VERIFICATION OF HIERARCHICAL CACHE COHERENCE PROTOCOLS FOR FUTURISTIC PROCESSORS

by

Xiaofang Chen

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2008

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Xiaofang Chen

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Ganesh L. Gopalakrishnan
	Steven M. German
	Ching-Tsun Chou
	John B. Carter
	Rajeev Balasubramonian

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Xiaofang Chen in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Ganesh L. Gopalakrishnan
Chair: Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Director

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

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, as distant memory accesses cost thousands of cycles. To combat this problem, one must design aggressively optimized cache coherence protocols. This introduces two problems for futuristic cache coherence protocols which will be hierarchically organized for scalable designs: design correctness and hardware implementation correctness. Experiences show that monolithic verification will not scale to hierarchical designs and implementations. Hence there exist two unsolved problems for futuristic cache coherence protocols: (i) handle the complexity of several coherence protocols running concurrently, i.e., hierarchical protocols, and (ii) verify that the RTL implementations correctly implement the specifications.

Our thesis is that formal methods based on model checking and assume guarantee verification methods can be developed to substantially ameliorate these problems faced by designers. More specifically, to solve the first problem, we develop assume guarantee reasoning to decompose a hierarchical coherence protocol into a set of abstract protocols. The approach is conservative, in the sense that by verifying these abstract protocols, the original hierarchical protocol is guaranteed to be correct with respect to its properties. For the second problem, we develop a formal theory to check the refinement relationship, i.e., under which conditions can we claim that an RTL implementation correctly implements a high level specification. We also develop a compositional approach using abstraction and assume guarantee reasoning to reduce the verification complexity of refinement check. Finally, we evaluate the solutions in collaboration with industry and partly mechanize the refinement check.

We propose research that will lead to a characterization of the proposed mechanisms through several verification tools and protocol benchmarks. For high level modeling and verification, we show that for three hierarchical protocols with different features which we developed for multiple chip-multiprocessors, more than a 20-fold improvement in terms of the number of states visited can be achieved. For refinement check, we show that for a driving coherence protocol example with realistic hardware features, refinement check can find subtle bugs which are easy to miss by checking coherence properties alone. Furthermore, we show that for the protocol example, our compositional approach can finish the refinement check within 30 minutes while a state-of-art verification tool in industry cannot finish in over a day. Finally, we extend a hardware language and a tool, to mechanize the refinement check process. In summary the research accomplished in this work substantiates our thesis statement with a body of results generated using tools that were constructed during this research.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 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 some on-die interconnect with large caches or even addressable memories inside each set of CPUs. Two classes of related 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 nontrivial relationships with other protocols, e.g., *hardware transaction memory* protocols [28, 106], and *power management protocols* [40]. For example, when a hardware transaction wins and commits, the cache lines affected by the losing transaction must all, typically, be invalidated. We will for now consider only coherence protocols, assuming the existence of suitable methods to decouple and verify related protocols such as hardware transaction memory protocols.

Future processors will employ extremely complex cache coherence protocols, to ensure high performance, and to accommodate manufacturing realities such as unordered coherence message networks, etc. For example, multicore chip wiring is an important topic which relates to cache coherence protocols. There will be multiple wire types characterized by speeds and availability numbers on multicore chips, including number available, ease of routability, and so on. There have been many recent studies [39,40,108] that show the advantages of judiciously employing fast but limited quantity wires for critical protocol messages, and slow but abundant wires for other messages. The allocation of wires in this manner will, unfortunately, make the protocols very complex. For instance, the resulting protocols will in most cases be unable to exploit the arrival order of the messages.

Because of the high performance demands placed on the shared memory subsystems, aggressively optimized cache coherence protocols will be employed for future multi-processors. Although there is a debate whether coherence protocols will be enforced globally in the system after 10 years when the number of cores move into the hundreds and the size of memory hits 512GB, there is no doubt that coherence protocols will still be employed locally, and scalable designs in this space will employ hierarchical protocols. For example, in [16], it is predicted that more flexible or even reconfigurable data coherency schemes will be employed for future processors. These protocols will be hierarchically organized, with the use of different protocols for different levels of the hierarchy. Such protocols will individually be quite complex, which makes it virtually impossible to debug these protocols through testing alone. On the other hand, next to floating-point units, memory system bugs including cache coherence protocol bugs are one of the most insidious, very difficult to work around through microcode patches, due to the fast hardwired logic used. They can also lead to software vulnerabilities, security holes, or even major chip recalls.

For the verification of cache coherence protocols, it can be done at two stages: presilicon and postsilicon. During the presilicon stage, one tests devices in a virtual environment with sophisticated simulation, emulation and formal verification tools. After the first silicon is ready, postsilicon validation is done in a system environment to flush out bugs missed in presilicon process. Postsilicon validation tests occur on actual devices running at speed in commercial, real world system boards using logic analyzer and assertion based tools. Typical postsilicon errors are from interactions between different components in very improbable corner cases [123]. In this work, we will only focus on presilicon cache coherence protocol verification.

Although there have been many previous hierarchical protocols, e.g., non-CMP [29, 59,65,86,88] or CMP [19,138], 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 decomposed verification is not entirely sufficient, as there is no theoretic or formal reasoning to ensure that the decomposition will not miss any bugs.

1.1 Verification of Cache Coherence Protocols

Currently a common approach to verification of cache coherence protocols is to verify either the high level protocol descriptions (or *specifications*), or the RTL implementations. For verification of high level 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 [50] or TLA+ [84], and exploring the reachable states through explicit state enumeration.

Symbolic methods, e.g., BDD [31] or SAT [129, 142], are yet to succeed for the verification of cache coherence protocols. For example, in [101], it was reported that a SAT solver fails to handle a modest benchmark protocol with about $40,000$ state variables in a few pages of description, while the same set of techniques can handle a microprocessor model with on the order of one million state variables. One possible reason is that most of the state variables are relevant in protocol property verification, while only a small fraction of the state variables are relevant in the CPU property verification. This shows that BDDs and SAT, which in almost all cases can enable scalable formal verification methods, can be ineffective when it comes to coherence protocols – protocols where localized reasoning is difficult without specialized and hand-crafted localization abstractions.

Accumulated experience, starting from early work of Yang et al. on UltraSparc-1 [141] to more modern ones [21, 41, 57], 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 [10, 61, 141]. However, these monolithic techniques will not be able to handle the very large state space of hierarchical protocols. Compositional verification techniques are *es-*

sential. That is, *scalable* formal verification of hierarchical cache coherence protocols as well as shared memory consistency protocols is crucial to the advancement of multicore processors as a viable computing platform. This dissertation contributes to ensuring the correctness of these protocols through scalable formal verification methods.

In practice, verification of coherence protocols only at the high level is insufficient. The semantic gap between high level protocol descriptions and their hardware implementations is nontrivial. One atomic step of a high level description is typically realized through a *hardware transaction* (or simply "transaction"), which is a multicycle activity consisting of one or more concurrent steps in each clock cycle. For example, an atomic step "invalidate a cache line" in the high level description can be implemented by three steps: (i) "calculate the victim addresses," (ii) "form the invalidation messages," and (iii) "send them out." This is to maximize overlapped computation, pipelining, and to take advantage of internal buffers and split transaction buses, etc. Moreover, a sequence of high level atomic steps may actually be realized through multiple transactions whose time intervals may overlap. For example, the third cycle of a transaction may consist of two concurrent steps, and this may happen in the same clock cycle as the first cycle of another transaction consisting of two concurrent steps. Unfortunately, there have been very few attempts at bridging the verification gap between high level protocol descriptions and their clocked implementations. Techniques to verify the gaps between protocol specifications and implementations are essential to be developed.

1.2 Problem Statement

In summary, for futuristic hierarchical cache coherence protocols, two problems must be solved. First, for high level specification protocols, it is possible that they cannot be formally verified because of the cartesian product of the state space of several coherence protocols running concurrently. This is true, as we will show in Section 3.1 for a protocol used for multiple chip multiprocessors (M-CMPs). This protocol models one address that is accessed by three NUMA (NonUniform Memory Access) chip multiprocessors (CMPs). We show that the high level specification protocol cannot be verified monolithically, using existing explicit state enumeration techniques, even on resourceful computing platforms (e.g., 18GB of memory) and compressed state representations (e.g., 40-bit of hash compaction).

Second, for RTL implementations of industrial protocols, even if we can formally verify the correctness of the high level specifications, bugs could be introduced in obtaining the RTL implementations. In Section 4.6, we will show that for an RTL implementation of a simple coherence protocol with some realistic features, although the coherence properties can be verified, the RTL implementation can still be buggy.

In modern approaches to formal verification, many practical problems are handled by creating overapproximated versions of a system design. This is akin to reasoning about obstacle avoidance in a robot's landscape by assuming that all objects are approximated to be spheres and then calculating the separation between objects. These are *conservative* approximations in that if the calculation predicts the situation to be safe, it indeed is safe. A central theme in this work is to create conservative abstractions in order to reduce the verification burden.

More specifically, for the first problem, i.e., verifying hierarchical protocols in the high level descriptions, as currently there is no publicly available hierarchical cache coherence protocol with reasonable complexity, we first develop several hierarchical protocols for M-CMPs with different features as the benchmarks. We then develop a divide-and-conquer approach to decomposing a hierarchical coherence protocol into a set of abstract protocols with smaller verification complexity. The basic idea includes abstraction and assume guarantee reasoning. The form of assume guarantee reasoning used in our work is circular: a component A in the system is verified under the assumption that the component B behaves correctly, and symmetrically, B is verified assuming the correctness of A. This form of assume guarantee principles was first advocated by [105] and later developed by [9, 13, 98], and also by [12, 137] in a real-time setting. The assume guarantee paradigm provides a systematic theory and methodology for ensuring the soundness of the circular style of postulating and discharging assumptions.

Our divide-and-conquer approach is conservative, in the sense that by verifying the abstract protocols, the original hierarchical protocol is guaranteed to be correct with respect to its coherence properties. Also, for verification of hierarchical protocols which use snooping protocols or the noninclusive cache organization (see Section 3.1), we use history variables [43, 47] in an assume guarantee manner, together with the divide-and-conquer approach. In more detail, we introduce a set of *auxiliary* variables to the protocols, and the value of each auxiliary variable is a function of those of the state variables already in the protocols. Furthermore, for hierarchical protocols, we propose that they be modeled and designed in a loosely-coupled manner, so that the verification complexity can be reduced to that of a nonhierarchical coherence protocol.

For the second problem, i.e., checking whether a hardware implementation of a coherence protocol meets its high level specification, we develop a formal theory of refinement check which defines a set of sufficient conditions under which an RTL implementation can be claimed to correctly implement a specification. In more detail, we use a *transaction* to group the implementation steps, to correspond to a single step in the high level specification. A transaction is defined to be a multiple cycle activity. Transactions, as we define, are not *atomic* activities in the sense of transactional memory [28, 106]. According to our usage, a transaction can be executed in multiple clock cycles.

In modern formal verification approaches, one verifies that a given design implements another by creating correspondences between "user visible variables," in our case these variables are the variables which appear in both the specification and the hardware implementation models. These are called *joint* variables by us. Given a specification and implementation models, we first combine them in a way such that whenever an implementation transaction is executed, the corresponding specification step is also executed. The basic idea of refinement check is that for each joint variable, if all the transactions that can write to the variable have either not started or already finished executing, then the value of the joint variable must be equivalent in both models.

We also develop a compositional approach to reducing the verification complexity of refinement check. The basic idea is using abstraction and assume guarantee reasoning. For abstraction, we remove certain details of the model to obtain a simplified model. This type of simplification is also conservative, in that if the simpler model can be verified, the original complex model can also be verified. Assume guarantee reasoning is a form of induction, in that it introduces assumptions and at the same time proves them. Furthermore, we develop a practical tool to check the refinement relationship, in collaboration with researchers from IBM.

We will demonstrate the advantages of our techniques through several verification tools and coherence protocol benchmarks, with the experimental results shown in Section 3.3.4 and Section 4.6.5. For high level modeling, we show that for three 2-level hierarchical protocols developed for M-CMPs, our approach can reduce more than 95% of the explicit state space. For refinement check, we show that for a driving cache coherence protocol example which has some realistic hardware features, although the high level specification and the RTL implementation satisfy the coherence properties check individually, our refinement check still finds three subtle bugs in the implementation. We believe that these bugs are easy to miss by writing assertions, as the process of verification by manually writing a collection of assertions is unreliable, while the refinement check is an automatic method for constructing such assertions. Furthermore, we show that for the protocol example, when the datapath of the cache line is of 10 bits in width, our compositional approach can finish the refinement check within 30 minutes while a state-of-art verification tool in industry could not finish in over a day.

The datapath of cache lines is used to model the cache lines for data related coherence properties. Basically, there are two categories of coherence properties: controller logic related properties and cache data related properties, to be described in Section 2.4. For high level protocol specifications, usually 1-bit of cache line is modeled for the sake of simplicity, while in real hardware the datapath width can be, e.g., 16 bytes. In general, it is difficult to figure out the smallest datapath width of cache lines in either a high level protocol specification or an RTL protocol implementation model, so that for the properties which hold in the model with those cache lines, they will hold in the real hardware with any datapath width. That is the basic motivation for our driving protocol example to model the cases of 1-bit and 10-bit for the datapath width.

1.3 Thesis Statement

Compositional formal verification techniques for multicore shared memory system protocols, both for the high level specifications and their hardware implementations, *can help scale up the verification capability.* In order to prove this thesis statement, we perform two classes of case studies to demonstrate the effectiveness our approaches.

For high level specification of a hierarchical coherence protocol, we decompose the hierarchical protocol into a set of abstract protocols with smaller verification complexity, by overapproximating it with different components. Also, we constrain the overapproximation by using counterexample guided refinement. Based on assume guarantee reasoning, our approaches are guaranteed to be conservative. Furthermore, by using a better interface characterization of hierarchical protocols, our approaches enable the possibility of verifying hierarchical protocols one level at time. Thus it can greatly scale the verification capabilities. Finally, with history variables our approaches can be applied to noninclusive and snooping hierarchical coherence protocols which cannot be decomposed in a straightforward way.

For hardware implementations of cache coherence protocols, we develop a set of sufficient conditions to show that the implementations meet the high level specifications. We relate one step in the high level specification to a multi-step transaction in the implementation. Being aware of the *joint* variables mapping, our approaches can check the joint variables equivalence. Also, we develop a modular refinement verification approach by developing abstraction and assume guarantee reasoning principles that allow implementation steps realizing a single specification step to be situated in sufficiently general environments.

1.4 Contributions

In this dissertation, we motivate, present, and evaluate the idea of scalable verification techniques. The primary contributions of this dissertation include the following.

- Develop several multicore hierarchical cache coherence protocols with reasonable complexity and realistic features as hierarchical coherence protocol benchmarks. Based on these protocols, develop a compositional infrastructure to decompose hierarchical protocols into a set of abstract protocols with smaller verification complexity. The approach is conservative for safety properties, and it is practical.

The techniques include abstraction, counterexample guided refinement and assume guarantee reasoning.

- Decompose hierarchical coherence protocols one level at a time, using the same compositional infrastructure. To employ this approach, propose that hierarchical coherence protocols be designed and implemented in a loosely-coupled manner. Also, use history variables in an assume guarantee manner, for hierarchical noninclusive and snooping protocols.

- Develop guided search to automatically identify whether an error trace from an abstract protocol corresponds to a genuine error in the original hierarchical protocol.

- Develop a transaction based modeling and verification approach to checking whether a hardware implementation of a cache coherence protocol correctly implements its high level specification. Develop a refinement theory and a monolithic approach for the verification. Also, develop a compositional approach using abstraction and assume guarantee reasoning to reduce the refinement check complexity.

- Present a comprehensive language and tool support both for the abstraction process in the hierarchical protocol verification, and for checking the refinement relationship between hardware implementations and specifications. Evaluate all these techniques to show their potential in reducing the verification complexity and detecting implementation bugs.

A key feature of our approach is that it requires only conventional model checking tools for supporting the compositional approach developed at the high level, and requires only existing HDL model checking frameworks for supporting the refinement check and the compositional approach at the implementation level.

1.5 Outline

This dissertation is organized into five chapters. The next chapter discusses the fundamentals of cache coherence protocols, and the problems existing for future hierarchical cache coherence protocols. Chapter 3 first presents three hierarchical coherence protocols, and then presents two compositional approaches which can greatly reduce the verification complexity. Chapter 4 first presents the refinement theory, and then presents an implementation of the refinement check. It also presents a compositional approach to reducing the verification complexity. In both chapters, the soundness proofs of our approaches are provided. Finally, Chapter 5 summarizes this dissertation and points out a few possible directions for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we will describe the background of cache coherence protocols and related work in the verification of cache coherence protocols. We will first describe the basic types of cache coherence states, common cache coherence protocols, caching hierarchies, and common coherence properties to be verified. After that, we will present some related work in the verification of hierarchical coherence protocols in the high level specifications, and the refinement check between protocol specifications and their hardware implementations.

2.1 Cache Coherence States

Coherence protocols use protocol states to keep track of read and write permissions of blocks present in processor caches. This section describes the well-established MSI and MESI protocol states [20, 113] that provide a set of common states for reasoning about cache coherence protocols. These protocols will be used in our hierarchical coherence protocol benchmarks to be described in Section 3.1. Other than these protocols, the MOESI protocol [46, 134] is well studied and also employed in commercial products, e.g., AMD Opteron [77].

2.1.1 The MSI Protocol

The MSI protocol is the basic cache coherence protocol used in multiprocessor systems. Each cache block can have one of the following three states:

- **M**odified: Also called dirty, meaning that only this cache has a valid copy of the block, and the copy in main memory is stale. A processor may read or write a modified copy, and the cache block needs to be written back before eviction.

- **S**hared: The block is unmodified and valid in the cache. A processor may read the block, but may not write it. The cache can evict the block without writing back to the backing storage.

- **I**nvalid: This block is invalid, or the block is not found in the cache. The block must be fetched from memory or another cache before it can be read or written.

These three states are used to directly enforce the coherence invariant by (i) only allowing a single processor in the modified state at a given time, (ii) allowing multiple processors in the shared state concurrently, and (iii) disallowing other processors in the shared state while a processor is in the modified state. Table 2.1 summarizes the basic operations of a processor of the MSI coherence protocol. In practice, there can be variants of the above simple MSI protocol, e.g., a processor is in the modified state while other processors are in the shared state. This can happen when a processor changes the cache to the modified state before receiving all the invalidation acknowledgments from those shared caches.

2.1.2 The MESI Protocol

The **MESI** protocol is a cache coherence protocol in which every cache line can have one of the four following states: **M**odified, **E**xclusive, **S**hared, **I**nvalid. The states of M, S and I are the same as in the MSI protocol. The state of E means that the cache block is only present in the current cache, but is *clean*, i.e., the data match those in the main memory. In order to introduce this extra state in the MESI protocol when a processor requests a new block, usually it has to evict a block currently in the cache. The effort

Table 2.1. MSI state transitions

	PROCESSOR ACTIONS			REMOTE REQUESTS	
STATE	READ	WRITE	EVICTION	READ	WRITE
M	HIT	HIT	WRITE BACK $\rightarrow$ I	SEND DATA $\rightarrow$ S	SEND DATA $\rightarrow$ I
S	HIT	WRITE REQ $\rightarrow$ M	SILENT DROP $\rightarrow$ I	ACK	ACK $\rightarrow$ I
I	READ REQ $\rightarrow$ S	WRITE REQ $\rightarrow$ M	NONE	NONE	NONE

required to evict a block depends on the coherence state of the block. For example, most protocols require a writeback to memory when evicting a block in modified state, and allow for a silent eviction of blocks in the shared state, as is the case in the MSI protocol.

