An Interface Aware Guided Search Method for Error-trace Justification in Large Protocols

Xiaofang Chen, Yu Yang and Ganesh Gopalakrishnan

UUCS-08-005

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

Abstract

Many complex concurrent protocols that cannot be formally verified due to state explosion can often be formally verified by initially creating a collection of abstractions (overapproximations), and subsequently refining the overapproximated protocol in response to spurious counterexample traces. Such an approach crucially depends on the ability to check whether a given error trace in the abstract protocol corresponds to a concrete trace in the original protocol. Unfortunately, this checking step alone can be as as hard verifying the original protocol directly without abstractions, which is infeasible. Our approach tracks the *interface behavior* at the interfaces erected by our abstractions, and employs a few heuristic search methods based on a classification of the abstract system generating these traces. This collection of heuristic search methods form a tailor-made guided search strategy that works very efficiently in practice on three realistic multicore hierarchical cache coherence protocols. It could correctly analyze 99 spurious error traces and 3 genuine error scenarios, each within 15 seconds. Also, on 94 of the 99 of the spurious errors, our approach can precisely report which transition in the abstract protocol is overly approximated that leads to the spurious error.

1 Introduction

The use of multiple shared memory cores organized around hierarchical cache coherence protocols is widely believed to be the main (if not the only) approach for future advancements in computing [1–4]. Central to such multicore machines are hierarchical cache coherence protocols in which multiple instances of coherence protocols are employed. For the verification of cache coherence protocols in their high level descriptions, 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 [5] or TLA+ [6], and exploring the reachable states through explicit state enumeration. Symbolic methods, e.g. BDD [7] or SAT [8], are yet to succeed for the verification of coherence protocols. For example, it was reported in [9] that a SAT solver fails to handle a modest protocol with about $40,000$ state variables, while the same set of techniques can handle a microprocessor model with on the order of one million state variables.

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 [10] and [11]. However, these monolithic techniques will not be able to handle the very large state space of hierarchical protocols, owing to the multiplication of the state space of the protocols running concurrently in the hierarchy. Compositional techniques are essential to scale up the verification capacity.

As far as we know, our previous work [3,4] are the first formal verification approach which is able to handle hierarchical protocols with complexity similar to that in the read world. 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.

Any conservative verification approach (such as ours) must come with methods for automatically eliminating false alarms. Without this, conservative approaches will waster 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.

The rest of the paper is organized as follows. Section 2 presents the background of our compositional approach. Section 3 describes our approach of error trace identification, including the basic idea, the interface aware guided search and the bounded search heuristics. Section 4 describes the implementation and experimental results of our approach on three hierarchical benchmark protocols. Finally, Section 6 summarizes our paper and provides concluding remarks.

2 Background

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 three 2-level coherence protocols for multiple chip-multiprocessors [4]. Two of the hierarchical protocols employ a directory based protocol in both levels, and one employs a snoopy and a directory based protocols. The cache state of MSI [12] or MESI [13] is used in each level. Also, features including unordered network channels, explicit writeback, and silent dropping on non-modified cache lines of the MESI cache state are modeled.

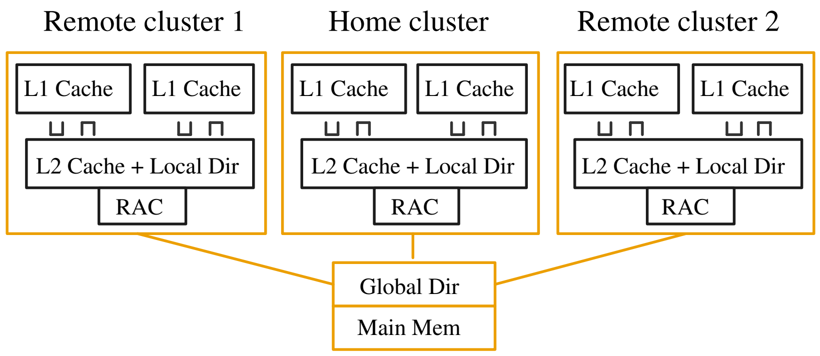

Figure 1: A 2-level hierarchical coherence protocol.

