include: (i) even a single abstract protocol models more than one level of the coherence protocols, e.g. as in Figure 5, thus still creating very large product space; (ii) details such as non-inclusive caches used in the other two benchmark protocols may not be handled. In this section, we present extensions to the compositional approach which can overcome these limitations.

5.1 Building Abstract Protocols One Level At A Time

We present a new abstraction approach, such that every resulting abstract protocol involves either an intra-cluster or the inter-cluster protocol, but never both. This approach is very similar to that in Section 4.2 in that each abstract protocol is obtained by overapproximating the original protocol.

In more detail, for the first benchmark protocol in Figure 1, our new approach will decompose it into four abstract protocols: one intra-cluster protocol for every cluster, and an inter-cluster protocol. Because the two remote clusters are of identical design, there are altogether three non-symmetric abstract protocols, as shown in Figure 8.

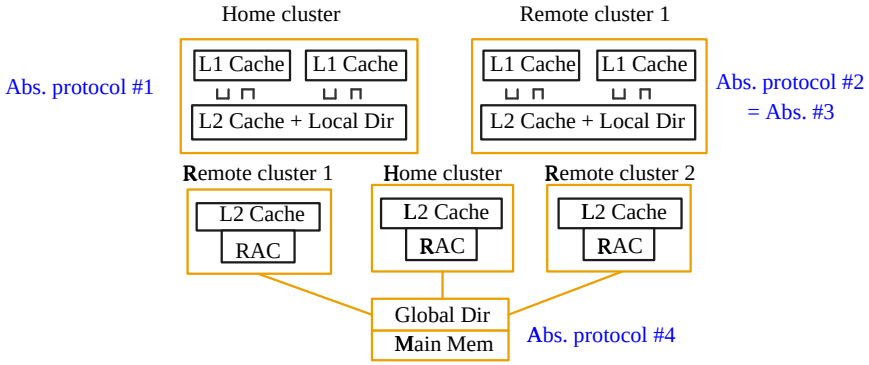

Fig. 8 Abstract protocols obtained via the new abstraction.

Again, the derivation of the abstract protocols from the hierarchical protocol involves the abstraction of state variables, the transition relation, and the verification obligations. The process of our new abstraction is the same as that in Section 4.2 so we skip it here.

To apply the new abstraction, we require that a hierarchical protocol be modeled in a *loosely coupled* manner, i.e. each rule of the protocol can only involve the state variables of one level of the protocol, including the variables shared by the neighboring levels. For

example, for the protocol shown in Figure 1 to be loosely coupled, we require that each rule only involves the variables of either an intra-cluster or the inter-cluster protocol, including the L2 cache line of the cluster which is shared by both the protocols. In the following, we will show an example which violates the loosely coupled modeling, and see how it prevents the new abstraction approach from being applied.

Consider a rule where the guard condition is: (i) the local directory of a cluster receives a shared request from an L1 cache line, (ii) the local directory is not busy on the line, i.e. not pending on other requests, and (iii) the L2 cache line is invalid. The action of the rule is: (1) the local directory is set to busy, (2) the RAC of the cluster is set to busy, (3) the request is forwarded to the global directory, and (4) the request from the L1 cache is absorbed. For this rule, clearly it involves both the intra- and the inter-cluster components, e.g. the local directory and the message to the global directory. Applying the new abstraction, in the abstract inter-cluster protocol, the rule will be abstracted such that the resulting guard becomes (iii) and the action becomes (2) and (3).

The above abstract rule simply states that whenever the L2 cache line is invalid, the RAC of the cluster will be set to busy and the a shared request will be sent to the global directory. Such an abstract rule is overly approximated, because it allows multiple requests for the same line to be issued, before a reply is received. To fix the spurious bug introduced by the abstraction, we need to strengthen the abstracted guard using the fact that the RAC cannot be busy if no coherence request has been sent to the global directory. At the same time, to ensure that the strengthening is sound, it has to be proved that when the local directory is not busy, the RAC must not be busy. This verification obligation requires that at least one abstract protocol maintains the details of the local directory and the RAC. However, no abstract protocol from the our new abstraction satisfies the requirement. Thus, the new abstraction approach cannot be applied.

In summary, if a hierarchical protocol is modeled as loosely coupled, our new abstraction can be applied so that more verification reduction can be achieved. Therefore, we suggest that hierarchical protocols be modeled as loosely coupled whenever the performance allows the tradeoff.

5.2 Verifying Non-inclusive Caches and Snoopy Protocols

In the previous sections, we have been using the first benchmark protocol, i.e. a 2-level directory based protocol with inclusive caches, as the driving example. As is known, there are many different characteristics between inclusive and non-inclusive caches, also between snoopy and directory based protocols. In this section, we will look at some of the differences and show how to extend our compositional approach to verify hierarchical protocols with non-inclusive caches and snoopy protocols.