The advantage of the MESI protocol vs. the MSI protocol lies in the fact that if the current cache has the exclusive state, it can silently drop the cache line without issuing the expensive writeback operation. On the other hand, when a data resides only in the main memory, and there is a read miss in one thread or agent, an exclusive state cache block can be supplied. One or two instructions later, when the thread or agent needs to write a new value to the same variable, there will be no write miss. In addition, the read-before-write operations, e.g., $x := x + 1$;, can be handled efficiently by the MESI protocol. Table 2.2 summarizes the basic operations of a processor of the MESI coherence protocol.

2.2 Common Cache Coherence Protocols

Coherence protocols encode the permissions and other attributes of blocks in caches with the coherence states. With these states, there are several classes of protocols which use different techniques to track the caching status [68].

- *Snooping* – Every cache that has a copy of the data form a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept. The caches are usually on a shared- memory bus, and all cache controllers

Table 2.2. MESI state transitions

	PROCESSOR ACTIONS			REMOTE REQUESTS	
STATE	READ	WRITE	EVICTION	READ	WRITE
M	HIT	HIT	WRITE BACK $\rightarrow$ I	SEND DATA $\rightarrow$ S	SEND DATA $\rightarrow$ I
E	HIT	HIT $\rightarrow$ M	SILENT DROP $\rightarrow$ I	ACK $\rightarrow$ S	ACK $\rightarrow$ I
S	HIT	WRITE REQ $\rightarrow$ E	SILENT DROP $\rightarrow$ I	ACK	ACK $\rightarrow$ I
I	READ REQ $\rightarrow$ S	WRITE REQ $\rightarrow$ E	NONE	NONE	NONE

monitor or *snoop* on the bus to determine whether or not they have a copy of the block that is requested on the bus.

- *Directory-based* – The sharing status of a block is kept in just one location, called the *directory*. All coherence requests will first go to the directory. The directory decides to reply or forward the requests, based on the sharing status of the block. There are two kinds of directories: centralized and distributed directories. The latter can be implemented, e.g., using linked lists [74, 111, 139].

- *Token-coherence* – Token coherence [91] employs token counting, i.e., each block has a fixed number of tokens, to enforce coherence. A processor is allowed to read a block only when it holds at least one token, and it is allowed to write only when it holds all of the block's tokens.

2.2.1 Snooping Protocols

Snooping protocols [33, 35, 46] are commonly used in shared-memory multiprocessors, especially in the level of caches closest to CPUs. The key characteristic that distinguishes snooping protocols from other coherence protocols is their reliance on a bus or a virtual bus for interconnect. Bus-based interconnect provides two properties for coherence protocols: (i) all requests that appear on a bus are visible to all components connected to the bus, and (ii) all requests are visible to all components in the same order, i.e., the order in which they gain the bus access. In essence, a bus provides low-cost atomic broadcast of requests.

The primary advantage of snoop-based multiprocessors is the low average miss latency, especially for cache-to-cache misses. This is because a request is sent directly to all the other processors and/or memory modules associated with the bus. The responder can immediately send the response. Other than this, bus-based snooping is relatively simple. Also, shared-buses are usually cost-effective and they are complexity-effective to implement coherence.

As for disadvantages, the biggest problem of bus-based protocols is scalability. More specifically, the bus is a mutually exclusive resource and only one processor can transmit at any given time. All processors need to participate in an arbitration phase before accessing the bus. Also, the bus clock cycle must be long enough so that signals can propagate to the entire bus. Moreover, the snooping protocols are still by nature broadcast-based protocols, so the bandwidth requirements will increase when the number of processors increase. Even after removing the bottleneck of a shared-wire bus or virtual bus, this broadcast requirements still limits the system scalability.

To increase effective system bandwidth of a bus, many enhancements have been proposed to snooping protocol designs. For example, split transaction protocols [35, 46] allow the requesting processor to release the bus while waiting for the responses. This mechanism can increase the bus efficiency by pipelining multiple requests. Other systems implement virtual buses [36, 130] using distributed arbitration, point-to-point links, and dedicated switch chips. Also, destination-set prediction [26, 92, 131] is proposed to reduce the bandwidth requirements of snooping. With these optimizations, many current snooping protocols can create high-bandwidth systems with dozens of processors.

2.2.2 Directory Based Coherence Protocols

Directory protocols allow cache coherence to scale beyond the number of processors that can be sustained by a bus. The idea is to maintain all the cache status explicitly in a directory where requests can go and look it up. Directory protocols avoid broadcast and a totally-ordered interconnect in exchange for adding indirection latency to some misses. As a result, they can support up to thousands of processors [69].

In more detail, a directory protocol works in such a way that whenever a processor issues a coherence request, it will send the request only to the home memory module for the block. The home memory module contains the directory which has the information about the cache status of the block. When the home memory module receives the request, it uses the directory information to decide whether to respond directly (or with data), or forward the request to other processors. One simple way to store the cache status in the directory is to use a bit vector, e.g., one bit per processor, for the sharers and the owner of the cache block. Alternatively, some directory schemes use entries at the memory and caches to form a linked list of processors sharing the block [74]. Linked list based directories can often scale better than centralized directories, at the price of longer average access time. Reference [119], it studies a distributed directory with a singly linked list directories, and it shows that the validation of such protocols is challenging because of the distributed algorithm used to maintain the linked lists. Many experimental [11, 64, 81, 86] and commercial [30, 59, 62, 77, 78, 85, 88, 138] multiprocessors employ directory protocols in either one or multiple levels.

One of the primary disadvantages of directory protocols is the relatively high latency of cache-to-cache misses. In more detail, the extra interconnect traversal and directory access is on the critical path of cache-to-cache misses. In some systems, the directory lookup is similar to that of main memory DRAM. Thus, placing this lookup on the critical path of cache-to-cache misses can increase the latency. Also, additional storage is needed to store the information of cache status in the directory.

Recently, a number of protocols have been proposed that combine snooping and directory protocols [92, 94]. Their goal is to achieve the lower latency of requests associated with snoop protocols with maintaining the lower bandwidth requirements of directory protocols. Also, there have been many studies [40, 108] that show the advantages of judiciously employing fast but limited quantity wires for critical protocol messages, and slow but abundant wires for other messages.

2.2.3 Token Coherence Protocols

The basic idea of token coherence [91, 93] is that simple token counting rules can ensure that the memory system behaves in a coherence manner. Token counting specifies that each block of the shared memory has a fixed number of tokens and that the system is not allowed to create or destroy tokens. A processor is allowed to read a block only when it holds at least one of the block’s tokens, and a processor is allowed to write a block only when it holds all of its tokens. These simple rules prevent a processor from reading the block while another processor is writing the block, ensuring coherence behavior at all times.

Compared with two other approaches to coherence, i.e., snooping and directory protocols, token coherence can adapt itself to capture many of the attractive attributes.

In [91], a broadcast version, a directory like, and a multicast version of token coherence protocols are proposed, each with different features. With these features, token coherence can achieve low-latency and direct processor-to-processor communication which is similar to snooping protocols. Also, it can be bandwidth efficient and does not require a bus or other totally-ordered interconnect, similar to directory protocols.

Token coherence prevents starvation by using persistent requests. A processor invokes a persistent request when it detects potential starvation. Persistent requests are able to obtain data and tokens even when conflicting requests occur. This is because once such requests are activated, they persist in forwarding data and tokens until the requests are satisfied. While conceptually appealing, this scheme has some potential difficulties in implementation. For example, one such difficulty is that persistent requests need an arbiter to avoid livelocks. However, the proposed distributed-arbitration [91] relies on point-to-point ordering of the interconnect.

In [95], token coherence was extended to M-CMP shared memory systems, such that the coherence protocols are flat for correctness, but hierarchical for performance. However, the drawbacks inherent in token coherence protocols still exist. For example, every line needs token storage in main memory. With CMPs, the protocol must be extended with additional storage and states to allow a local cache in the CMP to supply data to another local cache. Also, a nonmodified cache line which has tokens associated with it cannot be silently dropped. Some of these issues are addressed in [95]. As far as we know, token coherence has not been used in any commercial processor.

2.3 Caching Hierarchies

In shared memory systems, multi-level caches are often employed, with small fast caches backed up by larger slower caches. Figure 2.1 shows a simple example of two levels of caches in a CMP. In this example, the CMP has two processors each with an L1 cache, and the L2 cache is shared by the two processors.

Referring to Figure 2.1, we now introduce some terminology. The term *inclusive* means that the content of the L1 cache is a subset of that of the L2 cache. *Exclusive* means that any block that is present in an L1 cache cannot be present in the L2 cache.

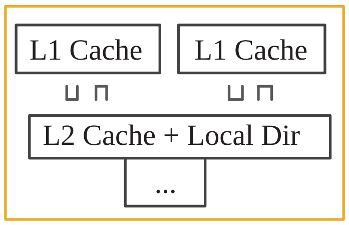


Figure 2.1. A CMP with two levels of caches.

Noninclusive lies between inclusive and exclusive: overlaps and without containment are allowed. For illustration, some processors of the Intel Pentium family [8] use noninclusive caches, and processors of AMD Athlon and Operton [6, 77] use exclusive caches.

There are some tradeoffs between inclusive and exclusive caches. For inclusive caches, the local directory at the L2 cache knows which L1 cache(s) have valid copies. Upon a cache miss in an L1 cache that hits in the L2, the cache controller only needs to copy the data to the missing L1 cache. However, for exclusive caches, the cache controller also needs to evict the data from the L2 cache. On the other hand, the effective cache size of exclusive caches can be the sum of the L1 and L2 caches. This is different from inclusive caches, where the duplication of the cache contents may effectively reduce the size of the combined caches. Also, to evict a line from an L2 cache in an inclusive cache, one has to do a *backward invalidation* of the L1 caches. That is, the same block must be evicted from all the L1 caches of the cluster. This is unnecessary for exclusive caches.

As for the verification complexity of hierarchical cache coherence protocols, we think that inclusive caches are easier to verify than exclusive or noninclusive caches. This is because for multicore coherence protocols with inclusive caches, the cache protocol which is used among the CMPs can simply check the L2 cache of a CMP to know whether the CMP has any valid copy. This is especially true when the cache protocol used inside a CMP does not employ a directory based protocol (e.g., instead employ a snooping protocol), or the directory entry for the cache block is not available. For our benchmark protocols to be described in Section 3.1, we will cover these cases. We show that to verify noninclusive multicore protocols, other techniques are needed than the compositional approach we develop for the inclusive protocols.

2.4 Common Coherence Properties to Be Verified

Pong and Dubois surveyed the basic verification techniques and the types of cache coherence protocol errors in [118]. Basically, there are two types of coherence properties to be verified: *safety* properties (bad things will never occur) and *liveness* properties (good things will occur in the future). These properties include:

- **Data consistency**. In a cache coherent system, data consistency among multiple data copies is typically enforced by allowing one and only one store in progress at any time for each block [124]. Concurrent accesses to the same block can be executed on different data copies but must appear to have executed atomically in some sequential order. The cache protocol must always return the latest value on each load.

- **Incomplete protocol specification**. When possible state transitions have been omitted in the protocol specification, it may happen that some protocol component receives a message which is not specified in its current state. Such *unexpected* message reception is an error condition. Since the message is not specified, the subsequent behavior of the protocol is not defined and is unpredictable.

- **Absence of deadlock and livelock**. A deadlock occurs when the protocol enters a state without possible exits. Deadlock must be avoided because the system is blocked forever. On the other hand, a livelock is a situation where protocol components keep exchanging messages but the system is not making any useful progress.

In our work, we mainly focus on the verification of safety coherence properties, i.e., the data consistency properties. Interested readers can refer to [17, 22, 100] for more work on the verification of liveness coherence properties.

2.5 Related Work

2.5.1 Verifying Hierarchical Protocol Specifications

Many previous hierarchical protocols, e.g., non-CMP [29, 59, 65, 86, 88] or CMP [19, 138], use directories at one or multiple levels of the hierarchy. For example, in Piranha [19] two levels of caches are used in a CMP, and the L2 cache is kept as noninclusive of the L1 caches on the chip. Also, both the intracluster and the interchip protocols use directory protocols. In Power4 [138], there are three levels of caches, and the directory and the controller of the L3 cache are on the chip while the actual L3 cache is on a separate chip. 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 theoretical or formal reasoning underlying the decomposition to ensure that no bugs will be missed.

For coherence protocols which only consider one level (or nonhierarchical protocols), various techniques have been proposed for the verification. These techniques basically include explicit state enumeration [133] and symbolic model checking [102]. Optimizations on these techniques include partial order reduction [24], symmetry reduction [25, 72], compositional reasoning [96], predicate abstraction [22, 49, 82], etc. There is also a large body of work on parameterized verification [42, 53, 54, 100, 117]. However, none of them have shown the capability to verify hierarchical coherence protocols with realistic complexity.

McMillan et al. [97, 102] modeled a 2-level MSI coherence protocol based on the Gigamax distributed multiprocessor [102]. In the protocol, bus snooping is used in both levels. SMV [97] was used to verify this protocol with two clusters each having six processors, for both safety and liveness properties. In terms of the complexity, many realistic details are abstracted away from the protocol so that the protocol does not have the typical complexity that a hierarchical coherence protocol has. Thus, it is not clear whether directly employing SMV can verify hierarchical protocols with realistic features.

Shen et al. [126] proposed an adaptive protocol called `Cachet` for distributed shared memory systems. Cachet integrates three microprotocols, each of which was optimized for a particular memory access pattern. The correctness of the protocol is enforced by two aspects. First, Cachet implements the Commit-Reconcile & Fences (CRF) [127], which exposes a semantic notion of caches, and decomposes load and store instructions into finer-grain operations. CRF exposes mechanisms that are needed to specify protocols precisely. It 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 [125]. For verification of cache coherence protocols, their approach concludes that any coherence protocol following the CRF memory model is correct by construction, including the adaptive protocol. Compared with their work, our approach is more general and it can be applied to other protocols following other memory models as well.

As said earlier, currently there are no publicly available hierarchical protocols with reasonable complexity. So we developed three 2-level coherence protocols used in M-CMP systems, each with different features. These protocols are perhaps the largest publicly available hierarchical protocols as far as we know. The protocol of MSI [20] or MESI [113] is used in each level. As to be described in Section 3.1, our protocols include many features such that even the protocol used in one chip (we also call it a *cluster*) is much more complex than popular academia benchmarks such as FLASH [81] and German [55]. Also, realistic features including silent drop on nonmodified cache lines and unordered network channels are modeled. Finally, snooping and directory based protocols are modeled, and the cache hierarchies including inclusive and noninclusive cases are modeled as well.

As said earlier, separable verification by manually abstracting other levels of a hierarchical protocol and considering only one level is not entirely sufficient. As far as we know, our work is the *first* which is able to handle these hierarchical protocols with complexity similar to that in the read world. Based on our benchmark protocols, we create a set of abstract protocols where each abstract protocol simulates the given hierarchical protocol. After these abstract protocols are created, verification consists of dealing with them in an assume guarantee manner, refining each abstract protocol guided by counterexamples.

Our approach of using abstraction, assume guarantee reasoning and counterexample guided refinement, derives the basic ideas from the work of Chou et al. [42] on parameterized verification for nonhierarchical cache coherence protocols. Their work is again attibuted to McMillan [99, 100], who added support for this style of reasoning into Cadence SMV. The idea is later formalized by Krstić [79] and Li [87]. Our approach of using abstraction is also similar to Lahiri and Bryant in [82]. The fact is that they use predicate abstraction techniques to automatically construct quantified invariants, while we use model checking to reduce verification complexity.

Also, we use *history variables* [43, 47] in addition to the compositional approach, to deal with noninclusive and snooping based hierarchical protocols. While the use of history variables in program verification goes back several decades, our usage is in the context of assume guarantee reasoning which is based on inductive reasoning.

Finally, we contribute simple but effective heuristics for error trace justification, i.e., when an error is reported from an abstract protocol, how do we identify whether it corresponds to a genuine error in the original hierarchical protocol? The basic idea of our approach is using guided search which has some similarity with directed model checking [51], in the sense that only a subset of the state space is explored to reach certain special states. We believe that this usage is an interesting technical contribution. The difference is that we use guided search in the hierarchical protocol 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 which have used symbolic methods to solve this problem, e.g., converting the error trace justification to a satisfiability problem. We think directly applying symbolic methods on hierarchical protocols for our compositional approach may not work satisfactorily, although further research is necessary to study this problem. Our heuristics are believed to be helpful for complexity reduction regardless of the specific verification approach taken.

2.5.2 Checking Refinement

For refinement check between high level specifications and RTL implementations of cache coherence protocols, Mellergaard et al. [104] presented an approach to model and verify (using theorem proving) digital systems using Synchronized Transitions [132]. There was no attempt to model implementations separately, nor to verify correspondence. However, synchronized transitions, and similar notations including Unity [34], term rewriting systems [15], Murphi, TLA+, and Bluespec [14] reveal the wide adoption of the guard/action notation among designers.

2.5.2.1 The Approach by Burch and Dill

Burch and Dill [32] proposed an approach to model instruction pipelines, and to verify the correspondence between pipelines and higher level instruction sequences, using symbolic execution and ‘flushing’ as a heuristic to build an abstraction map. The details of our approach for showing simulation are different in that: (i) we are not concerned with processing program instructions and the instruction level semantics, (ii) we are concerned about modeling details at the hardware level in terms of signal and variable updates, and variable write conflicts, and (iii) we partition correctness into global properties such as coherence at the specification level, and simulation of specifications by implementations at the next level.

2.5.2.2 The Approach of Aggregation of Distributed Actions

Our notion of transactions is related to the approach of Park et al. based on aggregation of distributed actions [114–116]. The fact is that their work is done between two abstracted protocols which both use interleaving executions, while our work is done between an abstract protocol with interleaving execution and an RTL implementation with concurrent execution. Also, they use theorem proving while we use model checking techniques. So their goal is to develop inductive theorems, while ours is to develop compositional approaches to reduce the verification complexity.

2.5.2.3 The Bluespec Approach

The Bluespec [1, 15, 70] approach proposed by Arvind et al. is probably the most closely related work to our overall approach in refinement check. Both approaches view the specification in terms of interleaved executions. The Bluespec compiler automatically synthesizes the specification into an RTL model, automatic scheduling the implementation actions according to predefined synthesis recipes.

Arvind et al. rely upon the correctness of synthesis tools in order to guarantee the correctness of implementations. The degree of trust one can place upon synthesis tools varies. In many situations, one has to verify the correctness of the synthesis tools themselves, e.g., as done in [109].

Also, automatic synthesis methods may not meet performance goals as well as design constraints that are important to handle in practice. They may also not give designers enough flexibility in handling pre-existing interfaces such as embedded split-transaction buses or packet routing networks around which implementations are designed. By allowing designers to express implementation details in terms of transactions and supporting the verification of refinement, our approach is better suited to many industrial design flows.

2.5.2.4 The Approach of Architecture Description Languages

Our work is similar to the approaches based on ADLs (An Architecture Description Language). ADLs are computer languages used to support architecture-based development, formal modeling notations and analysis, and development tools that operate on architectural specifications are needed. Examples of ADLs include C2 [103], Rapide [89], SADL [107] and so on. Qin et al. [120, 121] proposed to design embedded processors at the architecture/micro-architectural level. An ADL was proposed to model computer microarchitectures and to generate functional simulators automatically. While their work is another indication of the need for customized hardware description languages, our work is aimed at entirely different issues.

2.5.2.5 The Approach of OneSpin Solutions

Our work is also similar to the approach of OneSpin Solutions [52, 140] on "completeness" metrics. Their definition of *completeness* is that verification is complete when every possible input scenario has been applied to the design under verification, and every possible output signal has been shown to have its intended value at every point in time. Basically, they propose that enough properties should be checked on the implementation against the specification. The difference is that their approach needs to manually write those properties, while our approach can automatically generate the refinement assertions. Moreover, we have a compositional approach to reduce the refinement check complexity. Finally, it is not clear from their papers what is the underlying foundation behind their completeness metrics.