Figure 1 intuitively depicts one of our hierarchical protocols in which three non-uniform memory access clusters are modeled for one address: 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 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. Finally, the L1 and L2 caches are modeled as inclusive in the protocol, i.e. the content of the L1 caches is a subset of that of the L2 cache in the same cluster. For the other two hierarchical protocols, non-inclusve caches are modeled.

In the 2-level hierarchy of Figure 1, the intra-cluster protocol is used within a cluster to track which line is cached in what state at which cache(s). The inter-cluster protocol is used among clusters, tracking caching status in the cluster level. There are altogether four protocols running concurrently: three intra-cluster protocols and one inter-cluster protocol.

Based on these protocols, we developed a compositional approach [3]. 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, i.e. if the abstract protocols can be verified correct, the original hierarchical protocol must also be correct with respect to its verification obligations.

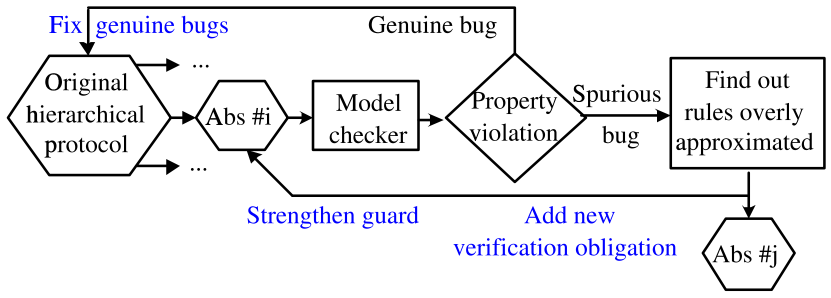

Figure 2: The workflow of our compositional approach.

Figure 2 shows the workflow of our compositional approach. 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. By 'overapproximate', we mean that for each abstract protocol, the set of its state variables is a subset of that of the original protocol, while the behavior involving the set of state variable are a superset of that in the original protocol. 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 answered in this paper.

For the hierarchical protocol shown in Figure 1, our compositional approach will decompose it into three abstract protocols. Figure 3 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 components 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.

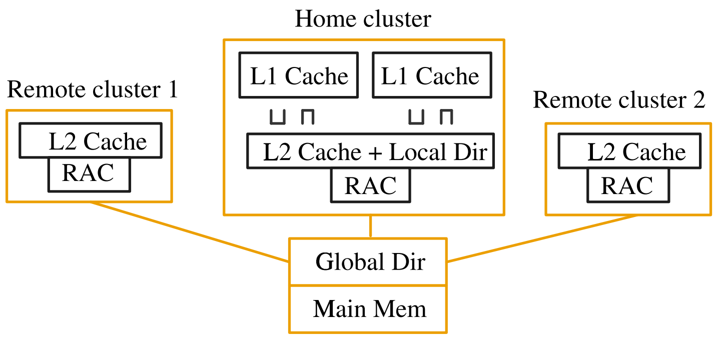

Figure 3: An abstract protocol.

We showed that our compositional approach can successfully verify one of our hierarchical protocols which cannot be verified through traditional monolithic model checking in [3]. However, several problems of the approach still exist and the limitations include: (i) even a single abstract protocol models more than one level of the coherence protocols, e.g. as in Figure 3, thus still creating very large product space; (ii) details such as non-inclusive caches and snoopy protocols used in the other two benchmark protocols cannot be handled.

To solve the above problems, we developed a new abstraction approach in [4]. In more detail, for the protocol in Figure 1, our new abstraction will decompose it into four abstract protocols: one abstract intra-cluster protocol for every cluster, and an abstract inter-cluster protocol. Because the two remote clusters are of identical design, there are altogether three non-symmetric abstract protocols, as shown in Figure 4. To apply the new abstraction approach, we require that a hierarchical protocol be modeled in a *loosely coupled* manner, i.e. each transition of the protocol can only involve the state variables of one level of the protocol, e.g. the components of an intra- or an inter-cluster protocol but not both.