Consider a commonly used coherence property – no two caches can write to the same line concurrently. For the inclusive benchmark protocol, this property can be represented with two verification obligations: (i) no two clusters can have their L2 cache lines both be exclusive or modified, and (ii) no two L1 cache lines can both be exclusive or modified in any cluster. Here, each verification obligation only involves the details of one level of the hierarchy; thus they can be verified in the abstract protocols using either of our abstractions in the compositional approach. However, for non-inclusive caches, when two L1 cache lines from different clusters are both exclusive or modified, their corresponding L2 cache lines may be invalid. Thus, with our per-level abstraction (verification one level at a time), it is unclear how to represent the above coherence property in abstract protocols.

Moreover, for non-inclusive caches, it may not be straightforward to refine an abstract protocol when spurious bugs are detected. For example, consider the writeback example again in Figure 6. After the rule is abstracted in the abstract inter-cluster protocol, it becomes such that the L2 cache line of the cluster can be updated at any time. In the inclusive caches, this overly approximated rule can be strengthened such that only when the L2 cache line is exclusive or modified, can the updates happen. However, the above guard strengthening cannot be applied to non-inclusive caches, because when a writeback request from an L1 cache is received, the corresponding L2 cache line can be invalid instead.

The above problems exist because when non-inclusive caches are employed in a cluster, the L2 cache line cannot represent as a *summary* of the L1 cache lines. To solve the problems, we introduce a set of auxiliary variables to the abstract protocols. The value of each auxiliary variable is a function of those that are already in the protocol. In more detail, we add an auxiliary variable *IE* (implicit exclusive) of boolean type to every cluster in a protocol with non-inclusive caches. The value of *IE* is defined in the abstract intra-cluster protocol, while *IE* will be used to refine the abstract inter-cluster protocol. Initially, all the *IE*'s are set to false. For our second benchmark protocol, i.e. a 2-level directory based protocol with non-inclusive caches, the *IE* of a cluster will be defined true if:

1. An L1 cache contains an exclusive or modified copy of the line, or
2. The networks inside the cluster contain a coherence reply with an exclusive or modified copy, or a grant reply from the broadcast channels, or
3. There is a writeback or shared writeback request of the line, or
4. There is an exclusive ownership transfer request of the line.

For our third benchmark protocol, the *IE* of a cluster will be defined true if any L1 cache of the cluster has an exclusive cache line. When *IE* is true, it means that the cluster must have an exclusive or modified copy in the cluster.

We also introduce a few other auxiliary variables to the second and third benchmark protocols in a similar way. After these variables are introduced, the two protocols can be verified in the same way as for the first benchmark protocol. For example, for the coherence property discussed at the beginning of this section, now the property can be represented as – (i) no two clusters can have $IE \vee (L2.State = Excl) \vee (L2.State = Mod)$ both true, and (ii) no two L1 caches can both be exclusive or modified in every cluster. Each of the properties can be verified using either of our abstractions in the compositional approach.

5.3 Experimental Results

We have applied the compositional approach with the per-level abstraction on the three benchmark protocols. Table 2, 3 and 4 show the experimental results on the three protocols, using the traditional monolithic model checking and our approach. The monolithic approach in Table 2 and 3 was performed on an Intel IA-64 machine with the 64-bit version of Murphi, and the rest were performed on a PC with an Intel Pentium CPU of 3.0GHz with the standard Murphi. In all the experiments, 40-bit hash compaction was used.

In more detail, we mainly consider the data consistency properties in these protocols. The properties include, e.g., (i) no two L1 caches can be both in the exclusive or modified state at the same time, (ii) each coherence read will get the latest copy in the system, and so on. More specifically, there are 10 coherence properties in the first protocol which employs an inclusive cache hierarchy, 13 properties in the non-inclusive protocol and 4 properties in

Table 2 Verification complexity of the first benchmark protocol.

Approaches	Protocols	# of states	Runtime (sec)	Mem (GB)	Verified?
Monolithic	Original	>438,120,000	>125,410	18	Non conclusive
Our compositional	Abs. inter	1,500,621	270	1.8	Yes
	Abs. intra 1	574,198	50	1.8	Yes
	Abs. intra 2	198,162	21	1.8	Yes

Table 3 Verification complexity of the second benchmark protocol.

Approaches	Protocols	# of states	Runtime (sec)	Mem (GB)	Verified?
Monolithic	Original	>473,260,000	>161,398	18	Non conclusive
Our compositional	Abs. inter	4,070,484	770	1.8	Yes
	Abs. intra 1	2,424,719	250	1.8	Yes
	Abs. intra 2	2,424,719	250	1.8	Yes

Table 4 Verification complexity of the third benchmark protocol.

Approaches	Protocols	# of states	Runtime (sec)	Mem (GB)	Verified?
Monolithic	Original	552,375	86	1.8	Yes
Our compositional	Abs. intra	1,947	6	1.8	Yes
	Abs. inter	15,371	7	1.8	Yes

the snoopy protocol. Finally, it takes about 10 to 20 iterations to refine each abstract protocol before all its verification obligations are verified.

From the tables, it is clear that our compositional approach is very effective in reducing the verification complexity of hierarchical protocols. More specifically, our approach can reduce the verification complexity of a hierarchical protocol to that of a non-hierarchical protocol, in which more than 95% of the state space reduction can be achieved.