2.5.2.6 The Approach of SystemC

Our work is also related to the approach of transaction-level modeling (TLM) [122] and verification library in SystemC (VLC) [7]. The basic idea of transactions in both works is similar. The difference is that a transaction in our work contains multiple transitions, while a transaction is a monolithic implementation in SystemC. In SystemC, a transaction is attached to the implementation using shared variables and signals. Also, TLM collects a set of commonly used hardware components and provides a set of high level interfaces. The advantages of using transactions in SystemC is that with these high level interfaces, people can build a hardware design more easily using these components and start simulation earlier and with fewer details.

However, it is not very clear what the underlying formal theory used in SystemC is from the available documents. Also, the notion of transactions used in their work is not as well specified and formal in terms of characterizing implementation behaviors. It seems that only simulation is used in SystemC, and it looks straightforward to adapt our approach to support such simulation-based testing.

2.5.2.7 Compositional Verification

Our compositional verification approach relies on input variable generalization and assume guarantee reasoning. Many previous approaches to assume guarantee reasoning have been developed for both software and hardware contexts, for example [75, 98]. Although we verify hardware, our model is related to models of concurrent software with interfering operations.

In our approach to abstraction, variables of the model can be abstracted to different degrees at different points in time. At some steps, a variable can be abstracted by input variables, while at other steps, the variable may not be abstracted. This selective abstraction approach has some resemblances to how the degree of input weakenings can be controlled in STE based verification (e.g., as in [67, Page 25]); however, the details are different. The basic idea of our abstraction is also similar to Kurshan's work in localization reduction [80]. In his localization reduction, some program variables are abstracted away with nondeterministic assignments. If a counterexample is found to be spurious, additional variables are added to eliminate the counterexample. The heuristic for selecting these variables uses information from the variable dependency graph [80].

A goal of our work is to develop methods for checking refinement that are computationally efficient and also reduce the manual burden of specifying and verifying the mapping between the implementation and the specification. To reduce the manual effort, we develop a theory of refinement checking that incorporates the notion of multicycle transactions and their mappings onto interleaving specifications.

CHAPTER 3

VERIFICATION OF HIERARCHICAL COHERENCE PROTOCOLS IN THE SPECIFICATIONS

In this chapter, we will present our work to the verification of hierarchical cache coherence protocols in the high level specifications. We will first describe three hierarchical coherence protocols that we developed for benchmarking hierarchical protocols, and then present our compositional approach to verifying them. We will then present an extension to the composition approach which can verify hierarchical protocols one level at time. Finally, we will describe our work which partly mechanizes the compositional approach.

3.1 Benchmarking Hierarchical Coherence Protocols

For the verification of hierarchical cache coherence protocols, as described in Section 1.2, currently there are no publicly available hierarchical protocols with reasonable complexity. In response to this problem, we first develop three 2-level coherence protocols used in M-CMP systems, each with different features. In these protocols, the first employs the inclusive cache hierarchy, the second employs the noninclusive hierarchy, and the third models a snooping protocol in the intracluster level which is the case for most processors nowadays. The protocol of MSI [20] or MESI [113] is used in each level. Also, realistic features including silent drop on nonmodified cache lines and unordered network channels are modeled.

3.1.1 The First Benchmark Protocol

Our first benchmark hierarchical coherence protocol is derived by combining features from the FLASH [81] and DASH [86] protocols. Such a protocol is realistic, as DASH

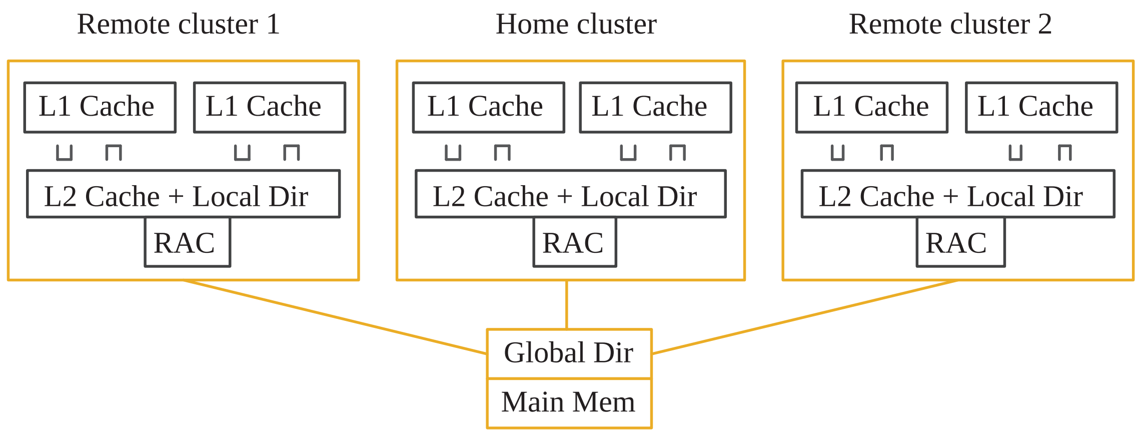


Figure 3.1. A 2-level hierarchical cache coherence protocol.

was originally developed for managing coherence across many clusters. It also renders our hierarchical protocol easy to understand for researchers who might want to attack this verification challenge problem. One address is modeled in our protocol, as is typical in model checking based verification for coherence. Figure 3.1 intuitively depicts the protocol.

As shown in Figure 3.1, the protocol is composed of three nonuniform memory access (NUMA) clusters: one home cluster and two identical remote clusters. Each cluster has two symmetric L1 caches, an L2 cache and a local directory. “RAC” 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 that there is only one memory is a consequence of the 1-address abstraction of our protocol. A typical approach taken during cache coherence protocol verification is to create verification models with one cache line. This is justified based on the fact that the behavior of most cache coherence protocols does not depend on the number of cache lines. Finally, the global directory is to manage data copies on the three clusters.

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

- For each cache line, if it is cached in an L1 cache then it must also be cached in the L2 cache of the cluster, i.e., the *inclusive* property;
- Both levels use MESI, supporting explicit writeback and silent drop. That is, a nonmodified line can be discarded at any time, without informing the local or the global directory.
- Both levels use unordered network channels; thus the protocol can model various hardware implementations of networks.

In the process of developing the hierarchical protocol, we discovered that it was not simply a matter of adapting FLASH and DASH to support MESI and silent drop, as such combinations can easily lead to livelock. One such scenario is shown in Figure 3.2.

In Figure 3.2, L1-1 of cluster-1 (or agent-1) initially has an exclusive copy. In Step 1, agent-1 silently drops the cache line. In Step 2, L1-1 of cluster-2 (or agent-2) requests a shared copy of the same line, and this request is finally forwarded to agent-1 in Step 5. In Step 6, agent-1 NACKs the forwarded request, because it no longer has the cache line. At this time, the local directory of cluster-1 has no information about the silent drop happening in agent-1. Also, it is not safe to for the local directory to use the copy in the

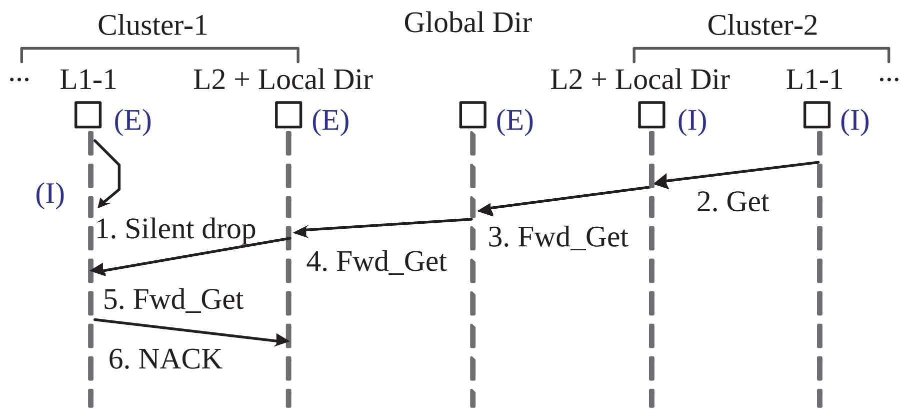


Figure 3.2. A scenario that can lead to the livelock problem.

L2 cache to reply the forwarded request, because there may exist an incoming writeback request from agent-1. So the local directory of cluster-1 can only forward the NACK reply to agent-2. This process can iterate such that the global directory keeps forwarding requests to cluster-1, and agent-1 keeps NACKing. This results in a livelock problem.

One possible solution to the livelock problem could be that whenever agent-1 replies a forwarded request, it also sends the reply to the local directory so that the latter can have more information on the cache line status. However, this approach still cannot solve the problem because when the local directory receives such a NACK reply, it is not clear whether agent-1 has silently dropped the cache line, or a writeback request from agent-1 is on the way, or a reply message to a former request of agent-1 has not been received yet.

There can be many solutions to the above livelock problem. Here we just describe one which may not be very efficient in performance. In more detail, we prevent the livelock problem by making writeback a blocking operation in the protocol, and also adding another type of NACK message indicating possible silent drops. By a blocking writeback, we mean that after an agent or cluster sends a writeback request, it cannot issue new requests or process forwarded requests, until an acknowledgment from the directory is received. At the same time, we propose a new type of NACK reply "NACK_SD", to notify the directory that silent drop may have happened. NACK_SD can only be used by an agent or cluster, when its cache line is invalid and it is not waiting for any reply of a former request. Upon receiving a NACK_SD, the directory will reset the exclusive ownership to itself if it is the current owner of the cache line. Figure 3.3 shows a simple example of how our approach works.

In this scenario, the L1 cache of L1-1 (agent-1) of cluster-1 initially has a modified copy. In Step 1, it writes the dirty copy back to the local directory. In Step 5, agent-1 receives the forwarded request from cluster-2; because it is blocked on the writeback, it has to wait for the acknowledgment. Only after the acknowledgment is received in Step 6, can agent-1 reply the forwarded request. At this time, agent-1 is invalid and it is not waiting for any reply, it will respond with NACK_SD. On receiving the reply, the local directory of cluster-1 simply sets the current owner to itself, which is already

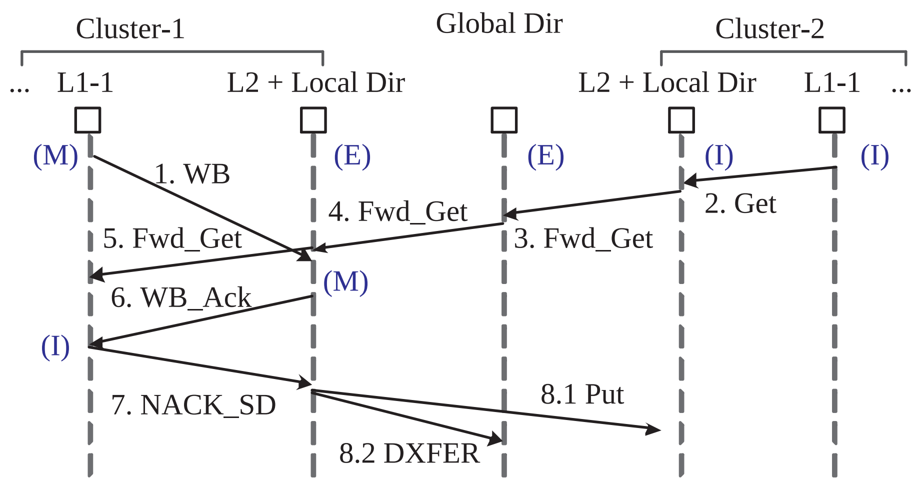


Figure 3.3. A sample scenario of the hierarchical protocol.

the case as the result of writeback. Finally, the local directory will send a grant reply to cluster-2 with the dirty data, and notify the global directory that exclusive ownership transfer (“DXFER”) happens.

This hierarchical protocol [2] coded in Murphi has about 2500 lines of code (LOC). It contains all the control logic and data coherence safety properties that the FLASH and DASH models originally have. To be more precise, the DASH model was from the Murphi distribution [3]. The FLASH model was from the work of Chou et al [42], which was translated from McMillan’s SMV code [100], which in turn was translated from Park’s PVS code [116]. Basically, these properties assert that (i) no two L1 caches in the same cluster can be both exclusive or modified, (ii) no two L2 caches can be both exclusive or modified, and (iii) every coherence read will get the latest copy in the system.

3.1.2 The Second Benchmark Protocol

Our noninclusive protocol [2] has the same configuration as the inclusive protocol presented in Section 3.1.1. The intracluster and intercluster protocols all use an invalidation-based directory MSI protocol. We choose MSI instead of MESI, because the noninclusive protocol is much more complex than the inclusive one, to be shown in the experimental results in Section 3.3.4.

We still use Figure 3.1 to represent the noninclusive coherence protocol. As defined by *noninclusive*, for any cache line in an L1 cache, it is not necessarily in the L2 cache of the same cluster. This characteristic is modeled in Murphi by allowing a cache line in the L2 cache to be silently dropped nondeterministically, if it is not the current owner of the line. Also, any shared cache line is allowed to be silently dropped. Moreover, to make the verification problem more interesting, we assume that when an L2 cache line is evicted, the local directory entry of the cluster is also evicted. Although this assumption may not be very practical for real coherence protocols due to performance issues, it forces us to come up with new verification techniques, other than those for the inclusive hierarchical protocol.

This noninclusive coherence protocol is different from the inclusive one in several aspects. First of all, for the network channels used inside a cluster, other than those used by the inclusive protocol, there also exists a set of broadcast channels. These channels are used when there is a cache miss in the L2 cache, and the local directory entry has been evicted out, i.e., it has no information on whether some L1 has a valid copy or not. At this time, the request for the cache miss will be broadcasted to all the L1 caches in the cluster. After the broadcast, if a reply containing a valid copy is received then we are done; otherwise the request will be forwarded to the global directory. With these broadcast channels, the L1 caches can send coherence requests to the local directory at any time, without contending with the broadcast requests.

Second, the characteristics of the noninclusive protocol can make the local directory have an imprecise record of a cache line. Figure 3.4 shows a simple scenario of how the imprecision can happen. In this example, the cache of L1-2 and the L2 in the cluster initially have a shared copy, and this information is stored in the local directory. In Step 1, the L2 cache line is evicted, together with the local directory information. In Step 2, the cache of L1-1 in the same cluster requests a shared copy. Because the local directory has no information on this line, the request is broadcasted in the cluster in Step 3; Figure 3.4 does not show the broadcast to L1-1, as it is the requesting agent. The request is NACKed

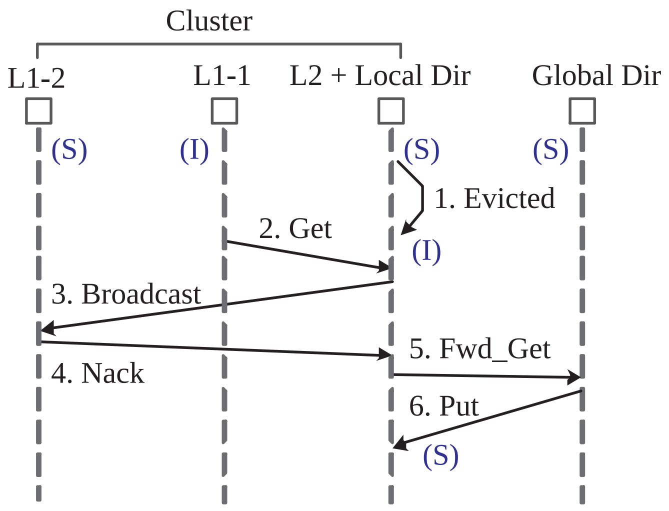


Figure 3.4. Imprecise information of the local directory.

in Step 4 because it is not safe for a shared copy to supply its data: the broadcast request could be interleaved with an invalidate request coming from outside the cluster. In Step 5, the request is forwarded to the global directory, and it is granted in Step 6. At this time, the local directory has lost the information that the L1-2 cache also has a shared copy. The problem with this kind of imprecision is that subsequent invalidations could miss the shared copy in the L1-2 cache, thus leading to coherence violations in certain systems.

We avoid this problem in the noninclusive protocol with a conservative solution. Back to Figure 3.4, when the global directory receives the forwarded request in Step 5, it can realize that the cluster already has a shared copy. Our solution is that in addition to the reply in Step 6, a tag is also attached indicating possible imprecision. When the reply and the tag are received, the local directory will record that all the L1 caches in the cluster have a valid shared copy, thus avoiding the imprecision. Here, recording all the L1 caches as shared is a conservative solution, as (i) the local directory does not know which specific L1 cache has a valid copy, and (ii) recording L1 caches as shared will only make the following invalidation requests invalidate some L1 caches that are invalid, which is unnecessary but does not affect coherence.

Our noninclusive coherence protocol [2] coded in Murphi has about 2500 LOC. Overall, it is similar to that in Piranha [19]. The differences are that Piranha makes many optimizations to improve performance. For example, one instance is that a duplicate copy of the L1 tags and state is kept at the L2 controllers, to avoid the use of snooping at L1 caches. Another instance is that, for L1 misses that also miss in the L2, they are filled directly from memory without allocating a line in the L2, to lower miss latency and better utilize the L2 capacity.

3.1.3 The Third Benchmark Snooping

Our snoop protocol [2] is much simpler than the other two benchmark protocols. It models two symmetric clusters where a snoop protocol is used inside a cluster, and a directory based MSI protocol is used among the clusters. Figure 3.5 intuitively shows the protocol.

In more detail, the snoop 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 usually can increase the bus efficiency by pipelining multiple requests. Thus they are more complex, and have more corner cases. Here because our main purpose is to check whether the approaches developed for the two other benchmarks can be applied for snooping protocols, a simple multicore protocol

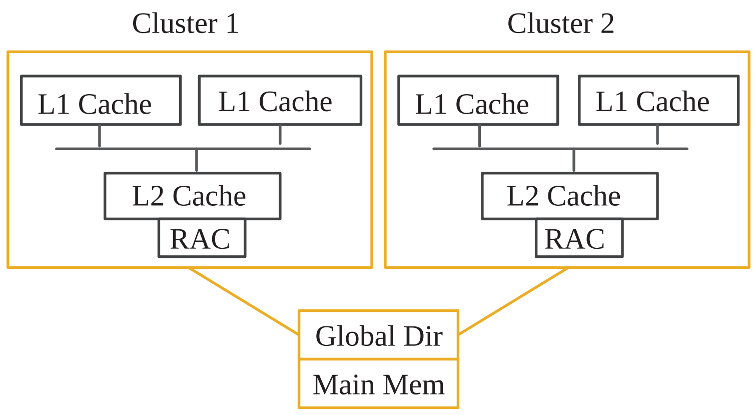


Figure 3.5. A multicore cache coherence protocol with snooping.

will suffice. Moreover, the high on-chip bandwidth and low on-chip latency can also justify our decision.

The L1 and L2 caches on a chip are modeled as noninclusive. More specifically, for a shared cache line in an L1 cache, the line can also be in the L2 cache. However, for an exclusive cache line in an L1 cache, the line cannot be in the L2 cache and vice versa. Explicit writebacks on exclusive cache lines are modeled in both the intracluster and the intercluster protocols. Different from the first benchmark protocol in Section 3.1.1, a cache agent will not be blocked after a writeback request, and no writeback acknowledgment will be issued. Also, silent drop on shared cache lines is not modeled. This protocol coded in Murphi altogether has about 1000 LOC.

3.2 A Compositional Framework

Our first two hierarchical protocols proved to be very complex such that 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 an SMP (symmetric multiprocessing) server, using 18GB of memory, and with 40-bit of hash compaction of the Murphi model checker. This is not surprising, considering the multiplicative effect of having four instances of coherence protocols running concurrently, i.e., one intercluster and three intracluster protocols, We believe all hierarchical coherence protocols will state explode in this manner. This section describes our compositional framework and the soundness of the approach.

The compositional framework consists of two parts: abstraction and assume guarantee reasoning. Given a hierarchical protocol, we first use abstraction to decompose it into a set of abstract protocols with smaller verification complexity, such that each abstract protocol overapproximates the original protocol on a subset of components. We then apply counterexample guided refinement on the abstract protocols to constrain the overapproximation, 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.