We showed that our compositional approach with the new abstraction can reduce the verification complexity of a hierarchical protocol to that of a non-hierarchical protocol, in which

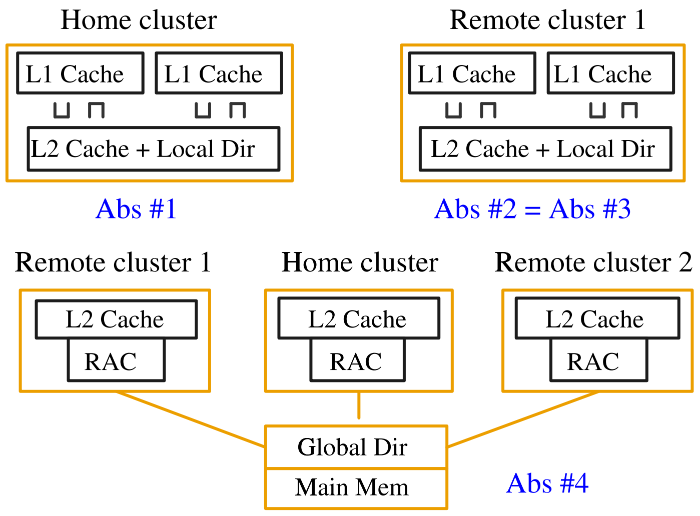

Figure 4: The abstract protocols of Figure 1.

more than 95% of the state space reduction can be achieved.

Also, to handle hierarchical protocols with non-inclusive caches and snoopy protocols, we extended the compositional approach in [4] by introducing auxiliary state variables to the abstract protocols, such that the auxiliary variable together with the L2 cache line can represent as a summary of the cache status in a cluster. The value of each auxiliary variable is a function of those state variables already in the protocol. By doing so, hierarchical protocols with non-inclusive caches and snoopy protocols can be verified with the same compositional approach as in [3].

To mechanize our compositional approach, there are altogether three steps of work:

1. Given a hierarchical protocol and the protocol components to be abstracted away, 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.

To mechanize the first step, we have implemented a tool in [4] to automate the process. For the third step, recent work [14] and [15] have proposed two different ways to mechanize it.

In [14], the ordering information in message flows of cache coherence protocols are utilized to automatically construct new verification obligations. In [15], 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.

This paper presents a solution to mechanize the second step, i.e. automatic abstract error trace identification. Our solution naturally capitalizes on our compositional approach described previously. 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.

3 Interface Aware Bounded Search for Error Trace Justification

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. We will introduce several notations for the definition.

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, $s : V \to D$. 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 $a : S \to S$. We write $g \mathbf{\rightarrow} a$ 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 $t = g \mathbf{\rightarrow} a$, 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 a model $M = (V, I, T, A)$ satisfies all the verification obligations of A then we denote it by $\models M$.

Coming back to the definition of a spurious or genuine abstract error, 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, $\forall vo \in A_i$, $vo(s_{kj})$ 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.

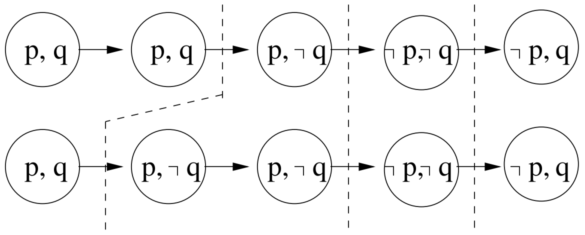

Figure 5: Classical notion of stuttering equivalence.

We now describe a heuristics to identify an error trace from an abstract protocol, based on the notion of *stuttering equivalence*. The classical definition of stuttering equivalence [16,17] is defined over Kripke structures [17] which has a function $L : S \rightarrow 2^{AP}$ that labels each state in S with the set of atomic propositions true in that state, where AP is a set of atomic propositions. Two paths $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ and $\rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$ are stuttering equivalent as shown in Figure 5, if there are two sequences of positive integers $0 = i_0 < i_1 < i_2 < \ldots$ and $0 = j_0 < j_1 < j_2 < \ldots$ such that for every $k \geq 0$,

$$L(s_{i_k}) = L(s_{i_k+1}) = \ldots = L(s_{i_{k+1}-1})$$
$$= L(r_{j_k}) = L(r_{j_k+1}) = \ldots = L(r_{j_{k+1}-1})$$

A finite sequence of identically labeled states is called a *block*.