5.4 Soundness Proof

The soundness of our approach on verifying hierarchical protocols one level at a time can be derived from that in Section 4.5, as follows. First, Theorem 1 of Section 4.5 still holds because the conditions of 1, 2, 3 and 4 do not depend on how the abstraction is performed on the original protocol, as long as the abstraction is conservative. Second, Theorem 2 still holds for our per-level approach despite the use of auxiliary variables. This is because the value of each auxiliary variable is a function of those that are already in the protocol, and this relationship is added as a verification obligation to an abstract protocol. As a result, introducing auxiliary variables will neither add new behaviors to nor remove existing behaviors from the protocol. Therefore, the soundness of our approach on verifying hierarchical protocols one level at a time can be derived from Theorem 2.

6 Mechanizing Our Compositional Approach

We have presented our compositional approach to verifying hierarchical cache coherence protocols in the previous sections. In this section, we will describe how to mechanize our compositional approach, esp. for the approach of verification one level at a time. To mechanize our approach, there are altogether three steps of work:

1. Given a hierarchical protocol and the protocol components to be eliminated, how to automatically generate the abstract protocol;
2. Once an abstract protocol is constructed and model checked, if a property violation is encountered, how to automatically identify whether the violation corresponds to a genuine error in the original hierarchical protocol;
3. If the violation is identified as spurious, how to automatically constrain and refine the abstract protocols.

In this section, we will describe our approaches for mechanizing the first two steps, with the main focus on the second step. For the third step, recent work [40] and [7] have proposed two different ways to mechanize it. In [40], the ordering information in message flows of cache coherence protocols are utilized to automatically construct new verification obligations. In [7], a Galois connection is assumed between the original and abstract protocols, and symbolic methods are used to automatically produce verification obligations provable in the Galois connection.

Given a hierarchical protocol and the protocol components to be eliminated, the process of constructing the abstract protocol is straightforward but tedious and time consuming. To mechanize this step, we have implemented a tool [1] based on Murphi. Basically, the abstraction procedure as described in Section 4.2 is recursively applied to every statement in the original protocol, to construct the abstract protocol. More detail about the implementation is described in [9]. In the following subsections, we will focus on mechanizing the second step.

6.1 Interface Aware Bounded Search for Error Trace Justification

Given an error trace from an abstract protocol, the process of identifying whether it corresponds to a genuine error in the original protocol requires deep knowledge of the protocol. To mechanize this process, we have developed a set of heuristics for it. The basic idea is to employ guided search to find out a *stuttering equivalent* execution of the abstract error trace in the original protocol. Our solution naturally capitalizes on our compositional approach described in the previous sections. In this space, no BDD/SAT based symbolic methods or even explicit state enumeration based methods, except for our methods described here, have handled the problem of error trace justification for hierarchical coherence protocols.

Given an error trace from an abstract protocol, let us first define whether this trace corresponds to a genuine error in the original hierarchical protocol. Let $M = (V, I, T, A)$ be a model which represents a hierarchical protocol. Let $M_i = (V_i, I_i, T_i, A_i), i \in [1..n]$ $(n \geq 0)$, be the set of abstract protocols of M that are built using our compositional approach. Let $p \in A$ be a verification obligation of M, and p_i be the corresponding verification obligation of p in each M_i. Suppose $E_k = s_{k0}, s_{k1}, \ldots, s_{km}$, $m \geq 0$, is an error trace of M_k ($k \in [1..n]$) which violates p_k. That is, for every vo in A_k, $vo(s_{kj})$ holds for every $j \in [0..m-1]$ while $p_k(s_{km})$ does not hold. We define E_k as a genuine error of M if there exists an execution of M that leads to the violation of p_k; otherwise E_k is a spurious error.

To identify an abstract error trace, a straightforward approach is to find out a stuttering equivalent execution in the original hierarchical protocol. The classical definition of stuttering equivalence is defined in [13] over Kripke structures. The notion of stuttering equivalence is used in our approach based on the following observation. For any execution $E = s_0, s_1, \ldots, s_m$ of M, because of the manner in which every abstract protocol M_i is constructed, there exists an execution $E' = u_0, \ldots, u_k$ of M_i for each i in $[1..n]$, such that

there exists $0 = j_0 < j_1 < \ldots < j_m \leq m$ such that

$$u_0 = (s_{j_0} \mid V_i) = \ldots = (s_{j_1-1} \mid V_i)$$

$$u_1 = (s_{j_1} \mid V_i) = \ldots = (s_{j_2-1} \mid V_i)$$

$$\ldots$$

$$u_k = (s_{j_m} \mid V_i) = \ldots = (s_m \mid V_i)$$

We will also denote every such E' as a stuttering equivalent execution of E on V_i. For every state s in E together with its index, s has a corresponding state s_i in E'. Thus we can map the index of s in E to that of s_i in E'. In more detail, for each state s of E together with the index idx of s in E, we define the *blocking index* (b_i) of (s, idx) to be the index of s_i in E'. For instance, in the above example, $b_i(s_{j_0}, j_0) = \ldots = b_i(s_{j_1-1}, j_1 - 1) = 0$, ..., and $b_i(s_{j_m}, j_m) = \ldots = b_i(s_m, m) = k$.