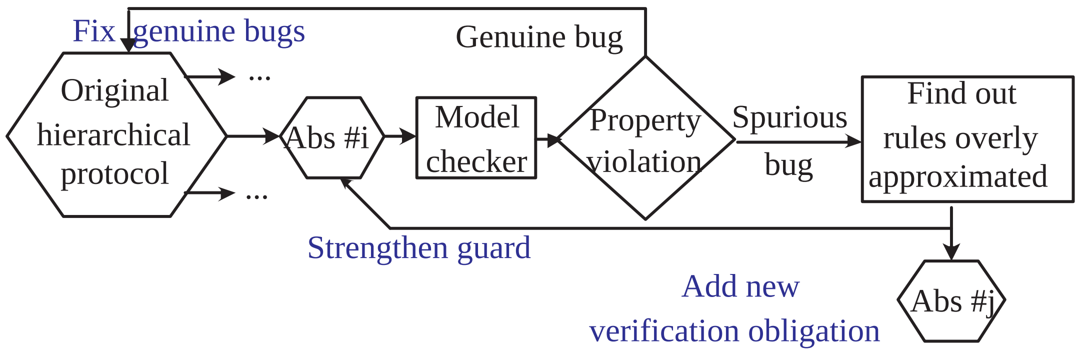


Figure 3.6. The workflow of our approach.

The workflow of our approach is shown in Figure 3.6. Given a hierarchical protocol, we first build a set of abstract protocols where each abstract protocol overapproximates, by construction, the original protocol. 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 is a superset of that in the original protocol. We will formally define overapproximation in Section 3.2.4. We then model check each of the abstract protocols individually. If a genuine bug is found in an abstract protocol `Abs #i`, we fix it in the original protocol and regenerate all the abstract protocols. If a spurious bug is reported in `Abs #i`, we constrain it and at the same time, add a new verification obligation to one of the abstract protocols `Abs #j`. In more detail, each abstract protocol employs assumptions that will be validated only by verifying some number of abstract protocols `Abs #j`'s that, then, justifies these assumptions. In turn, `Abs #j` will also employ assumptions that are justified by verifying some number of `Abs #i`'s.

We will take the inclusive multicore protocol as the driving example in this section. In Section 3.3, we will describe how to extend this framework to verify the hierarchical noninclusive protocol and the protocol with snooping.

3.2.1 Abstraction

For the hierarchical protocol shown in Figure 3.1, we decompose the protocol into three abstract protocols. Figure 3.7 intuitively shows one of the abstract protocols.

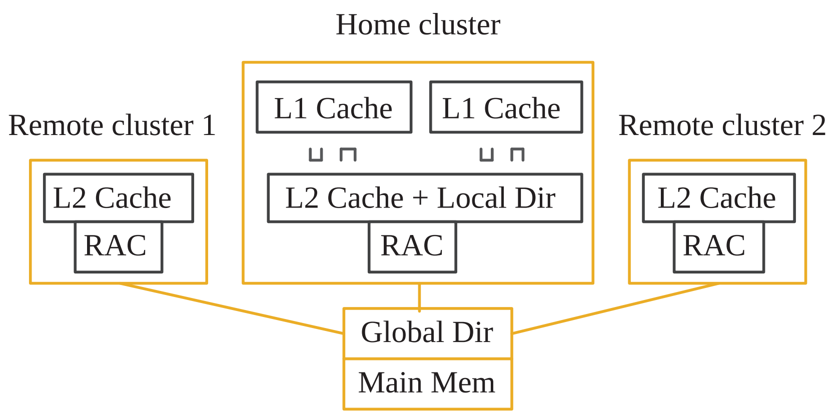


Figure 3.7. An abstract protocol.

Contrasting with Figure 3.1, we can see that it retains all the details of the home cluster while abstracts away some components of the two remote clusters. The other two abstract protocols are very similar, except that in each abstract protocol, one remote cluster is retained and the remaining clusters are abstracted. Due to the symmetry between the two remote clusters, the two abstract protocols obtained by retaining one of the remote clusters, respectively, are symmetric. So in the following we will only consider the two distinct (nonsymmetric) abstract protocols.

The derivation of the abstract protocols from the hierarchical protocol involves the abstraction of the global state variables, the transition relation, and the verification obligations. These steps will now be explained, and Section 3.2.4 will present a soundness proof for the compositional reasoning method.

For the abstraction of state variables, Figure 3.8 shows the data structure of a cluster in the original protocol (“ClusterState”), and that of an abstracted cluster in an abstract protocol (“AbsClusterState”), in the Murphi description language. *ClusterState* contains five blocks of information: (i) **B1**. all the L1 caches in a cluster, (ii) **B2**. the set of network channels used inside a cluster, (iii) **B3**. the local directory, (iv) **B4**. the L2 cache, and (v) **B5**. the controller of a cluster, which is used to communicate with other clusters and the global directory.

```
ClusterState: record
  -- 1. L1 caches
  L1s:  array [L1Cnt] of L1CacheLine;

  -- 2. local network channels
  UniMsg:    array [L1L2] of Uni_Msg;
  InvMsg:    array [L1L2] of Inv_Msg;
  WbMsg:     Wb_Msg;
  ShWbMsg: ShWb_Msg;
  NackMsg:   Nack_Msg;

  -- 3. local directory
  LocalDir: record
    pending:     boolean;
    ShrSet:      array [L1Cnt] of boolean;
    InvCnt:      CacheCnt;
    HeadPtr:     L1L2;
    ReqId:       L1Cnt;
    ReqCluster:  ClusterCnt;
  end;

  -- 4. L2 cache
  L2:            L2CacheLine;
  ClusterWb:  boolean;

  -- 5. remote access controller
  RAC: record
    State:    RACState;
    InvCnt: ClusterCnt;
  end;
end;
```

```
-- remove 1, 2, 3 from ClusterState
AbsClusterState: record
  -- 4. L2 cache
  L2:            L2CacheLine;
  ClusterWb:  boolean;

  -- 5. remote access controller
  RAC: record
    State:    RACState;
    InvCnt: ClusterCnt;
  end;
end;
```

Figure 3.8. The data structures of an original and an abstract cluster.

In these five blocks, the intracluster protocol only involves the first four blocks of information, and the intercluster protocol only involves the last two blocks. That is, the intracluster protocol does not involve the RAC details. On the other hand, the intercluster protocol only has the high level information of which cluster has a valid copy; it does not track the specific L1 caches' status. The *AbsClusterState* as shown in Figure 3.8 can be simply obtained by projecting the blocks B1, B2 and B3 away from *ClusterState*. In general, the set of state variables of each abstract protocol is a proper subset of that in the original hierarchical protocol. 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 3.2.4. Here, the abstraction of state variables for the inclusive multicore protocol is just a guideline for doing abstraction in practice, but in general any method that meets the minimal requirement as in Section 3.2.4 will also do.

Now let's look at how the global state variables in the abstract protocol shown in Figure 3.7 can be represented, compared with that in the original protocol. First of all, every cluster in the original protocol will be declared as *ClusterState*, while in the abstract protocol only the home cluster is declared as *ClusterState*, and the remaining two will be declared as *AbsClusterState*. Second, the global directory and all the network channels used among the clusters are retained in the abstract protocol. That is, they will be represented in exactly the same way as that in the original protocol. From the above discussion, it is clear that our abstraction of state variables is achieved by a projection of the state variables onto a subset of these variables, and then suitably modifying the transition relation as will be explained now.

For each transition (rule) in the original hierarchical protocol, a *set* of corresponding rules will be constructed for each abstract protocol. Given an original hierarchical protocol and the state variables to be projected away (eliminated), the basic idea of abstraction of transition relations is as follows. For every rule in the form of $guard \rightarrow action$, (1) for any assignment of the form `v := E` in $action$, if `v` is a variable that has been eliminated, then the whole assignment will be eliminated; otherwise if `E` contains even one variable that has been eliminated, we replace `E` with a nondeterministic selection over the type of E; (2) if a subexpression in $guard$ contains any variable that has been eliminated, we replace the subexpression with $true$.

In the following, we will only present the procedure to construct the transition relations from the original protocol in Figure 3.1, to the abstract protocol in Figure 3.7. The procedure for other abstract protocols is very similar so we skip it here. Our procedure is described in the context of Murphi, but the ideas are broadly applicable. We will use D to denote the set of state variables which are eliminated in the abstract protocol. We consider the rules one at a time in the original hierarchical protocol. Figure 3.9 shows

---

Preprocessing:

1. For every ruleset, convert it to a set of simple rules by enumerating all possible constant values for every parameter of the ruleset.

2. For every simple rule, if there exists a conditional statement in its action, i.e., an ’if’ or a ’switch’ statement, recursively convert the rule to a set of simple rules, such that the resulting rules no longer contain conditional statements. In more detail, the guard of each rule is the conjunction of the original rule guard and the condition to switch to that branch; the action is obtained by replacing the conditional statement in the original rule action with the body of the branch.

3. For every expression with the logical negation operator ($\neg$) in every simple rule, convert the expression into negation normal form, such that negation is only applied to propositional variables.

4. For every expression, if there exists a boolean subexpression involving $forall$ or $exists$ with parameters over multiple clusters including the home cluster, replace the subexpression with two conjunctive expressions: one with the parameter over the home cluster, and the other over the rest of the clusters.

5. For every remote cluster in the protocol, replace its data structure from `ClusterState` to `AbsClusterState`, i.e., replacing the concrete cluster with an abstract cluster.

Figure 3.9. The abstraction procedure (I) for the transition relation.

the steps to preprocess the rule, and Figure 3.10 shows the main abstraction procedure for the transition relation.

Now let's look at one example to see how a rule is abstracted using our procedure. Consider the scenario when a writeback request from an L1 cache is received by the local directory of the remote cluster 1 ($r1$). As shown on the left-hand side of Figure 3.11, when this request is received, the following statements will be executed: (i) the L2 cache copy is updated with the latest copy, (ii) the local directory sets the L2 cache be the current owner (*HeadPtr*) of the cache line, (iii) the L2 cache line is set to modified, (iv) the writeback acknowledgment is sent back to the L1 cache, and (i) the writeback request is reset.

For the abstract protocol shown in Figure 3.7 where the home cluster is retained, the local directory and all the network channels used inside the remote cluster will be eliminated. That is, the state variables *WbMsg, LocalDir, UniMsg* of *Clusters[r1]* are all in the set of variables to be eliminated. Applying our procedure, the guard of the rule will be converted to $true$, and the statements (ii), (iv) and (v) will simply be eliminated. The resulting rule is shown on the right-hand side of Figure 3.11.

For every verification obligation in the original hierarchical protocol, we abstract it aggressively in abstract protocols as follows:

1. Preprocess: If the verification obligation uses $forall$ as the top-level logic operator with parameters over multiple clusters including the concrete cluster, substitute the verification obligation with two verification obligations: one with the parameter of $forall$ over the retained concreted cluster, and the other over the rest of the abstracted clusters.

2. Procedure: If the verification obligation contains any state variable that is eliminated in the abstract protocol, replace the verification obligation with $true$.

The reason we abstract verification obligations in the above manner is as follows. For every verification obligation p in the original hierarchical protocol, the abstraction will generate a (or a set of) verification obligation p_i in each abstract protocol. We want the property that $\wedge p_i \Rightarrow p$ holds. This property, together with the fact that each abstract

Procedure: For the guard of every simple rule, if a literal (a propositional variable or its negation) in the guard contains any variable that has been eliminated, the literal turns into true. For every statement in the action of each simple rule:

- **Assignment**: For an assignment statement, if the left hand side involves any variable in D, remove the assignment; otherwise, if there exists any subexpression on the right hand side involving variables in D, replace the subexpression with a nondeterministic value in its type.

- **Forstmt**: If the quantifier in the forstmt ranges over multiple clusters including the home cluster, divide the statement into two forstmts: one with the quantifier over the home cluster, and the other over the remaining clusters. If the quantifier involves any variable in D, remove the forstmt. For every statement inside the forstmt, recursively apply the abstraction procedure.

- **Whilestmt**: For the expression that controls the whilestmt, if there exists a subexpression involving variables in D, replace the subexpression with a nondeterministic value. For every statement inside the whilestmt, recursively apply the abstraction procedure.

- **Aliasstmt**: If there exists any variable declaration involving variables in D, add that variable to the set D and recursively apply the procedure to every statement in the aliasstmt.

- **Proccall**: If there exists any parameter in the proccall involving variables in D, replace the parameter with a nondeterministic value. For each statement in the procedure, recursively apply the procedure.

- **Clearstmt, Putstmt, Errorstm**: If the target expression is in D, remove the statement. Otherwise if there exists an expression in the statement involving variables in D, replace that expression with a nondeterministic value.

- **Assertstmt**: For the expression inside the assertstmt, if there exists any subexpression involving variables in D, replace the subexpression with $true$.

- **Returnstmt**: For the expression in the returnstmt, if there exists any subexpression involving variables in D, replace the subexpression with a nondeterministic value.

Figure 3.10. The abstraction procedure (II) for the transition relation.

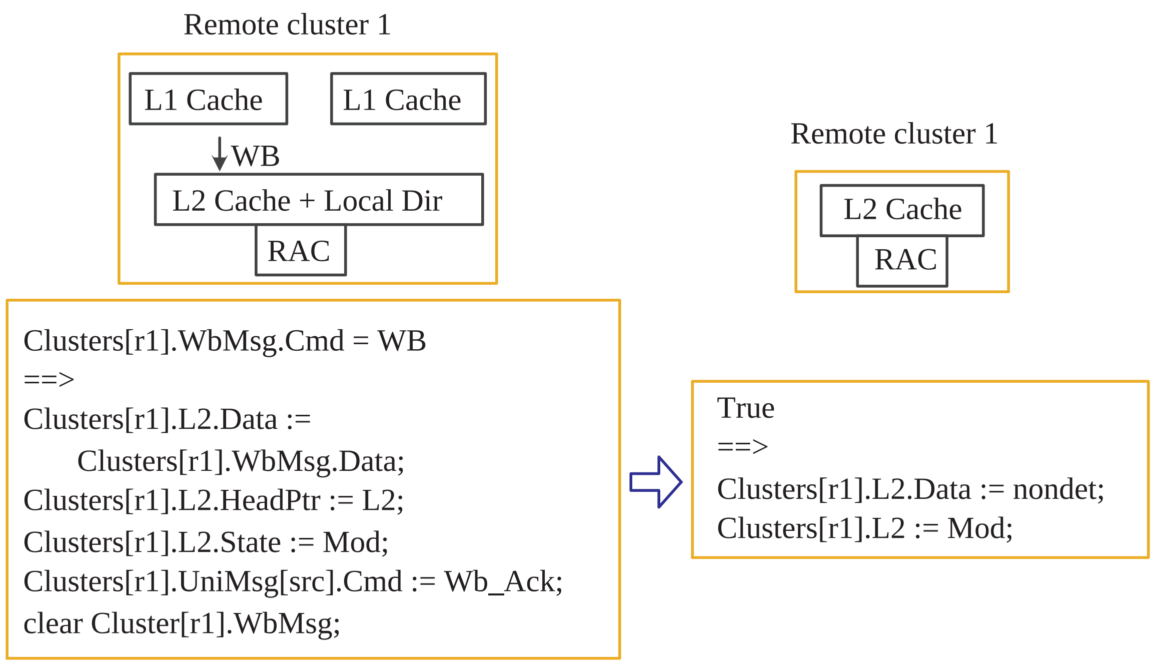


Figure 3.11. The Murphi rule for the writeback example.

protocol overapproximates the original protocol on a subset of components, can ensure that our compositional approach is conservative. The manner in which our abstraction is performed on verification obligations can help simplify the soundness argument of our approach, as to be described in Section 3.2.4.

Now consider every coherence property p in our inclusive multicore protocol [2] described in Section 3.1.1. After preprocessing, p is either abstracted to itself or $true$, and the property $\wedge p_i \Rightarrow p$ is valid. Thus, if the abstract protocols can be verified correct, the original protocol must be correct. In Section 3.3, we will describe the possible problems by directly applying the abstraction to the noninclusive hierarchical protocol and the protocol with snooping, and also present our solutions to them.

3.2.2 Assume Guarantee Reasoning

From Section 3.2.1, it is clear that each abstract protocol, by construction, overapproximates the original hierarchical protocol on those state variables that are retained in the abstract protocol. The overapproximation is achieved by the abstraction procedure on transition relations, such that the guard of each rule in the original protocol becomes more permissive, while the action allows more updates to happen.

As described in the workflow of Figure 3.6, 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 overapproximation in the abstraction process. The question of how to identify whether it is a genuine bug will be discussed in Section 3.4.

We fix genuine bugs in the original protocol, regenerate all the abstract protocols, and redo the model checking. If a spurious bug is reported in an abstract protocol, we constrain the abstract protocol and at the same time, add a new verification obligation to one of the abstract protocols. The question of adding the verification obligation to which abstract protocol depends on which one contains the details for verifying the verification obligation. Essentially, for every such new verification obligation, it suffices to prove the verification obligation in any of the abstract protocols (as each abstract protocol overapproximates the original protocol). Since each abstract protocol retains the details on a different subset of the original protocol, the abstract protocols in fact depend on each other to justify the soundness of our approach. More specifically, assume guarantee reasoning is used in the soundness proof, with formal details to be presented in Section 3.2.4. 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 $g \rightarrow a$. We then find out the corresponding rule in the original hierarchical protocol from which the abstracted rule is obtained. Let the original rule be $G \rightarrow A$. Now we strengthen the abstracted rule into $g \wedge p \rightarrow a$, where p is a formula which only involves the variables in both the intracluster and the intercluster protocols. 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 more permissive than g', and also G. Thus the strengthened guard $g \wedge p$ is still more permissive than G, i.e.,

$G \Rightarrow g \wedge p$. As for the question of how to choose such a g', it is very simple when G is in the form of a conjunctive formula, i.e., $G = g_1 \wedge \ldots \wedge g_n$, $n \geq 1$. In this case, any g_i, $i \in [1..n]$ can serve as the g'. Otherwise, we can simply choose $g' = G$.

Now let's look at one example to see how the refinement is performed. For this example, let M denote the original inclusive hierarchical protocol shown in Figure 3.1. 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 example in Figure 3.11 again. In this example, a writeback request from an L1 cache is received by the local directory in the remote cluster-1. In M_h, this rule will be abstracted to the one shown on the right-hand side of Figure 3.11.

It is clear that the above abstracted rule can easily introduce violations to the verification obligations. This is because the abstracted rule simply means that, at any time, the L2 cache copy of the remote cluster-1 can be updated. For property violations introduced by this abstracted rule, they correspond to a spurious bug because the abstracted guard $true$ is overly approximated compared with the guard of the rule in M.

To refine M_h, we do two things. On one hand, we constrain the guard of the abstracted rule such that only when the L2 cache copy is exclusive or modified, can its copy be updated. On the other hand, we add a new verification obligation (hopefully an invariant) to M_{r1}, which asserts that whenever a writeback request is received from an L1 cache in the remote cluster-1, the L2 cache copy must be exclusive or modified which is the characteristics of the inclusive cache. Figure 3.12 shows how M_h is refined.

In the refinement process for the abstract protocols of the first benchmark protocol, a genuine bug was found. The scenario of the genuine bug is shown in Figure 3.13. In this scenario, the remote cluster-1 (r1) initially holds a modified copy *d1* in the L1 cache-1 (agent1). In Step 1, agent1 writes back the dirty line. In Step 4, r1 receives a forwarded request from the remote cluster-2 (r2) for an exclusive copy. This request is forwarded to agent1 in Step 5. In Step 7, agent1 NACKs the forwarded request as it no longer has the cache line. Finally, in Step 8.1 r1 uses the cache copy in the L2 cache to reply to r2, and at the same time, send a message to the global directory notifying that r2 is now the owner cluster.