The notion of stuttering equivalence is adapted 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 $E_i = u_{i,0}, \ldots, u_{i,m_i}$ of M_i, $i \in [1..n]$, $m_i \geq 0$, such that for every such i, there exists $0 = j_0 < j_1 < \ldots < j_{m_i} \leq m$ such that

$$u_{i,0} = s_{j_0} \mid V_i = \ldots = s_{j_1-1} \mid V_i,$$
$$u_{i,1} = s_{j_1} \mid V_i = \ldots = s_{j_2-1} \mid V_i, \ \ldots$$
$$u_{i,m_i} = s_{j_{m_i}} \mid V_i = \ldots = s_m \mid V_i$$

We will also denote every such E_i as a stuttering equivalent execution of E on V_i, and vice versa. For every state s in E, s has a corresponding state s_i in E_i, and we denote the index of s_i in E_i as the *block index* of s in M_i, i.e. $b_i : S \rightarrow \mathbb{N}$. For example, in the above, $b_i(s_{j_0}) = \ldots = b_i(s_{j_1 - 1}) = 0, \ldots$, and $b_i(s_{j_{m_i}}) = \ldots = b_i(s_m) = m_i$.

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

3.1 Guided Search for Stuttering Equivalent Executions

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 to find 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. $b_i(s) = 0$. Afterwards, we consider all the enabled rules of M at s. That is, for $\forall t = g \longrightarrow a$ where $g(s)$ holds, $s' = a(s)$, and $b_i(s) = j$, $j \geq 0$, we consider three cases:

$$1.\ b_i(s') = j; \quad 2.\ b_i(s') = j + 1; \quad 3.\ b_i(s') \neq j, b_i(s') \neq j + 1.$$

The first case means that s' and s 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 4. 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.

3.2 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 2, 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 4. 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 $\delta_{i,j} \equiv u_{i,j+1} - u_{i,j}$ $= \{v \mid u_{i,j+1}(v) \neq u_{i,j}(v), v \in V_i\}$. 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 4. For M_4, i.e. the abstract inter-cluster protocol, it is obtained by retaining the components among clusters while abstracting away certain components inside clusters. For this 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 4 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 3. 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, $T_{Ai} = \{t_4 = g_4 \longrightarrow a_4 \mid t_4 \in T_4, \forall s_4 \in S_4, a_4(s_4) \mid \mathrm{If_i} = s_4 \mid \mathrm{If_i}\} \cup \{t = g \longrightarrow a \mid t \in T, \exists s \in S, a(s) \mid V_i \neq s \mid V_i\}$. 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 4 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.

3.3 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 $M_i = (I_i, V_i, T_i, A_i)$, 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 equivalent 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 protocol awareness. 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 ([18]) 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.

4 Implementation and Experimental Results

We have implemented a tool [20] which integrates the interface aware and bounded search heuristics, and we have applied it to the three benchmark protocols. 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. 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 6 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 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 ap-

```
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.
```

Figure 6: A spurious error trace and the result from our approach.

proach for the second and the third benchmarks. The situation is that in an abstract error trace, certain rules only update the auxiliary variables because of the abstraction. As a result, 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.

The third category is about the overapproximation in our compositional approach. Because each abstract protocol M_i overapproximates the original hierarchical protocol M on V_i, it is possible that for some executions of M_i, there do not exist stuttering equivalent executions in M. 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 error, there does not exist a stuttering equivalent execution because at the initial state of the original hierarchical 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 M 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.