Our notion of stuttering equivalent executions suggests that if E' is an error trace of M_i, then if we can find out a stuttering equivalent execution E of E' in M, E' must be a genuine error. We implement this idea in a straightforward manner as follows.

Guided Search for Stuttering Equivalent Execution

Given an error trace $E_i = u_{i,0}, \ldots, u_{i,m_i}$ from an abstract protocol M_i, we perform a guided search on the original protocol M trying to find out a stuttering equivalent execution. In more detail, we first select an initial state $s \in I$ such that $s \mid V_i = u_{i,0}$, i.e. $b_i(s, 0) = 0$. Afterwards, we consider all the enabled rules of M at s. That is, for every t of the form $g \longrightarrow a$ where $g(s)$ holds, $s' = a(s)$, and $b_i(s, \mathrm{idx}) = j$, $j \geq 0$, we consider three cases:

1. $b_i(s', \mathrm{idx} + 1) = j$;
2. $b_i(s', \mathrm{idx} + 1) = j + 1$;
3. $b_i(s', \mathrm{idx} + 1) \neq j$, $b_i(s', \mathrm{idx} + 1) \neq j + 1$.

The first case means that s' and s, together with their indices, share the same block index. The second case means that s corresponds to $u_{i,j}$, and s' corresponds to $u_{i,j+1}$. The third means that the rule t updates certain variables of M so that it no longer corresponds to either $u_{i,j}$ or $u_{i,j+1}$.

We implement the above idea using a modified breadth first search (BFS). In more detail, for each state of M, we associate it with an integer corresponding to the block index of E_i. A state of M will be maintained in the state space search if it falls into the first two cases, while the state will be dropped if it falls into the third case. The BFS will stop if a state is found to have the block index m_i. At this time the error trace E_i must be genuine; otherwise we claim it as spurious.

Unfortunately, the above simple approach does not work in practice. The reason is because for every state s in the BFS queue, there is a huge number of next states s' where s' and s share the same block index. For example, consider an error trace from the abstract intra-cluster protocol of the home cluster in Figure 8. We have the situation that – for any state s of the original protocol, all the next states which are obtained by updating one of the remote clusters, the global directory, or the main memory, will have the same index as s; so they will all be maintained in the state space search, i.e. added into the BFS queue. Thus, the state space of the guided search is that of another hierarchical protocol. Our initial experiments confirmed it.

Heuristic 1: Interface Aware Guided Search

In this section, we refine the simple guided search exploiting the *interfaces*. We define interfaces as the state variables of a hierarchical protocol which are shared by two abstract protocols. More specifically, let $M = (V, I, T, A)$ denote the original hierarchical protocol. Let $M_i = (V_i, I_i, T_i, A_i), i \in [1..n]$, be all the abstract protocols of M. From Section 5, our abstraction has the property that $\cup_{i\in[1..n]} V_i = V$ when the auxiliary variables are not considered. We define $V_i \cap V_j, i \neq j, i, j \in [1..n]$ as the interfaces of M_i and M_j. For example, for the protocol as shown in Figure 1, we have four abstract protocols (two of them are symmetric) as in Figure 8. Let M_1, M_2, M_3 be the three abstract intra-cluster protocols, and M_4 be the abstract inter-cluster protocol. According to our definition, $V_i \cap V_4, i \in [1..3]$ are the interfaces for the three clusters respectively, i.e. the state variables of the L2 cache line of each cluster.

With interfaces, the guided search can work more efficiently as follows. For any error trace $E_i = u_{i,0}, \ldots, u_{i,m_i}, m_i \geq 0$ from an abstract protocol M_i, we consider any two successive states of E_i. The basic idea is based on the observation that M_i only interacts with other abstract protocols through its interfaces. Thus, if the updates between two successive states do not involve interface variables, the guided search should not either. The idea can be stated more formally as follows. For every $j \in [0..m_i)$, we consider $\delta_{i,j} \equiv u_{i,j+1} - u_{i,j}$ $\equiv \{v \mid u_{i,j+1}(v) \neq u_{i,j}(v), v \in V_i\}$. Let $\mathrm{If_i}$ be the interfaces of M_i.

1. If $\delta_{i,j}$ only involves variables from $V_i \setminus \mathrm{If_i}$, then the guided search will only explore the rules of M that update $V_i \setminus \mathrm{If_i}$;
2. Otherwise, the guided search will explore the rules of M that update V.

More specifically, the above idea can be applied to our benchmark protocols as follows. Consider the protocols in Figure 1 and Figure 8. For M_4, i.e. the abstract inter-cluster protocol, if $\delta_{4,j}$ only involves the interfaces of a cluster, then the guided search can simply explore the rules of M that only update the state variables of *that* cluster. Otherwise, the components among the clusters are updated. For this case, guided search will only need to explore the rules of M that update the variables of V_4. This kind of search will be efficient, as for any $u_{i,j}, j \in [0..m_i)$, our interface aware approach only needs to search for a subset of the state space of a non-hierarchical protocol, before matching $u_{i,j+1}$.