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;
```

Figure 3.12. Refining the writeback example.

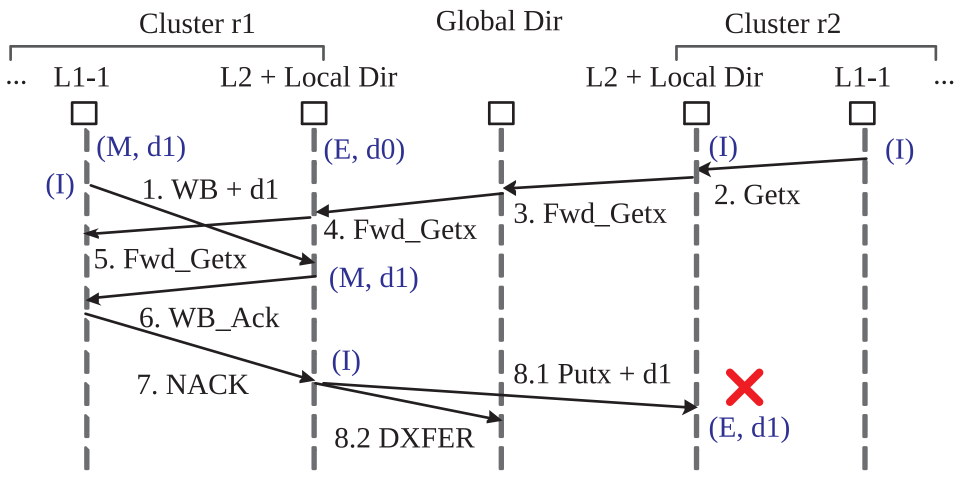


Figure 3.13. A sample scenario of the genuine bug in the original hierarchical protocol.

In Step 8.1 of the above scenario, r1 should indicate whether the data supplied to r2 are dirty or not, because these data are not sent to the global directory in Step 8.2. If there is no such indication, r2 could silently drop the cache copy later. This in effect throws away the only dirty copy in the system, violating the coherence protocol. 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.

3.2.3 Experimental Results

For the inclusive hierarchical protocol, Table 3.1 shows the experimental results, using the 64-bit version of the Murphi model checker [4]. All the experiments were performed on an Intel IA-64 machine, and 40-bit hash compaction of Murphi was used.

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. Among them, 1.5 days were spent on manually abstracting the model which was very tedious. We will present how this process can be automated in Section 3.3.1.

Table 3.1. Verification complexity of the first benchmark protocol (I).

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

3.2.4 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 it properties as well. We first introduce several notations for the proof.

3.2.4.1 Some Preliminaries

Let V be a set of variables and D be a set of values which are both nonempty. 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 $g \longrightarrow a$ 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.

For this proof, we only consider one notion of execution for models. An *interleaving* execution of a model is based on steps in which a single enabled transition fires. For a transition $t = (g \longrightarrow a)$, we write $s \xrightarrow{t} s'$ to denote that $g(s)$ holds and $s' = a(s)$. An interleaving 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 $\models M$.

3.2.4.2 A Theory Justifying Circular Reasoning

In the following, we will present two theorems to justify our compositional approach. Theorem 1 will provide 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 is also conservative.

Thus, the compositional framework as a whole 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 (i) $V_i \subseteq V$, (ii) $I_i \supseteq I \mid V_i$, (iii) $\forall t = g \rightarrow a \in T$, $\exists t_i = g_i \rightarrow a_i \in T_i$ such that $g \Rightarrow g_i$, and for every state s, s' s.t. $s \xrightarrow{t} s'$, $s \mid V_i \xrightarrow{t_i} s' \mid V_i$ holds, and (iv) $\forall p \in A$, $\exists p_i \in A_i$ 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 $p \in A$ 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 $p(s_m)$ does not hold. For every $i \in [1..n]$, there exists a valid execution of M_i as follows:

$$s_0 \mid V_i,\ s_1 \mid V_i,\ \ldots,\ s_m \mid V_i$$

We will use induction to prove that the above execution is valid for M_1, and for other M_i's they can be proved similarly.

- Base Case: $j = 0$. From (ii) 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 $s_k \xrightarrow{t_k} s_{k+1}$, from (iii) 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 valid in M_1. That is, the induction holds for $j = k + 1$.

Finally, we consider p and its corresponding verification obligation in each M_i. From (iv) in the theorem, it follows that for every $i \in [1..n]$, $\exists p_i \in A_i$, p_i is either p or $true$, and $\wedge_{i \in [1..n]} p_i \Rightarrow p$. Thus, there must exist an $r \in [1..n]$ such that $p_r = p$. Let $s_{m,r} = s_m \mid V_r$. Since $\models M_r$ holds, we have $p_r(s_{m,r})$ holds, i.e., $p(s_{m,r})$ holds. So p must hold at s_m. Contradiction. $\square$

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 (i) $V_i \subseteq V$, (ii) $I_i \supseteq I \mid V_i$, (iii) $\forall t = g \longrightarrow a \in T$, $\exists t_i = g_i \wedge e_i \longrightarrow a_i \in T_i$ such that $g \Rightarrow g_i$, and if $e_i \neq true$ 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 (iv) $\forall p \in A$, $\exists p_i \in A_i$ 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 exists a counterexample to a verification obligation $p \in A$ of M: $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{m-1}} s_m$ where $m \geq 0$, and $t_j = g_j \longrightarrow a_j$ for all $j \in [0..m-1]$, such that $p(s_m)$ does not hold. For every $i \in [1..n]$, there exists a valid execution of M_i as follows:

$$s_0 \mid V_i,\ s_1 \mid V_i,\ \ldots,\ s_m \mid V_i$$

We will use induction to prove the above execution for every M_i, $i \in [1..n]$.

- Base Case: $j = 0$. From (ii) 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. Let $s_{j,i} = s_j \mid V_i$ for $j \in [0..k]$.

- Induction Step: We consider $j = k+1$. From (iii) in the theorem, for $s_k \xrightarrow{t_k} s_{k+1}$ in M, for every $i \in [1..n]$, there exists $t_{k,i} = g_{k,i} \wedge e_{k,i} \longrightarrow a_{k,i} \in T_i$, such that $g_k \Rightarrow g_{k,i}$, if $e_{k,i} \neq true$ 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. The induction holds for $j = k+1$.

Finally, we consider p and its corresponding verification obligation in each M_i. From (iv) in the theorem, it follows that for every $i \in [1..n]$, $\exists p_i \in A_i$, p_i is either p or $true$, and $\wedge_{i \in [1..n]} p_i \Rightarrow p$. Thus, there must exist an $r \in [1..n]$ such that $p_r = p$. Let $s_{m,r} = s_m \mid V_r$. Since $\models M_r$ holds, we have $p_r(s_{m,r})$ holds, i.e., $p(s_{m,r})$ holds. So p must hold at s_m. Contradiction. $\square$

The difference of Theorem 2 with Theorem 1 lies 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 $g_i \wedge e_i \rightarrow a_i$, rather than $g_i \rightarrow a_i$ as 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.

3.3 Verification One Level at a Time

In the previous section, we present a compositional framework to verify hierarchical cache coherence protocols. Given a hierarchical protocol, we decompose it into a set of abstract protocols with smaller verification complexity. By only verifying the abstract protocols, we can conclude the correctness of the original hierarchical protocol. Although this approach has been successfully applied to the inclusive hierarchical protocol which cannot be verified through traditional monolithic model checking, several problems exist in the approach. The drawbacks of the approach include:

- Even a single abstract protocol models more than one level of the coherence protocols, thus still creating very large product space;
- Details such as noninclusive caching hierarchies, and hierarchical protocols which use snooping, may not be handled;
- All the abstract protocols are created manually, which is tedious and error prone.

We will present these limitations in more detail in the following.

First of all, for the abstract protocol as shown in Figure 3.7, we can see that it contains two protocols: one intracluster and one intercluster protocols. The product state space of two protocols running concurrently can still make the state space of an abstract protocol very large. This is in fact confirmed by the experimental results in Table 3.1. There, one abstract protocol still has more than 0.6 billion of states. This is the typical number of states currently a monolithic model checker can explore concretely. It is clear that for a hierarchical protocol which is more complex than the inclusive multicore benchmark protocol, even our compositional approach cannot handle the complexity.

Second, we have only applied the compositional approach to the inclusive multicore protocol. There could be other complications arise in the noninclusive multicore protocol, and the one with snooping. This is because for the inclusive protocol, the L2 cache block can be directly used as a “summary” of the intracluster protocol. However, for noninclusive and snooping protocols, there is no component that can be used as a summary of the cluster, in a straightforward way.

Finally, all the abstract protocols that we have presented are constructed manually. Though the process is quite simple, it is time consuming and error prone. Can we mechanize the process of creating the set of initial abstract protocols? That is, given an original hierarchical protocol, and the set of state variables to be projected way for each abstract protocol, we want to obtain the abstract protocols automatically.

In this section, we will present extensions to the compositional framework which can overcome all the limitations. We develop a new abstraction approach such that by using the same compositional framework, we can achieve the verification of hierarchical protocols one level at a time. For noninclusive and snooping protocols, we use history variables in an assume guarantee manner, so that these protocols can be verified in the same way as for the inclusive protocol. Finally, we describe the procedure for the new abstraction process, and present an implementation which has mechanized the process.

3.3.1 Abstraction One Level at a Time

We present a new decomposition approach, such that every resulting abstract protocol involves either an intracluster or the intercluster protocol, but never both. This approach is very similar to that in Section 3.2.1, in that each abstract protocol is obtained by overapproximating the original protocol, by projecting away different components.

For the inclusive multicore protocol in Figure 3.1, our new approach decomposes it into four abstract protocols: one intracluster protocol for every cluster, and the intercluster protocol. Because the two remote clusters are of identical design, there are altogether three distinct abstract protocols, as shown in Figure 3.14.

Again, the derivation of the abstract protocols from the hierarchical protocol involves the abstraction on the global state variables, the transition relation, and the verification

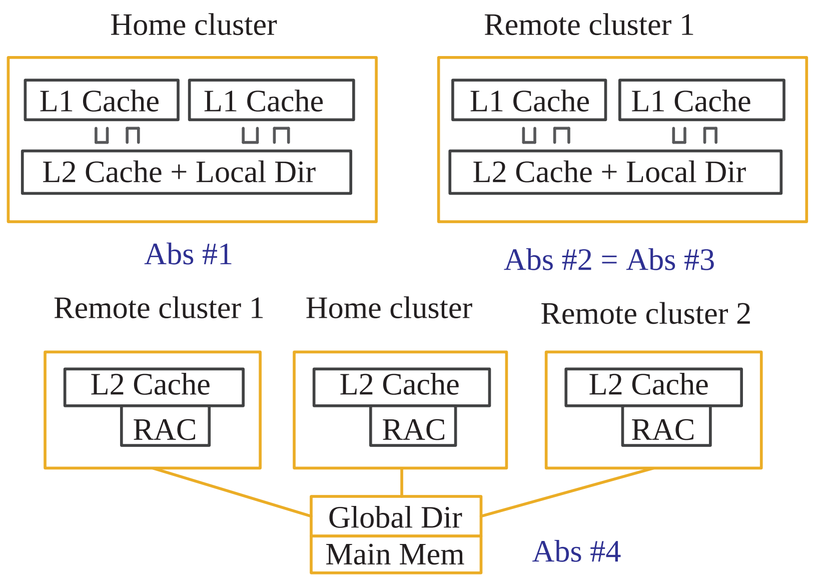


Figure 3.14. The abstract protocols obtained via the new decomposition approach.

obligations. We will describe the abstraction process for the state variables and the transition relation. The process for the verification obligation is the same as that in Section 3.2.1 so we skip it here.

For a cluster in the original hierarchical protocol, it will be abstracted differently in the abstract intracluster and the intercluster protocols. Figure 3.15 shows the abstracted data structures of a cluster, in contrast to the concrete one in Figure 3.8. For every abstract intracluster protocol, its data structure can be represented as an *AbsIntraClusterState*. That is, the RAC inside the cluster is dropped from the original protocol; all the other clusters, the global directory, and the network channels used among the clusters, are also dropped. For the abstract intercluster protocol, every cluster can be represented as an *AbsInterClusterState*, and the networks among the clusters and the global directory are retained. In summary, our abstraction on state variables is still a projection function.

The process of abstraction on transition relations is very similar to that in Section 3.2.1. The difference is that an abstract intracluster protocol can now contain just one cluster. So the *forall* and *exists* expressions, and the *forstmt* statements need to be processed differently. In the following, we present the procedure to construct the transition

```
AbsIntraClusterState: record
  -- 1. L1 caches
  L1s:  array [L1Cnt] of L1CacheLine;

  -- 2. local network channels
  UniMsg:     array [L1L2] of Uni_Msg;
  InvMsg:     array [L1L2] of Inv_Msg;
  WbMsg:      Wb_Msg;
  ShWbMsg: ShWb_Msg;
  NackMsg:    Nack_Msg;

  -- 3. local directory
  LocalDir: record
    pending:     boolean;
    ShrSet:       array [L1Cnt] of boolean;
    InvCnt:       CacheCnt;
    HeadPtr:      L1L2;
    ReqId:         L1Cnt;
    ReqCluster:  ClusterCnt;
  end;

  -- 4. L2 cache
  L2:              L2CacheLine;
  ClusterWb:  boolean;
end;
```

```
AbsInterClusterState: record
  -- 4. L2 cache
  L2:              L2CacheLine;
  ClusterWb:  boolean;

  -- 5. remote access controller
  RAC: record
    State:    RACState;
    InvCnt: ClusterCnt;
  end;
end;
```

Figure 3.15. An abstracted cluster in the intracluster and intercluster protocols.

relations for abstract protocols. We still use D to denote the set of state variables that are eliminated in the abstract protocol. Figure 3.16 shows the procedure for abstracting the transition relation.

Given a hierarchical protocol and the state variables to be projected away, we have implemented a tool that can automatically generate the abstract protocol. The tool was implemented by Michael DeLisi, based on the distribution of Murphi. The source code and the description of the tool can be found at [2].

Preprocessing:

- 1. 2. 3. The same as in Section 3.2.1.
- 4. For *forall* and *exists* expressions, do nothing.
- 5. In the abstract intercluster protocol, replace the data structure of each cluster from *ClusterState* to *AbsInterClusterState*. In every abstract intracluster protocol, replace the structure of the cluster that the protocol models to *AbsIntraClusterState*.

Procedure:

- Forstmt: If the quantifier in the forstmt involves any variable in D, remove the forstmt. Otherwise, recursively apply the procedure to every statement inside the forstmt.
- For the guards and the rest of statements, they are processed in the same way as in Section 3.2.1.

Figure 3.16. The procedure for abstracting the transition relation.

As a general overview, the tool takes two input files: the first is the original protocol; the second contains the type and variable declarations that are retained in the abstract protocol, also the abstracted startstate which is optional. The current implementation assumes that all the expressions in the original protocol model is already in negation norm form, i.e., the negation operator can only be applied to propositional variables. Internally, for every type or variable declaration in the original protocol, the declaration is looked up in the second file to see if the declaration is eliminated. The implementation is more complex than this, because the internal of Murphi creates temporary type declarations and compare two types based on their names. Once all the declarations that are eliminated in the abstract protocol are obtained, the abstraction procedure is recursively applied on every statement in the original protocol.

After the abstract protocols are obtained using the "per-level" abstraction, they can be refined in the same way as in Section 3.2.2. That is, for genuine bugs, they will be fixed in the original hierarchical protocol; for spurious bugs, they can be refined using assume guarantee reasoning. Finally, if all the abstract protocols can be verified correct, the original protocol must be correct as per Theorem 1 and 2.

3.3.2 Using "Per-Level" Abstraction

From the previous section, it is clear that the new abstraction approach is very similar to the previous one. Now comes the question when can we use the new abstraction approach, and what are the restrictions on the new approach? The answer is that if different levels of a hierarchical protocol are modeled as *loosely coupled*, then the per-level abstraction can be applied.

By loosely coupled, we mean that every transition of the hierarchical protocol can only involve the state variables of one level, including the *interfaces* to interact with other levels, but never more. Here, we use interfaces to denote the common state variables used by two neighboring levels. For example, in the multicore protocol shown in Figure 3.1, the L2 cache block is the interface between the intracluster and the intercluster protocols.

We now show one example of a rule which violates the loosely coupled modeling, and see how it prevents the per-level abstraction from being applied. The conditions under which this rule can be executed are: (i) the local directory of a cluster receives a coherence request from an L1 cache, (ii) the L2 cache does not have the cache line, and (iii) the local directory is not busy, i.e., not pending on other requests for the same address. Once this rule is enabled, it will make the following updates: the local directory is set to busy, the RAC of the cluster is also set to busy, the request is forwarded to the global directory, and the original request is reset. The left-hand side of Figure 3.17 shows the rule, where *GUniMsg* are the networks among the clusters and *GDir* is the global directory.

It is clear that the above rule not only involves the intracluster details, but also the RAC and the networks to the global directory, i.e., the intercluster details. Applying the per-level abstraction, in the abstract intercluster protocol, this rule will be abstracted so that the resulting guard simply becomes (ii), i.e., the L2 cache does not have the line; the action becomes that the RAC is set to busy, and the request is forwarded to the

```
Clusters[c].UniMsg[src].Cmd = Get &
Clusters[c].L2.State = Invld &
Clusters[c].LocalDir.pending = false
==>
Clusters[c].LocalDir.pending := true;
Clusters[c].RAC.State := Wait;

GUniMsg[c].Src := c;
GUniMsg[c].Dest := GDir;
GUniMsg[c].Cmd := Get;

clear Cluster[c].UniMsg[src];
```

```
Clusters[c].UniMsg[src].Cmd = Get &
Clusters[c].L2.State = Invld &
Clusters[c].LocalDir.pending = false
==>
Clusters[c].LocalDir.pending := true;
Clusters[c].Req.Cmd := Get;

clear Cluster[c].UniMsg[src];
```

```
Clusters[c].Req.Cmd = Get     &
Clusters[c].RAC.State = None
==>
Clusters[c].RAC.State := Wait;