From the above, it is clear that when an abstract error trace begins with the additional behavior that do not exist in the original protocol, there will not exist any stuttering equivalent execution. This means that our approach of searching for a stuttering equivalent execution for error trace justification is a heuristic, i.e. it does not guarantee to always correctly identify the error trace.

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 bug 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.

5 Related Work

Our compositional approach to verifying hierarchical protocols includes abstraction, assume-guarantee reasoning and counterexample guided refinement. We derive the basic ideas from Chou et. al.'s work [21] on parameterized verification for non-hierarchical cache coherence protocols. Their work is again attributed to McMillan's work [22, 23] who added support for this style of reasoning into Cadence SMV. The idea is later formalized by [24] and [25]. Our approach of using abstraction is also similar to the work of environment abstraction by [26].

Our work in this paper on guided search has some similarity with directed model checking [27], in the sense that only a subset of the state space is explored to reach certain special states. The difference is that we use guided search in the hierarchical protocol trying to match the error trace from an abstract protocol, while directed model checking employs heuristics, e.g. distance estimation to error states, for fast error discovery.

In predicate abstraction and counterexample guided refinement, there also exists the problem of checking an abstract counterexample trace. There has been a large body of research who uses symbolic methods to solve the problem, e.g. converting the error trace justification to a satisfiablity problem. Our heuristics are believed to be helpful for complexity reduction regardless of the specific verification approach taken.

6 Conclusions and Future Work

We have presented an interface aware guided search method to mechanize part of the compositional approach for verifying hierarchical cache coherence protocols. Given an error trace from an abstract protocol, our approach tries to find a stuttering equivalent execution in the concrete protocol. The experiments on three hierarchical protocol benchmarks with realistic features show that our approach is very efficient in identifying all of the spurious/genuine bugs. Also, in most of the spurious errors it can precisely report the problematic rules. For future work, we plan to investigate how to integrate the automatic refinement of abstract protocols, to our mechanization approaches.

References

[1] J. A. Darringer. Multi-Core Design Automation Challenges. In *Design Automation Conference*, pages 760–764, 2007.

[2] Exploiting Concurrency Efficiently and Correctly – (EC)^2. CAV 2008 Workshop.

[3] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou. Reducing Verification Complexity of a Multicore Coherence Protocol Using Assume/Guarantee. In *Formal Methods in Computer Aided Design*, pages 81–88, 2006.

[4] 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.

[5] 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.

[6] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[7] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, pages 677–691, 1986.

[8] L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability. In *Computer Aided Verification*, pages 17–36, 2002.

[9] K. L. McMillan and N. Amla. Automatic Abstraction Without Counterexamples. In *Technical Report of Cadence*, 2003.

[10] 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.

[11] G. Gopalakrishnan and W. Hunt. Formal Methods in Industrial Practice: A Sampling. In *Formal Methods in System Design*, 2003.

[12] 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.

[13] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Int'l Symposium on Computer Architecture*, 1984.

[14] M. Talupur and M. Tuttle. Going with the Flow: Parameterized Verification using Message Flows. In *Formal Methods in Computer Aided Design*, 2008.

[15] J. Bingham. Automatic Non-interference Lemmas for Parameterized Model Checking. In *Formal Methods in Computer Aided Design*, 2008.

[16] L. Lamport. *What Good is Temporal Logic*. Elsevier, 1983.

[17] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[18] 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.

[19] S. German. Tutorial on Verification of Distributed Cache Memory Protocols. In *Formal Methods in Computer Aided Design*, 2004.

[20] http://www.cs.utah.edu/formal_verification/mtv08-submission.

[21] 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.

[22] K. L. McMillan. Circular compositional reasoning about liveness. In *International Conference on Correct Hardware Design and Verification Methods*, 1999.

[23] 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.

[24] S. Krstić. Parameterized system verification with guard strengthening and parameter abstraction. In *Automated Verification of Infinite-State Systems*, 2005.

[25] 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.

[26] S. K. Lahiri and R. E. Bryant. Constructing Quantified Invariants via Predicate Abstraction. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2004.

[27] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit-state model checking with HSF-SPIN. In *SPIN Workshop*, pages 57–79, 2001.