Similarly, error traces from an abstract intra-cluster protocol M_i as in Figure 8 can be identified. More specifically, if $\delta_{i,j}$ does not involve the interfaces of that cluster, the guided search should not either. However, when $\delta_{i,j}$ involves the interfaces of M_i, the guided search may have to consider the updates of the whole hierarchical protocol. This is because, e.g. when a request is sent from the home cluster to the global directory, the rest of the hierarchical protocol may have to be updated, before a reply can be received by the home cluster. Thus, our interface aware guided search may not work efficiently for error trace justification for abstract intra-cluster protocols.

To solve the above problem, we develop the following solution. Since the interface aware approach can work efficiently for the abstract inter-cluster protocol, we can model check this protocol first. By doing so, before working on the abstract intra-cluster protocols, we would have refined M_4 to a manner such that all its verification obligations hold. At this time, we can construct a new abstract protocol for each abstract intra-cluster protocol similar to that in Figure 5. That is, for each abstract intra-cluster protocol M_i, we construct a protocol M_{Ai} in which the cluster i is retained while all the other clusters are abstracted as in M_4. More formally, for the hierarchical protocol in Figure 1, three new abstract protocols $M_{Ai} = (V_{Ai}, I_{Ai}, T_{Ai}, A_{Ai}), i \in [1..3]$, will be constructed after M_4 is refined. Here, $V_{Ai} = V_4 \cup V_i$ and $A_{Ai} = A_4 \cup A_i$. Let $\mathrm{If_i} = V_4 \cap V_i$. Then $I_{Ai} = \{s_{40} \cup (s_{i0} \mid \mathrm{If_i}) \mid s_{40} \in I_4, s_{i0} \in I_i,$ $s_{40} \mid \mathrm{If_i} = s_{i0} \mid \mathrm{If_i}\}$. Finally, $T_{Ai} = \{t_4 = g_4 \rightarrow a_4 \mid t_4 \in T_4, \forall s_4 \in S_4, a_4(s_4) \mid \mathrm{If_i} = s_4 \mid$

If$_i$} $\cup$ $\{t = g \rightarrow a \mid t \in T, \exists s \in S, a(s) \mid V_i \neq s \mid V_i\}$. In T_{Ai}, the first part includes all the transitions from M_4 which do not update the interfaces of M_i, and the second part includes all the transitions from M that update V_i.

Now for every error trace E_i from the abstract intra-cluster protocol M_i, we can perform the interface aware guided search on M_{Ai} instead of M. In fact M_{Ai} is an abstract protocol of M, and M_i is an abstract protocol of M_{Ai}. Thus, we can use the interface aware guided search to first find a stuttering equivalent execution E_{Ai} of E_i in M_{Ai}, and if E_i is reported as genuine in M_{Ai}, we then try to find a stuttering equivalent execution of E_{Ai} in M.

We have implemented the interface aware guided search based on Murphi, and applied it to identify the error traces from the abstract protocols of the three benchmark protocols. Section 6.2 describes more detail. For many of the error traces, our interface aware approach can correctly identify the spurious/genuine error case with 1GB of memory. However, there are still some error traces for which our approach failed to identify even with 1.5GB of memory because there are too many states in the guided search. In the following, we will present another heuristic to improve the guided search by using bounded search.

Heuristic 2: Bounded Search

The basic idea of our bounded search is that given an error trace $E_i = u_{i,0}, u_{i,1}, \ldots, u_{i,m_i}$, $m_i \geq 0$ from an abstract protocol M_i, we restrict the BFS depth of the guided search for each block index $j \in [0..m_i)$ to a certain bound. If we cannot find a state with the block index $j+1$ using the bounded search, then we claim that there does not exist a stuttering equivalent execution of E_i, i.e. the error is spurious.

We choose the bound based on the following heuristics. Let $tr_j = u_{i,j} \xrightarrow{t_{i,j}} u_{i,j+1} (j \geq 0)$ be an arbitrary transition in an abstract protocol $M_i = (V_i, I_i, T_i, A_i)$, where $u_{i,j}, u_{i,j+1}$ are reachable states of M_i. We associate every such tr_j with a bound bnd as follows:

1. If there does not exist a stuttering equivalent execution of tr_j in M, then we set the bound of tr_j to be any natural number, e.g. $bnd(tr_j) = 1$;
2. Otherwise, there exists an execution $s_k, s_k+1, \ldots s_l$, $k, l \geq 0$ such that $s_k \mid V_i = s_{k+1} \mid V_i = \ldots = s_{l-1} \mid V_i = u_{i,j}$, $s_l \mid V_i = u_{i,j+1}$, and $s_k, \ldots, s_l$ are reachable states of M. There can exist multiple such executions in M for tr_j, and we set the bound of tr_j to be the minimum number of $l - k$.

Finally, we set the bound of the abstract protocol M_i, $bnd(M_i) = \max\{bnd(tr_j) \mid tr_j = u_{i,j} \xrightarrow{t_{i,j}} u_{i,j+1}$, $u_{i,j}$, $u_{i,j+1}$ are reachable states of $M_i\}$.