GUniMsg[c].Src := c;
GUniMsg[c].Dest := GDir;
GUniMsg[c].Cmd := Get;
```

Figure 3.17. Example rules of tightly and loosely coupled modeling.

global directory. The abstracted rule is in fact overly approximated. This is because it allows multiple requests for the same address to be issued continuously, before a reply is received.

To fix the spurious bug, we need to strengthen the abstracted guard, using the fact that the RAC cannot be busy if no request has been sent to the global directory. To ensure that the strengthening is sound, it has to be proved that when the local directory is not busy, the RAC of the cluster 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 per-level abstraction satisfies the requirement. Therefore, the verification obligation cannot be verified and the approach is not guaranteed to be conservative.

If the above rule can be modeled in a different manner, then the problem will disappear. For example, the right-hand side of Figure 3.17 shows two rules which achieve the same effect as that on the left-hand side. A new state variable *Req* is added to the cluster data structure, and it is also part of the interface. In the first rule, *Req* records the request, and in the second rule the request is forwarded to the global directory. After the per-level abstraction, in the abstract intercluster protocol, the guard of the first rule simply becomes $true$, the action sets the *Req*, and the second rule is abstracted to itself. In the refinement process for the first abstract rule, a new verification obligation will be added – whenever a local directory is not busy, the *Req* of the cluster contains no requests. This verification obligation can be proved by any abstract intracluster protocol. Thus the approach is conservative.

In summary, if different levels of a hierarchical protocol are modeled as loosely coupled, the per-level abstraction can be applied, so that more verification reduction can be achieved. As a result, we propose that hierarchical protocols be modeled as loosely coupled, when the performance allows the tradeoff.

3.3.3 Using History Variables

With the compositional approaches that we develop, in order to verify that a hierarchical protocol is correct with respect to its verification obligations, we need to verify that all the abstract protocols are correct with respect to their verification obligations individually. Moreover, for each verification obligation p in the hierarchical protocol, the corresponding abstracted verification obligation p_i in each abstract protocol needs to satisfy the following property: $p_i = p$ or $p_i = true$, and $\wedge_i p_i \Rightarrow p$ is valid.

For the inclusive cache hierarchy, the above can be achieved in a straightforward manner. For example, consider a commonly used coherence property: no two caches can write to the same address concurrently. For the inclusive multicore protocol, this property can be represented with two verification obligations:

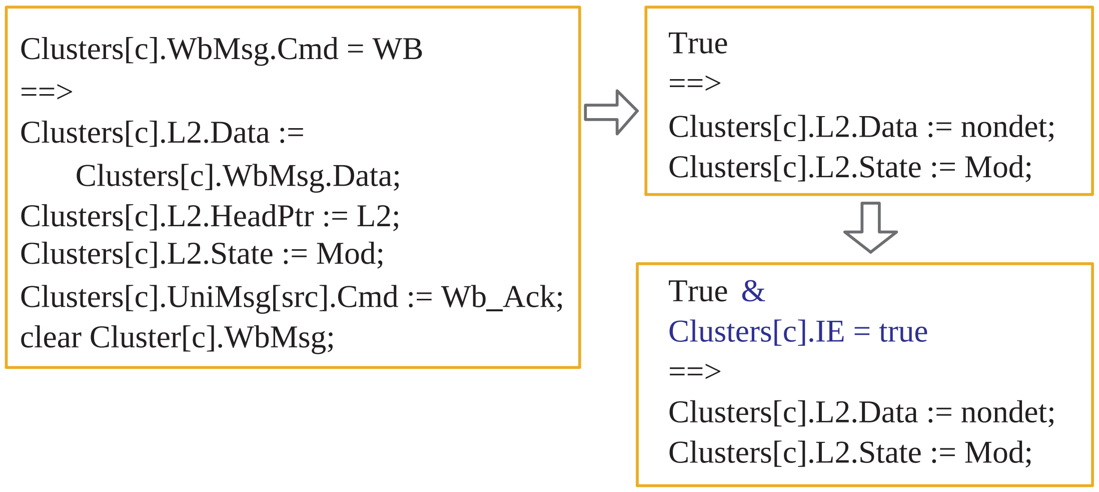
- No two clusters can have their L2 caches both be exclusive or modified;
- No two L1 caches can both be exclusive or modified in each cluster.

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 the abstractions that we develop. However, for noninclusive caches and multicore snooping protocols, when two

L1 caches from different clusters are both exclusive or modified, their corresponding L2 caches may not have the cache line. With the per-level abstraction, since every abstract protocol only maintains the details of at most one cluster, it is not straightforward how to represent the above coherence property.

Moreover, for noninclusive caches and multicore snooping protocols, it may not be straightforward to find out the appropriate constraints for guard strengthening, for spurious bugs detected during refinement. For example, consider the writeback example again shown in Figure 3.18 in the noninclusive protocol. For this example, when it is abstracted in the abstract protocol where the intracluster details are eliminated, it becomes 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 noninclusive caches. This is because when an L1 cache line is exclusive or modified, the corresponding L2 cache line can be invalid instead.

The above problems happen because in hierarchical protocols where noninclusive caches or snooping are employed in a level, there is no component that can represent as



```
Clusters[c].WbMsg.Cmd = WB
==>
Clusters[c].L2.Data :=
      Clusters[c].WbMsg.Data;
Clusters[c].L2.HeadPtr := L2;
Clusters[c].L2.State := Mod;
Clusters[c].UniMsg[src].Cmd := Wb_Ack;
clear Cluster[c].WbMsg;
```

```
True
==>
Clusters[c].L2.Data := nondet;
Clusters[c].L2.State := Mod;
```

```
True  &
Clusters[c].IE = true
==>
Clusters[c].L2.Data := nondet;
Clusters[c].L2.State := Mod;
```

Figure 3.18. The writeback example before/after abstraction, and during refinement.

a *summary* of that level. This makes it difficult to apply our compositional approach in a straightforward manner.

To solve the problems, we use history variables [90] in an assume guarantee manner. The basic idea underlying classical history variables is the notion of statifying the past, i.e., introducing auxiliary variables whose values in the current state reflect all the facts we need to know about the past. History variables were frequently used in reasoning about communication systems in the past [60, 66, 71], and they are also used in recent work for pipelined verification [76].

In our case, we introduce an auxiliary variable for each cluster in the multicore protocol with noninclusive caches and with snooping. The value of each auxiliary variable is a function of those that are already in the protocol. Together with the L2 cache line, the auxiliary variable is able to represent as a summary of the caching status in the cluster. Strictly, these auxiliary variables are not history variables but *current* variables, because each auxiliary variable is a function of a subset of the state variables already in the protocol, in the current state.

More specifically, for every cluster in the protocol, we add an auxiliary variable IE(implicit exclusive). The value of IE is defined in the abstract protocol where the cluster is retained as concrete, while it will be used in the abstract protocols where the cluster is abstracted. Initially, IE is set to false. In our second benchmark protocol, IE is defined to be true if one of the following conditions holds:

1. If an L1 cache contains an exclusive or modified copy, or
2. If 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. If there is a writeback or shared writeback request, or
4. If there is an exclusive ownership transfer request.

In our third protocol with snooping, IE is defined to be true if any L1 cache on the chip has an exclusive cache line. A verification obligation is added to the abstract protocol where IE is defined, to assert that the value of IE indeed follows the above definition.

When IE is true, it means that the cluster must have an exclusive or modified copy in the cluster, somewhere else other than in the L2 cache.

We also introduce a few other auxiliary variables to the noninclusive protocol and the snoop protocol in a similar way. After these variables are introduced and added into the interface variables, these protocols can be processed in the same way as for inclusive protocols. For example, for the problem on the coherence property, 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 (iii) no two L1 caches can both be exclusive or modified in every cluster. Each of the properties can be proved using either of our abstractions.

3.3.4 Experimental Results

We have applied the compositional approach with the per-level abstraction on the three benchmark protocols. Tables 3.2, 3.3 and 3.4 show the experimental results on the three coherence protocols, using the traditional monolithic model checking and our approach. In the tables, "Runtime" is counted in seconds, and "Mem" means how many GB's of memory are used in the experiment. The monolithic approach in Table 3.2 and 3.3 was performed on an Intel IA-64 machine employing 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.

For the multicore inclusive protocol, model checking on the original protocol using the monolithic approach failed after more than 0.4 billion of states, due to state explosion. Using our approach with the per-level abstraction, the inclusive protocol can be verified in about six minutes, with less than 2GB of memory. The multicore noninclusive protocol

Table 3.2. Verification complexity of the first benchmark protocol (II).

Approaches	Protocols	# of states	Runtime	Mem	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 3.3. Verification complexity of the second benchmark protocol.

Approaches	Protocols	# of states	Runtime	Mem	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 3.4. Verification complexity of the third benchmark protocol.

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

can be verified in about 20 minutes, with less than 2GB of memory. From the above tables, we can see that our approach can reduce more than 95% of the state space of the original protocols.

3.3.5 Soundness Proof

The soundness of our approach on verifying hierarchical protocols one level at a time can be derived from that of our compositional approach described in Section 3.2.4. First of all, Theorem 1 of Section 3.2.4 still holds here. This is because the conditions of (i), (ii), (iii) and (iv) 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 new approach despite the use of history variables. This is because the value of each history 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 history variables does not introduce any new behavior to the protocol. Moreover, whenever a history variable is used by a constraint e_i for guard strengthening for an abstracted rule $g_i \rightarrow a_i$, a new verification obligation $g' \Rightarrow e_i$ will be added to an abstract protocol, where g is the guard of the original rule and $g \Rightarrow g'$ is valid. Therefore, the soundness of our approach can be derived from Theorem 2.

3.4 Error-trace Justification

To mechanize our compositional approach described in the previous sections, there are altogether three steps of work to be automated. Referring to Figure 3.6, the three steps are: (i) given a hierarchical protocol and the protocol details to be abstracted away, how to automatically generate the abstract protocol; (ii) once an abstract protocol is obtained and model checked, when an error is found, how to automatically identify if it is a spurious or a genuine error in the original hierarchical protocol; and (iii) if the above error is identified as spurious, how to automatically constrain and refine the abstract protocols.

In Section 3.3.1, we have presented an approach to mechanizing the first step. For the third step, recent work [27, 136] from Intel has proposed two different approaches to mechanizing the step. More precisely, their approaches are proposed for the same problem of our third step, i.e., automatic refinement of abstract protocols, for the work of Chou et al. [42] on parameterized verification of nonhierarchical cache coherence protocols. In [136], the authors propose to use the ordering information in message flows of cache coherence protocols to construct new verification obligations for parameterized verification. In [27], the author assumes a Galois connection between the concrete and abstract protocols, and uses symbolic methods to automatically produce new verification obligations provable in the Galois connection.

In this section, we will present a solution to mechanize the second step that requires deep protocol understanding, i.e., human insights. We develop guided search to find a *stuttering equivalent* execution of the abstract error trace in the concrete protocol, and we will present two optimizations that can limit the scope of the transitions to be examined in the guided search. 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 the new methods described here, have handled the problem of error trace justification for hierarchical coherence protocols.

Given an error trace in an abstract protocol, let us first define whether this trace corresponds to a spurious or 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,

- $\forall vo \in A_i$, $vo(s_{kj})$ holds for every $j \in [0..m-1]$, and
- $p_k(s_{km})$ 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. In more detail, we say that E_k is a genuine error trace of M if $\exists E = s_0, s_1, \ldots, s_q$ ($q \geq 0$) of M such that $p_k(s_j)$ holds for every $j \in [0..q-1]$ while $p_k(s_q)$ does not hold. We say that E_k is a spurious error of M if E_k is not genuine.

3.4.1 Spurious/Genuine Error Identification

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 [45, 83] is defined over *Kripke structure*s [45] 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 3.19, 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 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 $E_i = u_{i,0}, \ldots, u_{i,m_i}$

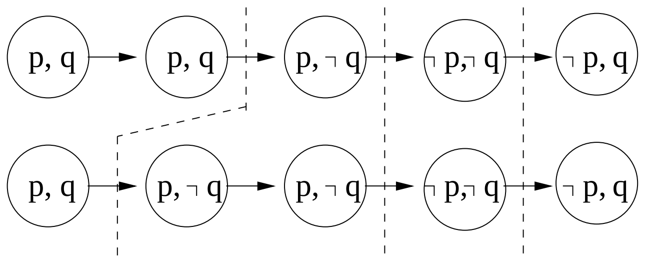


Figure 3.19. Stuttering equivalence.

of M_i, $i \in [1..n]$, $m_i \geq 0$, such that for every such i, there exists $0 = j_0 < j_1 < \ldots < j_{m_i}$ such that

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

$$u_{i,1} = s_{j_1} \mid V_i = \ldots = s_{j_2 - 1} \mid V_i,$$

$$\ldots$$

$$u_{i,m_i} = s_{j_{m_i}} \mid V_i = \ldots = s_m \mid V_i$$

In the rest of the dissertation, we will also denote every such E_i as a stuttering equivalent execution of E on V_i, and vice versa. We denote the index of the state in E which corresponds to the states in E_i as the *block index*: $b_i : S \rightarrow N$. That is, $b_i(s_0) = \ldots = b_i(s_{j_1 - 1}) = 0$, . . ., and $b_i(s_{j_{m_i}}) = \ldots = b_i(s_m) = m_i$, $i \in [1..n]$.

The above suggests that if E_i is an error trace of M_i, which violates a verification obligation p_i and corresponds to a genuine error of M, then E must be an error trace of M that violates p_i. Based on this, we develop a heuristic for error trace identification. A straightforward implementation is as follows – given an error trace from an abstract protocol, we use guided search on the original hierarchical protocol trying to find a stuttering equivalent execution. In the following, we will first describe the basic idea of the simple guided search, and then present two optimizations.

3.4.2 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 hierarchical protocol M trying to construct a stuttering equivalent execution, to identify the error trace. 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 $R \subseteq T$ at s. That is, for $\forall t \in R, t = g \longrightarrow a$ where $g(s)$ holds, $s' = a(s)$, and $b_i(s) = j$, $j \geq 0$, we consider three cases:

- $b_i(s') = j$;
- $b_i(s') = j + 1$;
- $b_i(s') \neq j, b_i(s') \neq j + 1$.

The first case means that s' is still in the same block as s, and the second means s' is in the next block than that of s. The third case means that the rule t updates certain variables of M so that it no longer matches either of the abstract states $u_{i,j}$ or $u_{i,j+1}$.

Based on the above analysis, we present a heuristics to find out whether there exists a stuttering equivalent execution of E_i in M, using the breadth-first search (BFS). In more detail, we associate each state of M with an integer which is the block index of E_i. The initial states of M that match with $u_{i,0}$ will have the the block index 0. For every state s of M in the BFS queue, we consider every next state s' of s. If s' satisfies either of the first two cases in the above, it will be added into the state queue of BFS. Otherwise, s' will simply be dropped. In the end, if a state s is found to have the block index m_i, then the error trace E_i must be genuine; otherwise we claim it as spurious.

Unfortunately, this simple approach does not work in practice because for every state s in the BFS queue, there are a huge number of next states s' where the index of s' is equal to that of s. For example, consider `Abs #1` in Figure 3.14. Since `Abs #2-4` have many private variables that do not intersect with the variables of `Abs #1`, we have the situation that – for any state s in the queue, 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 be added into the queue. Thus, the state space of the guided search is that of another hierarchical protocol, i.e., the state explosion problem still exists, and our initial experiments confirmed it.

3.4.3 Interface Aware Guided Search

In this section, we refine the simple guided search exploiting the *interfaces*. The idea is based on the observation that when a rule in the abstract error trace only updates the state variables of a cluster, it is sufficient for the guided search to explore only those rules updating that cluster.

In more detail, 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. Also, let $M_i = (V_i, I_i, T_i, A_i), i \in [1..n]$ be all the abstract protocols of M. From Section 3.3, our abstraction has the property that $\cup_{i \in [1..n]} V_i = V$ when history variables (i.e., 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. Specifically, for the protocol as shown in Figure 3.1, we have four abstract protocols (two of them are identical due to symmetry) as in Figure 3.14. Let M_1, M_2, M_3 be the three abstract intracluster protocols, and M_4 be the abstract intercluster protocol, i.e., M_i is `Abs #i`. According to our definition, $V_i \cap V_4, i \in [1..3]$ are the interfaces for the three clusters respectively. In the following, for this protocol, we will denote $V_i, i \in [1..3]$ as the *intra cluster* variables, and $V_i \setminus (V_i \cap V_4), i \in [1..3]$ as the *inner cluster* variables. We will also denote $V_4 \setminus \cup_{i \in [1..3]} V_i$ as the *outer cluster* variables.

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. More formally, 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.

The above idea is based on the observation that M_i only interacts with other abstract protocols through its interfaces. Thus, if the updates $\delta_{i,j}$ do not involve interface variables, the guided search should not either.

More specifically, consider the protocols in Figure 3.1 and Figure 3.14. For M_4, if $\delta_{4,j}$ only involves the interfaces of a cluster, then the guided search will only explore the rules in M that update the intra cluster variables for *that* cluster. Otherwise, the guided search will restrict to explore the rules which 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 nonhierarchical protocol, before matching $u_{i,j+1}$.

Similarly we can process error traces from abstract intracluster protocols $M_i, i \in [1..3]$ as in Figure 3.14. The difference is that if $\delta_{i,j}$ involves the interfaces of that cluster, the guided search may have to consider the updates of the whole hierarchical protocol. This is because in M, the variables in $V \setminus V_i$ may have to be updated, before reaching a state that matches $u_{i,j+1}$. Thus, it is possible that the state space of the guided search can still be very large.

To solve the above problem, we develop the following solution. Since the interface aware approach can work efficiently for "inter cluster abstractions" (e.g., M_4), we will model check this protocol first, with guided search and counterexample guided refinement. Thus, before working on the abstract intracluster protocols, we would have refined M_4 in a manner such that all its verification obligations hold. Therefore, for M_1 in Figure 3.14, the guided search can replace the original protocol with that in Figure 3.7. For the original protocol in Figure 3.1, there will be three such protocols as in Figure 3.7, and we denote them as $M_{Ai}, i \in [1..3]$.

More formally, $M_{Ai} = (V_{Ai}, I_{Ai}, T_{Ai}, A_{Ai}), i \in [1..3]$, can be defined by M, M_i and M_4 as follows. First, $V_{Ai} = V_4 \cup V_i$ and $A_{Ai} = A_4 \cup A_i$. Second, 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, $T_{Ai} = \{t_4 = g_4 \longrightarrow a_4 \mid t_4 \in T_4, \forall s_4 \in S_4, a_4(s_4) \mid \mathrm{If_i} = s_4 \mid \mathrm{If_i}\} \cup \{t = g \longrightarrow a \mid t \in T, \exists s \in S, a(s) \mid V_i \neq s \mid V_i\}$. Here, the first part of T_{Ai} 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.

For the above replacement, there exist two questions with regard to why and how the replacement can be done. Intuitively, given an error trace E_i from an abstract intracluster protocol M_i, if with the original multicore protocol M, our guided search can find a stuttering equivalent execution that reports E_i as genuine, then by replacing M with M_{Ai}, we want the guided search also report E_i as genuine. The replacement is a good heuristics (as will be shown in Section 3.4.5) because of our compositional approach has the following property. The abstract intercluster protocol $M_4 = (I_4, V_4, T_4, A_4)$ overapproximates the original protocol on the state variables V_4. Also, any abstract intracluster protocol only interacts with the rest of the protocol through the interface which is a subset of V_4. Thus, it is a good heuristics to replace M with M_{A_i}.

More specifically, suppose $E_i = u_{i,0}, \ldots, u_{i,m_i}, m_i \geq 0$ is an error trace from an abstract intracluster $M_i = (I_i, V_i, T_i, A_i)$. Also, suppose there exists a stuttering equivalent execution $E = s_0, \ldots, s_{n0}, s_{n0+1}, \ldots, s_{n1}, \ldots, s_{nm}$ of M and E_i is reported as genuine, $n0, n1, \ldots, nm \geq 0$. That is, $s_0 \mid V_i = \ldots = s_{n0} \mid V_i = u_{i,0}$, $s_{n0+1} \mid V_i = \ldots = s_{n1} \mid V_i = u_{i,1}$, …, $s_{nm} \mid V_i = u_{i,m_i}$. Now because M_{Ai} overapproximates M on $V_{Ai} = V_i \cup V_4$, $E_{Ai} = s_0 \mid V_{Ai}, \ldots, s_{nm} \mid V_{Ai}$ is a valid execution of M_{Ai}. Thus, E_{Ai} is a stuttering equivalent execution of E_i.

On the other hand, we can think M_{Ai} as an abstract protocol of M, and M_i as an abstract of M_{Ai}. Thus, we can use the 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}, then trying to find a stuttering equivalent execution of E_{Ai} in M.

For the second question as for how to construct M_{A_i} in Figure 3.7, we can simply replace the data structures, the variable declarations, and the rules involving the remote clusters, with that in the already refined abstract intercluster protocol. Such replacements are very routine, and they do not require any understanding of the protocol details. The process has some similarities with that of generating the abstract protocols (as in Figure 3.14) from the original multicore protocol (as in Figure 3.1), which we have mechanized. Currently the replacement is done manually, and we hope to finish automating it soon.

3.4.4 Interface Aware Bounded Search

In the previous section, we optimize the straightforward approach for searching for a stuttering equivalent execution of an abstract error, with interfaces. According to the definition, it is clear that interfaces can be directly obtained once the original hierarchical protocol is abstracted. Thus, the interface aware approach does not require more human insights than the abstraction procedure. We can further improve the interface aware approach in the previous section with *bounded* search. That is, given an abstract error trace $E_i = u_{i,0}, u_{i,1}, \ldots, u_{i,m_i}, m_i \geq 0$, we limit the BFS depth of the guided search for each block index i, $i \in [0..m_i]$. Ideally, we want the bound to have the property that if the bounded search cannot find a stuttering equivalent execution, then the guided search without bound cannot either.

More specifically, we define a slightly different bound than the ideal bound as follows. 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. Now we associate every such tr_j with a bound. There are two cases:

1. There exists an execution $s_k, s_k + 1, \ldots s_l$ in the original multicore protocol M, $k, l \geq 0$ 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 could be multiple such executions in M for tr_j, and we define the bound of tr_j to be the minimum number $l - k$. That is, we choose the bound of tr_j, $bnd(tr_j)$, to be the length of the shortest stuttering equivalent execution of tr_j in M.

2. If there does not exist a stuttering equivalent execution of tr_j in M, we define the bound of tr_j to be any natural number, e.g., $bnd(tr_j) = 1$.

Finally, we define 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 definition, we select the value of bounds as follows. Consider any transition $tr = s \xrightarrow{t} s'$ in the abstract intercluster protocol. Case 1: tr only updates the state variables not in interfaces, i.e. the outer variables. Then if there exists a stuttering equivalent execution of tr in the original multicore protocol, our interface aware guided search will only explore the rules which update the outer cluster variables. Thus $bnd(tr) = 1$. 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', which is most likely between a request and a reply which involve interface variables.

For example, consider a transition in the abstract intercluster 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 in this cluster and also no other pending requests, so it places a shared request to be sent to the global directory. Thus, the bound of this transition is 3.

Similarly we can set bounds for each abstract intracluster protocol. The only difference is that for a rule from a request is placed on an cluster, to a reply is received from outside the cluster, the stuttering equivalent execution may involve updates from all the other clusters, the main memory, and the global directory. Fortunately, the replacement that we developed in Section 3.4.3 as shown in Figure 3.7 can help solve this problem.

In theory, choosing a bound as in the above requires deep protocol understanding. Fortunately, in practice we find that for high level protocol descriptions e.g., in Murphi, the value of the candidate bound is reasonably small, where the value 10 or 15 is usually big enough. For the three multicore protocol benchmarks presented in Section 3.1, we find that the bound 10 works perfectly for all of them.

3.4.5 Implementation and Experimental Results

We have implemented the interface aware bounded search method based on the Murphi distribution. The tool is available for downloading at [5]. We overwrote the simulation option of the model checking provided by Murphi, with the interface aware bounded search. Our interface aware bounded search is basically a breadth-first search. Figure 3.20 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.