Based on the above definitions, we select the candidate bounds as follows. Consider any transition $tr = s \xrightarrow{t} s'$ in the abstract inter-cluster protocol. Case 1: tr only updates the state variables not in interfaces. In this case, our interface aware guided search as described in the first heuristic will only explore the rules that update the inter-cluster protocol components. Thus we set $bnd(tr) = 1$. Case 2: tr updates interface variables. In this case, we choose $bnd(tr)$ to be the length of the shortest stuttering equivalent execution between s and s'.

For example, consider a transition in the abstract inter-cluster protocol from the initial state to the state where a shared request is issued by the home cluster. The bound of this transition corresponds to the following execution – (i) an L1 cache of the home cluster initiates a shared request, (ii) the request is sent to the local directory, and (iii) the local directory checks that there is no valid cache line and also no other pending requests; so it issues a request to be sent to the global directory. Thus, the bound of this transition is 3.

Similarly we choose candidate bounds for each abstract intra-cluster protocol. The only difference is that when interface variables are updated in a transition tr, the stuttering equiv-

alent execution may have to involve updates from the rest of the hierarchical protocol. Again the solution that we developed in the first heuristic by introducing M_{Ai}'s can help solve the problem.

In theory, choosing a bound as in the above manner requires knowledge of the protocol. In most cases, the bound of a protocol can be provided by designers for just once. Fortunately, in practice we find that for high level protocol descriptions in Murphi, e.g. the FLASH [25] and GERMAN [19] protocols, candidate bounds are reasonably small where the value 10 or 15 is usually big enough. For example, we find that the bound 10 works perfectly for all of our three benchmark protocols.

```
 1: GUIDEDSEARCH(AbsStates[], N) {
 2:   let s = ChooseStartState(AbsStates[0]);
 3:   i = 0;
 4:   while (i < N − 1) {
 5:     let δ_A = AbsStates[i + 1] − AbsStates[i];
 6:     let move_A = move_func(δ_A);
 7:     clear(Q); enqueue(Q, s);
 8:     bound = 0;
 9:     while (bound < BOUND) {
10:       while (¬isEmpty(Q)) {
11:         ss = dequeue(Q);
12:         for each enabled rule r at ss {
13:           let ss′ = r(ss);
14:           let δ = ss′ − ss;
15:           let move = move_func(δ);
16:           if (move_A ≠ move) continue;
17:           if (¬move_A and
18:                ¬ updateSameCluster(δ_A, δ))
19:             continue;
20:           if (isProjection(AbsStates[i + 1], ss′)) {
21:             s = ss′;
22:             clear(Q_n); goto L1;
23:           }
24:           if (isProjection(AbsStates[i], ss′))
25:             enqueue(Q_n, ss′);
26:         }
27:       }
28:       Q = Q_n; clear(Q_n);
29:       bound ++;
30:     }
31:     return false;
32:     L1: i ++;
33:   }
34:   return true;
35: }
```

Fig. 9 Interface aware bounded search.

6.2 Implementation and Experimental Results

We have implemented a tool [2] which integrates the interface aware and bounded search heuristics, and we have applied it to the three benchmark protocols. Our interface aware bounded search is basically a breadth-first search. Figure 9 shows the main algorithm, in

which $AbsStates[]$ contains the sequence of the abstract states of the error trace, and N is the length of the trace.

Based on the error trace, the algorithm tries to find a stuttering equivalent execution. The function $ChooseStartState$ tries to find an initial state with the block index 0. The main loop from Line 4 to 34 tries to match the error trace till the index N. If so, we report the error as genuine, otherwise spurious.

In the algorithm, BOUND is the value of the bound that we have selected for the abstract protocol, and $bound$ is a local variable. The function $move\_func$ takes a parameter of state updates and checks whether the updates involve any variable from a user-provided file. When using the tool, users are expected to provide a file containing a set of state variables related to interfaces. For `Abs #4` in Figure 8, the file should contain all the the state variables of `Abs #4` excluding the interfaces (i.e., the L2 caches). While for the abstract intra-cluster protocols, the file should contain all the state variables from `Abs #4`. Line 16 means that we expect the guided search to update the same set of variables as δ_A does. Line 17 means that if both δ_A and δ update the intra-cluster state variables, but not on the same cluster, then we can also discard ss'. Finally, $isProjection$ is a function that simply tests whether the first parameter is a projected state of the second.

Before applying the guided search, we had previously verified the benchmarks expending a lot of manual labor to classify error traces as genuine or spurious. So we inserted one bug in each benchmark individually, with two bugs in the inter-cluster level and one in the intra-cluster level. The three benchmarks altogether generate eight distinct abstract protocols, which again generate 102 error traces.

For all these traces, our tool is able to correctly report the spurious/genuine case, each within *15 seconds*. The amount of memory required for each identification is less than 1GB. The maximum number of states explored among all the guided search is $6,622$, which is very small compared with the state space of more than 0.4 billion of one benchmark protocol.

Abstract error trace:

```
Invariant "MemDataProp" failed.
   1. Rule L2_Reset_pending, p:Home fired.
   2. Rule L2_inReq_OutReq, p:Home fired.
   3. Rule Dir_HomeGetX_PutX fired.
   4. Rule Cluster_WriteBack, p:Home fired.
   5. Rule L2_recv_OutReply, p:Home fired.
   6. Rule Cluster_inReq_WB, p:Home fired.
   7. Rule L2_Recv_WB, data:Datas_2,
         p:Home fired.
```

Guided search:

```
   1. Rule L2_Reset_pending, p:Home fired.
   2. Rule L2_inReq_OutReq, p:Home fired.
   3. Rule Dir_HomeGetX_PutX fired.

   The abstract error trace is spurious
   with bound=10, 1186 states have been
   explored.
```

Fig. 10 A spurious error trace and the result from our approach.

Furthermore, in 94 of the 99 all spurious error traces, our tool can precisely tell which rule in the abstract protocol is problematic, i.e. overly approximated. Figure 10 shows a

sample scenario for one of the cases. Here, the first half of the figure shows an error trace from an abstract protocol, and the second shows the output from our tool. For this example, our tool reports that the given error trace is spurious, after exploring $1,186$ states. Moreover, from the output it indicates that the rule *Cluster_WriteBack* in the abstract protocol is problematic, as the first three rules in both traces are the same. For the remaining five spurious error traces, they fall into three categories.

The first category is that it is possible that two rules in the original hierarchical protocol are different, while their abstracted rules are equivalent. We denote such two rules as *abstract equivalent.* Abstract equivalent rules are straightforward to realize and they do not make it difficult for users to tell the problematic abstract rule.

The second category is about the auxiliary variables introduced in our compositional approach for the second and the third benchmarks. The situation is that the auxiliary variables only exist in the abstract protocols but not in the original protocol. As a result, certain rules of an abstract protocol can only update the auxiliary variables, and the guided search may report a stuttering equivalent execution containing irrelevant rules than those in the abstract error trace. Again, this case will not make it harder for users to recognize the problematic abstract rule.

Finally, the third category is a little more complicated. Because each abstract protocol M_i overapproximates the original hierarchical protocol M on V_i, it is possible that for some executions E_i of M_i, there do not exist stuttering equivalent executions in M but there exists states of M which after projecting on V_i, are equal to the states of E_i. Under these situations, our guided search approach will always report the error traces as spurious.

In our experiments, one case falls into this category. For this case, the abstract intra-cluster protocol from the second benchmark has an abstract error trace of length 10 beginning with the following three rules: (i) a remote cluster is invalid and it receives an exclusive request from another cluster; (ii) the remote cluster starts processing the request; (iii) the remote cluster again receives a shared request from another cluster. For this trace, our tool will report that the problematic rule is (i) while the real one is (iii).

The reason our tool reports the rule of (i) is because at the initial state of the original protocol, only the main memory has a valid copy; thus it is impossible for the remote cluster to receive requests from others. However, there does exist an execution of the original protocol but it is not stuttering equivalent. The execution begins with the initial state; then the remote cluster requests an exclusive copy and gets granted; now when another cluster requests an exclusive copy, the global directory will forward it to the remote cluster; the remote cluster writes back the exclusive copy to the main memory before the outside request is received. At this time, the remote cluster is in the invalid state, and rules (i) and (ii) will execute.

In summary, the experiments show that our interface aware bounded search is very efficient to identify the spurious or genuine abstract error traces – it can correctly report all the 102 errors, and in 94 of the 99 all spurious cases, found the exact location of the error automatically. Without our approach, it requires designers to identify every such error trace, while with our approach, once the configuration files and the bounds are available (they can be set by designers just once), our tool can automatically identify the errors.

7 Summary

In this paper, we investigate techniques to reduce the verification complexity of hierarchical cache coherence protocols in the high level descriptions. As currently there is no hierarchical

protocol benchmark with reasonable complexity, we first develop three 2-level hierarchical protocols with different features: inclusive and non-inclusive cache hierarchies, and snoopy and directory based protocols. These protocols are modeled with certain realistic features so that their verification complexity is similar to those used in practice.

Based on these benchmarks, we develop a compositional approach with two different abstraction techniques. Given a hierarchical protocol, our approach first decomposes it into a set of abstract protocols with smaller verification complexity. The abstract protocols are then refined in an assume-guarantee manner. By only verifying the abstract protocols we can conclude the correctness of the original hierarchical protocol. For the benchmarks, we show that our compositional approach can reduce the verification complexity of the hierarchical protocols, to that of a non-hierarchical protocol.

Finally, we have partly mechanized our compositional approach. That is, given a hierarchical protocol and the protocol details to be abstracted away, we can automatically generate the abstract protocol. Also, given an error trace produced from an abstract protocol, we have developed a set of heuristics to automatically identify whether it corresponds to a genuine error in the original hierarchical protocol. For all the three benchmarks, we show that our heuristics are very effective in identifying all the genuine and spurious error traces. For future work, we plan to investigate how to integrate the automatic refinement of abstract protocols, to our mechanization approaches. We are also interested in combining our compositional approach for hierarchical protocol verification, with the parameterized verification approaches for non-hierarchical protocols (e.g. [12]).