```
GUIDEDSEARCH($AbsStates$[], $N$) {
  let $s$ = ChooseStartState($AbsStates[0]$);
  i = 0;
  **while** $(i < N - 1)$ {
    let $\delta_A = AbsStates[i+1] - AbsStates[i]$;
    let $outer\_move_A$ = is_outer_move($\delta_A$);
    clear($Q$); enqueue($Q, s$);
    $bound = 0$;
    **while** ($bound < BOUND$) {
      **while** ($\neg isEmpty(Q)$) {
        $ss$ = dequeue($Q$);
        **for each** enabled rule $r$ at $ss$ {
          let $ss' = r(ss)$;
          let $\delta = ss' - ss$;
          let $outer\_move$ = is_outer_move($\delta$);
          **if** ($outer\_move_A \neq outer\_move$) **continue**;
          **if** ($\neg outer\_move_A$ and
              $\neg$ updateSameCluster($\delta_A, \delta$))
            **continue**;
          **if** (isProjection($AbsStates[i+1], ss'$)) {
            $s = ss'$;
            clear($Q_n$); **goto** L1;
          }
          **if** (isProjection($AbsStates[i], ss'$))
            enqueue($Q_n, ss'$);
        }
      }
      $Q = Q_n$; clear($Q_n$);
      $bound$ ++;
    }
    **return** false;
    **L1**: $i$ ++;
  }
  **return** true;
}
```

Figure 3.20. Interface aware bounded search.

Based on the error trace, the algorithm tries to find a stuttering equivalent execution in the original hierarchical protocol or the protocol as in Figure 3.7. 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 is_outer_move 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 3.14, the file should contain all the the state variables of `Abs #4` excluding the interfaces (i.e., the L2 caches). While for the abstract intracluster 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 intracluster 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.

We have applied the interface aware bounded search to the three multicore coherence protocol benchmarks as described in Section 3.1, and they can be accessed at [5]. Before applying the guided search, we had previously verified our three benchmark protocols, 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 intercluster level and one in the intracluster 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 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 (see Figure 3.2).

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 3.21 shows

```
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 3.21. A spurious error trace and the result from our approach.

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. In the following we describe the remaining five cases of spurious error traces, which 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. That is, let $t_1 = g_1 \rightarrow a_1$, $t_2 = g_2 \rightarrow a_2$ be two rules of $M = (V, I, T, A)$ where $\exists s \in S$ such that $a_1(s) \neq a_2(s)$. In an abstract protocol $M_i = (V_i, I_i, T_i, A_i)$, it happens that $a_1(s) \mid V_i = a_2(s) \mid V_i$ for $\forall s \in S$, and the abstracted guard of g_1 implies that of g_2. We denote such t_1 as an *abstract equivalent* rule of t_2 in M. Because of abstract equivalent rules, our approach can report an execution which seems different from the provided abstract error trace. For example, in the output of guided search in Figure 3.21, the second rule can be an abstract equivalent rule of *L2_inReq_OutReq*. In our experiments, 3 of 101 cases fall into this category, and it is straightforward for users to realize them.

The second category is about the history variables which are introduced in our compositional approach (see Section 3.3.3) for the second and the third benchmarks. The situation is that in an abstract error trace, certain rules only update the history variables while these variables do not exist in the original protocol. As a result, the guided search may report a stuttering equivalent execution containing irrelevant rules than those in the abstract error trace. For example, suppose in Figure 3.21 the third rule of the abstract error trace of M_i only updates the history variables. Then it is possible that the third rule in the guided search output is any rule $t = g \longrightarrow a$ of M where $a(s) \mid V_i = s \mid V_i$ for $\forall s \in S$. In our experiments, one case falls into this category.

Finally, the third category is a little more complicated. Because each abstract protocol M_i overapproximates M on V_i, it is possible that for some executions of M_i, there do not exist stuttering equivalent executions in M. If such an execution is part of an abstract error trace, our guided search will not be able to find a stuttering equivalent execution in M, i.e., it will report spurious. In our experiments, one case falls into this category.

In this case, the abstract intracluster protocol from the second benchmark has an abstract error trace of length 10, which begins with the following three rules:

1. The remote cluster is in the invalid state, and it receives an exclusive request from outside the cluster;

2. The remote cluster starts processing the exclusive request;

3. The remote cluster receives a shared request again from outside the cluster.

For this trace, the problematic rule is on the 3^{rd} rule because the remote cluster has not finished processing the first request yet. For this error, there does not exist a stuttering equivalent execution of the multicore protocol M beginning with the initial state and can match the 1^{st} and the 2^{nd} rules. The reason is that at the initial state of M, only the main memory has a valid copy of the data. So it is impossible for the remote cluster to receive any request from others.

However, there does exist an execution of M that can match rules 1 and 2 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 1 and 2 will execute. For the abstract error trace to be stuttering equivalent to the above execution of M, the abstract error should begin with the following six rules.

1. The remote cluster is in the invalid state, and it requests an exclusive copy;
2. The remote cluster receives the exclusive copy;
3. The remote cluster writes back the exclusive copy to the main memory;
4. The remote cluster is in the invalid state, and it receives an exclusive request from outside the cluster;
5. The remote cluster starts processing the exclusive request;
6. The remote cluster receives a shared request again from outside the cluster.

From the above, it is clear that the abstract protocols can have more behavior than the original protocol on the state variables that are maintained in the abstract protocols, i.e., each abstract protocol overapproximates the original protocol. Therefore, when an abstract error trace begins with the behavior (a sequence of rules) which does not exist in the original protocol, there will not exist any stuttering equivalent execution in the original protocol. This also 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 cases, found the exact location of the bug automatically. Without our approach, designers are required to identify every such error trace, while with our approach, once the configuration file and the bound are available, our tool can automatically identify these errors.

3.5 Summary

In this chapter, 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 multicore protocols with different features: inclusive and noninclusive cache hierarchies, and with snooping. These protocols are modeled with certain realistic features so that their verification complexity is similar to those that are used in practice.

Based on these benchmarks, we develop two novel compositional approaches. Given a hierarchical protocol, both approaches first decompose it into a set of abstract protocols with smaller verification complexity. The abstract protocols are then refined in an assume guarantee manner. Both approaches are conservative in that if all the abstract protocols can be verified correct, the original protocol must be correct as well.

One difference between these approaches is that in the second approach, we decompose hierarchical protocols one level at a time. In order to apply this approach, we also propose that hierarchical protocols be developed and modeled in a loosely coupled manner. Furthermore, the second approach extends the first one, by using history variables in an assume guarantee manner. This extension makes it possible for the second approach to verify all the three benchmarks, not just the inclusive one with the first approach.

For the benchmarks, we show that our second approach can reduce the verification complexity of the multicore protocols, to that of a nonhierarchical protocol. Finally, we also develop a set of heuristics to automatically identify whether an error trace produced from an abstract protocol corresponds to a genuine error in the original hierarchical protocol or not. For all the three benchmarks, we show that our approach is very effective in identifying all the genuine and spurious error traces. For future work, we plan to investigate how to combine the automatic guard strengthening for refinement from others, with our error trace identification.

CHAPTER 4

REFINEMENT CHECK BETWEEN PROTOCOL SPECIFICATIONS AND IMPLEMENTATIONS

In the previous chapter, we develop a set of techniques to scale the formal verification of hierarchical cache coherence protocols in their high level descriptions. In practice, verification of coherence protocols only at the high level is insufficient. The semantic gap between high level protocol descriptions and their hardware implementations is nontrivial. In this chapter, we will develop methodologies to check whether a hardware implementation of a coherence protocol correctly implements its high level description.

Nowadays, with the growing complexity of the internal organization of modern processors and other digital systems, it is important to develop notations, verification methodologies and tools that ensure correctness as well as high performance of the overall design, and allow designers to make design choices. To put things into sharper focus, consider the design of a modern high performance cache coherence protocol as an example. A designer initially conceptualizes the design in terms of atomic transitions that fire under certain conditions, and move request/response packets between various directories, caches, and processor cores. Accumulated experience, starting from the early work of Yang et al. on UltraSparc-1 [141] to more modern ones [21, 41, 57], shows that designs captured at the high level descriptions can be model checked to help eliminate high level concurrency bugs.

It is also widely known that the atomic transitions used at the specification level are implemented in hardware over multiple clock cycles (a *transaction* in our terminology), with one or more implementation steps happening in each clock cycle of the transaction. In addition, while the specification level models the desired computation according to the *interleaving* model where only one specification transition fires at a time, the implementation level often starts a second transaction before the first transaction has finished, again to maximize overlapped computation, pipelining, and to take advantage of internal buffers and split transaction buses. In today's design contexts, designers are seriously under-equipped concerning (i) notations that allow them to specify the designs of such aggressively optimized implementations, (ii) theories for formally relating such implementations against specifications, and (iii) compositional methods for verifying implementations against specifications that have already been verified at the interleaving model level for global properties.

In this chapter, we will develop a formal theory of refinement, showing how a collection of such implementation transitions can be shown to realize a specification. We will also develop a modular refinement verification approach by developing abstraction and assume guarantee principles that allow implementation transitions realizing a single specification transition to be situated in sufficiently general environments. Finally, we develop a tool called *Muv* together with researchers from IBM to check the refinement between protocol specifications and their hardware implementations. For refinement check, we first extend the Murphi description language with hardware features, and we call it *Hardware Murphi*. In more detail, we assume that each transition (rule) in Hardware Murphi once enabled, will take one clock cycle to finish the execution. Also, top level rules in Hardware Murphi are implicitly concurrent, i.e., in each clock cycle all the enabled rules can fire concurrently. Given a protocol specification and its hardware implementation in Hardware Murphi, Muv will automatically translate them into a synthesizable VHDL model, and also generate a set of assertions for the refinement check. Applying our refinement check to a driving coherence protocol benchmark with realistic features, we show that our approach can generate enough assertions to catch the bugs in the RTL implementation where these bugs are easy to miss by writing assertions.

This chapter will be organized as follows. We will first present some preliminaries to the formal definition of our refinement check. We then present how to check the refinement, and describe a monolithic approach for checking the refinement. We will then present a compositional approach which can scale the monolithic refinement check, and finally describe the tool Muv.

4.1 Some Preliminaries

4.1.1 States, Transitions, Models

Let V be a set of variables and D be a set of values which are both nonempty. A *state* s is a function from variables to values, $s : V \rightarrow D$. For state s, variable v, and value $\alpha \in D$, $s[v \leftarrow \alpha]$ denotes the state that is the same as s except $s[v \leftarrow \alpha](v) = \alpha$. 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 $g \longrightarrow a$ to denote the transition with guard g and action a. We can also form the action of a transition by composing two or more functions. If $a, b : S \rightarrow S$, then $g \longrightarrow a; b$ denotes the transition $g \longrightarrow \lambda s.b(a(s))$.

We associate with each transition $t = (g \longrightarrow a)$ two sets of variables called the read and write sets of t, $R(t)$, $W(t)$, respectively. The write set of t is the set of variables whose value can be changed by the transition in some state, $W(g \longrightarrow a) = \{v \mid \exists s \in S : g(s) \wedge a(s)(v) \neq s(v)\}$. The read set of a transition is the set of variables that can affect either the enabling of the guard, or the value written to one of the variables in the write set, $R(g \longrightarrow a)$ is

$$\begin{array}{lll}\{v & \mid & \exists s \in S, \exists \alpha \in D : (g(s) \not\equiv g(s[v \leftarrow \alpha])) \vee \\ & & (a(s) \neq a(s[v \leftarrow \alpha]))\}\end{array}$$

We say $t\ reads\ v$ if $v \in R(t)$; $t\ writes\ v$ if $v \in W(t)$.

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 assertions (state formulas over V) which can be empty.

4.1.2 Executions

We consider two notions of execution for models. An *interleaving* execution of a model is based on steps in which a single enabled transition fires. For a transition $t = (g \longrightarrow a)$, we write $s \xrightarrow{t} s'$ to denote that $g(s)$ holds and $s' = a(s)$. An interleaving 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 $s_i \xrightarrow{t_i} s_{i+1}$.

We now introduce the *concurrent* executions of a model. First we define the concurrent firing of a set of transitions. If $t_1 = (g_1 \rightarrow a_1), t_2 = (g_2 \rightarrow a_2)$ are two transitions with $W(t_1) \cap W(t_2) = \emptyset$, we write $s \xrightarrow{\{t_1,t_2\}} s'$ to denote that $g_1(s)$ and $g_2(s)$ hold, and $s' = (a_1(s) \mid W(t_1)) \cup (a_2(s) \mid W(t_2)) \cup (s \mid V - W(t_1) - W(t_2))$. In the preceeding, we form the function s' by taking the union of restricted functions. We define the notion of the concurrent firing of a set E consisting of more than two transitions similarly, and write it as $s \xrightarrow{E} s'$.

The idea of a concurrent execution of a model is that all of the enabled transitions fire at each step. However, we have to deal with the problem of write conflicts between different transitions. This is because if two transitions t_i, t_j are both enabled at s, and they both write a variable v, then the value of v cannot be defined at the next state. For a state s such that two transitions t_1, t_2 are enabled, and $W(t_1) \cap W(t_2) \neq \emptyset$, we say there is a write-write conflict at s. When a write-write conflict occurs at s, for the ease of exposition, we will define execution to simply stay at s (we can also define this as an error). Thus, a concurrent execution of a model (V, I, T, A) is a sequence of states $s_0, s_1, \ldots$, such that $s_0 \in I$ and for all $i \geq 0$, $s_i \xrightarrow{E_i} s_{i+1}$. Here, E_i is the set of *all* transitions (maximal set) enabled at state s_i if there is no write-write conflict at s_i, i.e., $s_i \xrightarrow{E_i} s_{i+1}$. Otherwise, $s_i = s_{i+1}$.

Now we define *labelled* executions. Given an execution $S_e = s_0, s_1, \ldots$ of a model, a labelled execution of S_e is a sequence: $s_0 \xrightarrow{E_0} s_1 \xrightarrow{E_1} \ldots$. If S_e is an interleaving execution, then for each $i \geq 0$, E_i is a transition with $s_i \xrightarrow{E_i} s_{i+1}$. Otherwise if S_e is a concurrent execution, E_i is a set of transitions (defined, as above, to be maximal set) such that $s_i \xrightarrow{E_i} s_{i+1}$, or $s_i = s_{i+1}$. In executions we may write t for the singleton set of transitions $\{t\}$ if the context makes the intended usage clear.

4.1.3 Annotations

A transition may be annotated by associating formulas for preconditions and postconditions with the transition. We write $g \rightarrow \{P\}a\{Q\}$ to denote an annotated transition $g \rightarrow a$ with precondition P and postcondition Q. The formulas P, Q are formulas over the variables of the model. We can omit $\{P\}$ or $\{Q\}$ in the cases when it is equivalent to

$\{true\}$. For an execution $S = s_0 \xrightarrow{E_0} s_1 \xrightarrow{E_1} \ldots$, we say that S satisfies the annotation $g \rightarrow \{P\}a\{Q\}$, if whenever $s_i \xrightarrow{E_i} s_{i+1}$ and E_i contains (if S is a concurrent execution) or is (if S is an interleaving execution) the transition $g \rightarrow a$, then $P(s_i)$ and $Q(s_{i+1})$ both hold. The intuition is that if a transition fires in an execution, then both the precondition and the postcondition must hold. An annotated model is one which has one or more annotated transitions.

Let $M = (V, I, T, A)$ be a model and p be a formula. For an execution (either interleaving or concurrent) $S_e = s_0, s_1, \ldots$ of M, we say S_e satisfies p if $p(s_i)$ holds for every $i \geq 0$. If M is a model in which only interleaving executions are considered, we say M satisfies p if every interleaving execution of the model satisfies p. We denote this by $M \models p$. Similarly, we define $M \models p$ for a model M where only concurrent executions are considered. For an annotated transition t of M, we also write $M \models t$ to denote that every execution of M satisfies t. Finally, if M satisfies its set of annotated transitions and all the assertions of A at every reachable state, then $\models M$ holds.

4.1.4 Transactions

We are interested in showing the correspondence between implementation and specification models, where the implementation may take many steps to accomplish the work of one step in the specification. To help define the correspondence between an implementation and specification, we introduce the notion of a *transaction*.

The purpose of a transaction is to collect the implementation transitions that correspond to a single transition of the specification. Each transaction is a set of transitions, and it has a unique state variable called the *alive* variable. In an initial state of the model, all of the alive variables for transactions are set to false. No transitions in a transaction are enabled until a transition called the *trigger* of the transaction is fired. If the transaction takes multiple steps to finish, then the alive variable is set to true. Once the guard of the trigger transition is enabled, other transitions in the transaction can fire. The other transitions can continue to fire until a *closing* transition resets the alive variable to false. Such a closing transition is assumed to always exist, and it could be the same as the trigger.

We now describe transactions more precisely. A transaction is a set of transitions, formed as follows. Let $t = (g \longrightarrow a)$ be a transition, and T be a possibly empty set of transitions. Then $U = \mathrm{transaction}[t; T]$ is the following set of transitions:

$$\begin{array}{l}\{(g \wedge \neg alive) \longrightarrow (a; alive \leftarrow MultiCycle)\} \bigcup \\ \cup_{(g_i \longrightarrow a_i) \in T} \{(g_i \wedge (g \vee alive)) \longrightarrow a_i\}\end{array}$$

This is the only type of transaction that we will consider, i.e., it is a transition t plus a set of transitions T. In the above formula, $alive$ is the alive variable unique to the transaction U, and t is the trigger transition. The first half of the formula says that the trigger transition fires when its guard g is true and the alive variable is false. Firing the trigger transition updates the state by the action a. The function $MultiCycle$ is true in a state if U will take more than one cycle. Every transaction U corresponds to a $MultiCycle$ function. The alive variable is set to true if the transaction takes multiple steps.