References

1. http://www.cs.utah.edu/formal_verification/hldvt07.
2. http://www.cs.utah.edu/formal_verification/FMSD-submission.
3. D. Abts, D. Lilja, and S. Scott. Toward Complexity-Effective Verification: A Case Study of the Cray SV2 Cache Coherence Protocol. In *Int'l Symp. Computer Architecture Workshop Complexity-Effective Design*, 2000.
4. L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Int'l Symposium on Computer Architecture*, 2000.
5. Faa Baskett, Taa A. Jermoluk, and Daa Solomon. The 4d-mp graphics superworkstation: Computing + graphics = 40 mips + 40 mflops and 100, 000 lighted polygons per second. In *Digest of Papers, Thirty-Third IEEE Computer Society Int'l Conference*, pages 468–471, 1988.
6. B. Batson and L. Lamport. High-Level Specifications: Lessons from Industry. In *Formal Methods for Components and Objects*, 2002.
7. J. Bingham. Automatic Non-interference Lemmas for Parameterized Model Checking. In *Formal Methods in Computer Aided Design*, 2008.
8. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
9. X. Chen, Y. Yang, M. Delisi, G. Gopalakrishnan, and C.-T. Chou. Hierarchical Cache Coherence Protocol Verification One Level at a Time through Assume Guarantee. In *IEEE Intl. High Level Design Validation and Test Workshop*, 2007.
10. L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J.B. Carter. Interconnect-Aware Coherence Protocols for Chip Multiprocessors. In *Int'l Symp. on Computer Architecture*, 2006.
11. C. T. Chou, S. M. German, and G. Gopalakrishnan. Tutorial on Specification and Verification of Shared Memory Protocols and Consistency Models. In *Formal Methods in Computer-Aided Design*, 2004.
12. C.-T. Chou, P. K. Mannava, and S. Park. A Simple Method for Parameterized Verification of Cache Coherence Protocols. In *Formal Methods in Computer Aided Design*, 2004.
13. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.
14. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, 2000.
15. M. Clint. Program Proving: Coroutines. In *Acta Informatica*, pages 50–63, 1973.

16. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *IEEE Intl. Conference on Computer Design: VLSI in Computers and Processors*, 1992.
17. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit-state model checking with HSF-SPIN. In *SPIN Workshop*, pages 57–79, 2001.
18. E. A. Emerson and V. Kahlon. Reducing Model Checking of the Many to the Few. pages 236–254. Intl. Confererence on Automated Deduction, 2000.
19. S. German. Tutorial on Verification of Distributed Cache Memory Protocols. In *Formal Methods in Computer Aided Design*, 2004.
20. S. M. German. Formal Design of Cache Memory Protocols in IBM. In *Formal Methods in System Design*, 2003.
21. K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and Design of AlphaServer GS320. In *Architecture Support for Programming Languages and Operating Systems*, 2000.
22. G. Gopalakrishnan and W. Hunt. Formal Methods in Industrial Practice: A Sampling. In *Formal Methods in System Design*, 2003.
23. C. N. Ip and D. L. Dill. Verifying Systems with Replicated Components in Murphi. In *Computer Aided Verification*, pages 147–158, 1996.
24. S. Krstić. Parameterized system verification with guard strengthening and parameter abstraction. In *Automated Verification of Infinite-State Systems*, 2005.
25. J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Intl. Symposium on Computer Architecture*, pages 302–313, 1994.
26. S. K. Lahiri and R. E. Bryant. Constructing Quantified Invariants via Predicate Abstraction. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2004.
27. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
28. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Int'l Symposium on Computer Architecture (ISCA)*, pages 148–159, 1990.
29. Y. Li. Mechanized proofs for the parameter abstraction and guard strengthening principle in parameterized verification of cache coherence protocols. In *ACM Symposium on Applied Computing*, pages 1534–1535, 2007.
30. M. R. Marty. *Cache Coherence Techniques for Multicore Processors*. PhD thesis, University of Wisconsin-Madison, 2008.
31. M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M.K. Martin, and D. A. Wood. Improving multiple-CMP systems using token coherence. In *Intl. Symposium on High Performance Computer Architecture*, 2005.
32. K. L. McMillan. Circular compositional reasoning about liveness. In *International Conference on Correct Hardware Design and Verification Methods*, 1999.
33. K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Correct Hardware Design and Verification Methods*, pages 179–195, 2001.
34. K. L. McMillan and J. Schwalbe. Formal Verification of the Gigamax Cache-Consistency Protocol. In *Int'l Symposium on Shared Memory Multiprocessing*, 1991.
35. Kenneth L. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, 1992.
36. K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: log-based transactional memory. In *High-Performance Computer Architecture*, 2006.
37. S. Owicki. A Consistent and Complete Deductive System for the Verification of Parallel Programs. In *Symposium on Theory of Computing*, 1976.
38. M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Int'l Symposium on Computer Architecture*, 1984.
39. X. Shen, Arvind, and L. Rudolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. In *Int'l Conference on Supercomputing*, 1999.
40. M. Talupur and M. Tuttle. Going with the Flow: Parameterized Verification using Message Flows. In *Formal Methods in Computer Aided Design*, 2008.
41. L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein. System Design Methodology of UltraSPARC-I. In *Design Automation Conference*, 1995.