A transition $t_i = g_i \longrightarrow a_i$ in T is confined to fire within the scope of the transaction U. More specifically, t_i fires when its guard g_i is true and either of the following condition holds:

1. The alive variable of U is false and the trigger guard is true.

2. The alive variable is true.

Case 1 above occurs when the transaction U becomes alive, i.e., transitions in T are permitted to fire on the same step as the trigger transition. Case 2 occurs when the alive variable is true, i.e., transitions in T can fire after the trigger transition, till (including) the step of the closing transition.

For any transition $t = (g \longrightarrow a)$ in a transaction, we use $enable(t)$ to represent the conditions under which t is enabled. According to the above definitions, $enable(t) = g \wedge \neg alive$ if t is the trigger transition. Otherwise, $enable(t) = g \wedge (g_\tau \vee alive)$, where τ is the trigger transition of the transaction and g_τ is its guard.

As said earlier, transactions provide a way of organizing the transitions that realize the operations by the hardware into mutually exclusive sets of transitions. However, note that during execution, the execution semantics are defined in terms of the concurrent executions generated by $\cup_i U_i$ where U_i is a transaction. This means that transitions from different transactions can fire concurrently.

In the sequel, we study read-write relations between concurrent transactions in a way that is analogous to concurrency and variable access in threaded programs. We define the read and write sets of a transaction U_i to be $R(U_i) = \cup_{t \in U_i} R(t)$, $W(U_i) = \cup_{t \in U_i} W(t)$, respectively.

4.1.5 Implementation Model

We define an *implementation* model, or low-level model, to be a model $M_L = (V_L, I_L, T_L, A_L)$, where T_L is the union of a set of transactions, $T_L = \cup_i U_i$, and the transactions U_i are pairwise disjoint sets of transitions. This is the structure that we will assume for implementations. By pairwise disjoint, we mean that even if two transactions U_i and U_j can contain the same transition t, because each transaction associates with its own alive variable and a trigger transition, the conditions under which t can fire in U_i and U_j are different. Therefore it is reasonable for us to assume that transactions are pairwise disjoint sets of transitions.

We say that a variable v is *active* in state s in an implementation model, if there is a transaction U with $v \in W(U)$, and the alive variable for U is true in s. For a state s, let $\text{active}(s)$ be the set of all variables active in s, and $\text{inactive}(s)$ be the set of model variables not in $\text{active}(s)$.

4.2 Checking Refinement

We consider three kinds of variables in the implementation model and discuss how each kind of variable corresponds to the specification. An *interface variable* must match the specification model “at all times.” A *transactional variable* must match the specification model except when the variable is active. The specification for a transactional variable is comparable to a resource invariant [112] in a shared variable program with critical sections. The similarity is that a transactional variable in our theory and a shared variable used in critical sections [112] both have a specified value when the variable is not being updated. Finally, the implementation can have additional variables that are not present in the specification and are not constrained.

In the above, interface variables are introduced for certain applications to augment our refinement check with additional invariance properties. For the coherence protocol as to be described in Section 4.6.4, the implementation model contains transactional variables but no interface variables.

For transactional variables, consider an analogy to the approach of Burch-Dill flushing [32]. In their approach, one needs to provide an abstraction function that maps states in the implementation model to specification model states. Such a function maps "unclean" implementation states into the final clean state that is produced by the transaction. In this sense, our transactional variables are similar to the state variables in their implementation model. However, our approach is applicable without the assumption that the system can be driven into a globally inactive state.

Returning to the definition of refinement check. Let $M_H = (V_H, I_H, T_H, A_H)$ be a model and $M_L = (V_L, I_L, T_L, A_L)$ be an implementation model over variables V_H, V_L respectively, where $V_H \subseteq V_L$. Let $V_I \subseteq V_H$ be the set of interface variables and $V_T \subseteq V_H$ be the set of transactional variables. We say that M_L implements M_H with interface variables V_I and transactional variables V_T if for every concurrent execution $l_0, l_1, \ldots,$ of M_L, there exists an interleaving execution $h_0, h_1, \ldots,$ of M_H, and an increasing sequence of natural numbers $n_0 < n_1 < \ldots$ such that for all $i \geq 0$,

$$\begin{array}{ll} & (l_i \mid V_I = h_{n_i} | V_I) \\ \wedge & (l_i \mid (V_T \cap \text{inactive}(l_i)) = h_{n_i} \mid (V_T \cap \text{inactive}(l_i))). \end{array}$$

In this definition, step i of the implementation is matched by the specification at step n_i, and variables in V_I plus those in V_H that are inactive must have the same value in both models. If M_L implements M_H, then whenever M_L reaches a cleanly halted state with no transactions in progress, all of the variables in V_I and V_T must be in a state reachable in the specification M_H.

In the above definition, we require that for every concurrent execution of M_L, there exists an *interleaving* execution of M_H. The requirement of the interleaving semantics on M_H is because in this work, we only consider specifications modeled in high level languages in terms of interleaving atomic steps, in guard/action languages such as Murphi or TLA+. Such specifications have been used in modern industrial practice since the 1990s.

This definition can be augmented with additional invariant assertions to express stronger properties. For example, one invariant could be that n_i is the number of transactions in M_L that have completed at l_i. Another example could be that for all the transitions which are fired between h_{n_i} and $h_{n_{i+1}}$, their write sets are disjoint.

Proposition 1. Suppose M_H is a model and M_L is an implementation model such that M_L implements M_H with V_I and V_T. If $M_H \models P$, for an invariance property P, then $M_L \models \mathrm{inactive}_P \Rightarrow P$, where $\mathrm{inactive}_P$ is a formula which asserts that all the transactional variables in $\mathrm{vars}(P)$ are inactive. The formula $\mathrm{inactive}_P$ can be expressed using the alive variables of transactions.

The above proposition holds because when a transactional variable in $\mathrm{vars}(P)$ is inactive, it must match that in the specification model. Also, from the definition, every interface variable must match that in the specification model. So $M_H \models P$ implies $M_L \models \mathrm{inactive}_P \Rightarrow P$.

4.3 Monolithic Model Checking

The refinement of the specification model M_H by the implementation model M_L can be checked in a straightforward way by constructing a *monolithic checking* model, called M_{MC}. M_{MC} defines all the possible executions of the implementation, and for each execution, asserts that the states reached by the implementation can also be reached through specification transitions. In effect, M_{MC} executes a cross product construction. In Section 4.4, we describe a compositional checking method that allows much of the checking to be performed on smaller models.

Let $M_H = (V_H, I_H, T_H, A_H)$ be a specification model and $M_L = (V_L, I_L, T_L, A_L)$ be an implementation model, where $V_H \subseteq V_L$. Let V'_H be a fresh set of variables, with one new variable corresponding to each variable in V_H. For a set of variables V, V' is the result of replacing all variables from V_H in V with the corresponding primed variable. That is, for $V \subseteq V_L$, $V' = \{v' \mid v \in V \cap V_H\} \cup (V - V_H)$. For $V \subseteq V_L - V_H,\ V' = V$. In addition, priming of expressions is defined to be the priming of the free variables of the expression.

We define the monolithic model as follows, $M_{MC} = (V_L \cup V_H', I_{MC}, T_{MC}, A_{MC})$. The variables V_L will be updated according to the implementation model M_L, while the variables V_H' will be updated according to the specification model M_H.

We will check that for each initial state of the implementation, there exists an initial state of the specification that matches on the variables in V_H, that is, $(I_L \mid V_H) \subseteq I_H$. Then we define the initial states of M_{MC} to be $I_{MC} = \{s_L \cup (s_L \mid V_H)' : s_L \in I_L\}$. Thus, for each initial state s_L of M_L, we start M_{MC} in the state consisting of s_L and a primed copy of the state s_L over the variables in V_H'.

Our method is based on showing that for each transaction U_i, there is corresponding transition u_i in the specification such that U_i implements u_i. We call u_i the specification transition of U_i, written $\mathrm{spec}(U_i)$. We assume the specification always contains a stuttering transition of the form $true \longrightarrow id$, where for all states s, $id(s) = s$. For a transaction U_i that only writes to variables in $V_L - V_H$, we can assign the specification transition to be the stuttering transition.

We check refinement by finding for each transaction U_i a transition c_i in U_i that acts as a *commit* transition in the sense that whenever c_i fires, the specification transition u_i is enabled on the primed variables. This allows the model M_{MC} to fire the specification transition u_i on the primed variables whenever c_i fires. We will describe our heuristics for finding the commit transitions in Section 4.6.5.

The transitions of M_{MC} are defined as follows. For each transaction U_i, let the commit transition c_i be $cg_i \longrightarrow ca_i$. To build the cross product model, we replace c_i with a transition that executes c_i on the unprimed variables while executing $spec(U_i)$ on the primed variables. Let $spec(U_i)$ be $sg_i \longrightarrow sa_i$. The transition

$$cg_i \longrightarrow \{sg_i'\} sa_i';\ ca_i$$

has the same guard as the commit transition, and uses a precondition annotation to check that the guard of the specification transaction is enabled on the primed variables. When this transition fires, it executes the action sa_i on the primed variables and executes ca_i on the unprimed variables. For each transaction U_i of M_{MC}, we call the precondition

sg_i in the above formula the specification *enableness* condition of U_i. Using the above transition, we can define the transitions of M_{MC} to be

$$\bigcup_i (U_i - \{c_i\} \cup \{cg_i \rightarrow \{sg_i'\} sa_i';\ ca_i\})\,.$$

To check the implementation relation, we define assertions in M_{MC}. We check the interface variables by defining the invariant assertions of M_{MC} to be $A_{MC} = \{v = v' \mid v \in V_I\}$. These assertions check that the implementation always matches the specification on interface variables.

For transactional variables, we check the implementation relation as follows. Consider a transaction U_i and a transactional variable v. If v is written by either U_i or the specification transition $spec(U_i)$, then we assert that $v = v'$ holds at the end of the transaction. We implement these checks by adding a postcondition annotation of the form

$$\{\neg alive_i \Rightarrow vars_i = vars_i'\},$$

to each transition of transaction U_i. In this formula, $alive_i$ is the alive variable of U_i and $vars_i$ is defined to be the vector of all variables written by either a transition of U_i or written by $spec(U_i)$. Attaching this postcondition to each transition t of U_i has the following meaning: if the transaction U_i completes on a given step of the execution of M_{MC}, then $alive_i$ will be false in the following state. The postcondition asserts that $vars_i$ must equal $vars_i'$ in this state. These assertions check the necessary conditions for the transactional variables.

4.3.1 Checking Correct Implementation

Let $S_{mc} = mc_0, mc_1, \ldots,$ be an execution of M_{MC}. If all the specification enableness conditions are satisfied in this execution, one or more specification transitions can fire at each mc_i. It follows that $mc_0 \mid V_H', mc_1 \mid V_H', \ldots,$ is a *concurrent* execution of the specification. For $i \geq 0$, let $h_i = mc_i \mid V_H'$. Also, let S_h be the labelled concurrent execution $h_0 \xrightarrow{E_0} h_1 \xrightarrow{E_1} \ldots,$ where for $i \geq 0$, $E_i = \{u_{i,1}, \ldots, u_{i,n_i}\}$ is the set of specification transitions that fire simultaneously at state h_i in the execution S_h.

We would like to know when the concurrent execution of the specification transitions in S_h can be converted into an interleaving execution. First, there cannot be a write-write conflict in E_i, for any i. Second, for each i, the concurrent firing of the transitions in E_i must be equivalent to some sequential firing order. This is because we define specification models to follow the interleaving semantics of the executions.

For any two transitions t_1, t_2, say t_1 must precede t_2, written $t_1 \prec t_2$, if $R(t_1) \cap W(t_2) \neq \emptyset$. For each set E_i, define $\prec_i^*$ to be the transitive closure of the $\prec$ relation on the transitions in E_i. If the relation $\prec_i^*$ is irreflexive, i.e., no transition in E_i must preceed itself, then there is a sequential firing of the transitions in E_i that updates the state from h_i to h_{i+1}. If the relation $\prec_i^*$ is irreflexive for all i, we say that *serializability* holds for S_h. Finally, if for every such S_h in M_{MC}, serializability holds for S_h, we say that serializability holds for M_{MC}.

The following theorem states the necessary conditions for checking the implementation relation. Assuming that all E_i are write-write conflict free, and S_h is serializable, then there exists a permutation π_i over each set E_i, such that $\pi_i(E_i)$ describes an interleaving execution from h_i to h_{i+1}. More specifically, let $\pi_i(E_i) = w_{i,1}, \ldots, w_{i,n_i}$. Then, it is clear that

$$\begin{array}{l}
h_0 = h_{0,1} \xrightarrow{w_{0,1}} h_{0,2} \xrightarrow{w_{0,2}} \ldots \xrightarrow{w_{0,n_0}} h_{0,n_0+1} = h_1 \\
h_1 = h_{1,1} \xrightarrow{w_{1,1}} h_{1,2} \xrightarrow{w_{1,2}} \ldots \xrightarrow{w_{1,n_1}} h_{1,n_1+1} = h_2 \\
\ldots \\
h_i = h_{i,1} \xrightarrow{w_{i,1}} h_{i,2} \xrightarrow{w_{i,2}} \ldots \xrightarrow{w_{i,n_i}} h_{i,n_i+1} = h_{i+1} \\
\ldots
\end{array}$$

forms an interleaving execution of the specification that goes through the states h_0, h_1,. . .,$h_i, \ldots$.

Theorem 1. Suppose M_{MC} is the monolithic model for implementation M_L and specification M_H. If (i) $(I_L \mid V_H) \subseteq I_H$, (ii) no write-write conflicts exist at any state of M_{MC}, (iii) serializability holds for M_{MC}, (iv) $W(spec(U_i)) \subseteq W(U_i)$ holds for each transaction U_i in M_L, and $(v) \models M_{MC}$, then M_L implements M_H with interface variables V_I and transactional variables V_T.

Proof: We will use induction for the proof. In more detail, for every concurrent execution $S_L = l_0, \ldots, l_m (m \geq 0)$ of M_L, from the definition of M_{MC}, it follows that there exists a concurrent execution of M_{MC}: $mc_0, \ldots, mc_m$, such that $mc_i \mid V_L = l_i, 0 \leq$

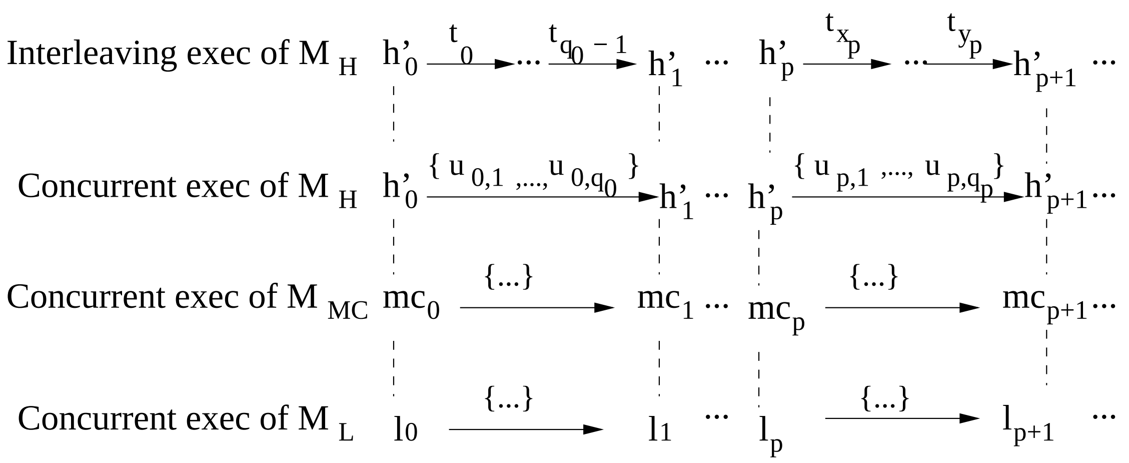


Figure 4.1. The executions of M_{MC}, M_L, and M_H

$i \leq m$. For $i \geq 0$, let $h_i^{'}$ denote $mc_i \mid V_H'$; then $h_0^{'}, \ldots, h_m^{'}$ is a concurrent execution of the specification in M_{MC}. Let $L_H^{'}$ denote the labelled execution of this concurrent execution:

$$h_0^{'} \xrightarrow{E_0} \ldots \xrightarrow{E_{m-1}} h_m^{'}$$

For every $0 \leq i \leq m-1$, let $\{u_{i,1}, \ldots, u_{i,q_i}\}$ be all the transitions in E_i. Because of Parts (i) and (ii) of Theorem 1, there exists an interleaving execution

$$S_H^{'} = h_0^{'} \xrightarrow{t_0} \ldots \xrightarrow{t_{k-1}} h_k^{'} = h_m^{'}$$

where $k = \sum_{i:[0..(m-1)]} q_i$. In addition, $t_0, \ldots, t_{q_0-1}$ is a permutation over $\cup_{j:[1..q_0]} u_{0,j}$. For $0 < i < m$, let $x_i = \sum_{k:[0..(i-1)]} q_k$ and $y_i = \sum_{k:[0..i]} q_k - 1$. Then t_{x_i} is the first transition belonging to E_i in $S_H^{'}$, t_{y_i} is the last transition belonging to E_i, and $t_{x_i}, \ldots, t_{y_i}$ is a permutation over E_i. Also, the n_i's in the definition of "M_L implements M_H" on Page 85 are: $n_0 = 0$ and $n_i = x_i$ for $i > 0$. Figure 4.1 shows these executions.

We now apply induction on m to prove that M_L implements M_H. Because for every $v \in V_I$, $v = v' \in A_{MC}$ and $\models M_{MC}$ holds, $v = v'$ holds for every state of M_{MC}. So we will only consider V_T in the following.

- Base case, $m = 0$: Consider $l_0 \in I_L$. According to the definition of I_{MC}, there exists an initial state mc_0 of M_{MC} such that $l_0 = mc_0 \mid V_L$. For l_0, all *alive* variables are false; so all the variables in V_L are inactive. Therefore we have $l_0 \mid (V_H \cap inactive(l_0)) = l_0 \mid V_H$. Let lh_0 denote $l_0 \mid V_H$, and recall that $h_0^{'} = mc_0 \mid V_H'$. The definition of I_{MC} ensures that $lh_0 = h_0^{'}$. Hence the conclusion holds for $m = 0$.

- Induction Hypothesis: Suppose the conclusion holds for $0 \leq m \leq p$, $p \geq 0$.
- Induction Step, $m = p+1$: Let $\mathrm{iars_i} = V_H \cap \mathrm{inactive}(l_i)$, $0 \leq i \leq p+1$. For every variable $v \in \mathrm{iars}_{p+1}$, v can come from three sources as listed in the following.

 1. $v \in \mathrm{iars}_p$, and v is not in the write set of any transition which is fired between l_p and l_{p+1}. In this case, $l_p(v) = l_{p+1}(v)$. By induction hypothesis, $l_p(v) = h_p^{'}(v^{'})$. Also, since $W(\mathrm{spec}(U_i)) \subseteq W(U_i)$ holds for every transaction U_i, the value of $v^{'}$ stays unmodified from $h_p^{'}$ to $h_{p+1}^{'}$, i.e. $h_p^{'}(v^{'}) = h_{p+1}^{'}(v^{'})$. So $l_{p+1}(v) = h_{p+1}^{'}(v^{'})$ holds.
 2. $v \in \mathrm{iars}_p$, but v is in the write set of some transaction U_i which starts at l_p and finishes at l_{p+1}. In other words, v is both inactive at l_p and l_{p+1}, but its value may have changed because of U_i. In this case, the annotation $(\mathrm{vars}_i = \mathrm{vars}_i')$ is checked and proved in $\models M_{MC}$. Because of $v \in \mathrm{vars}_i$, it follows that $l_{p+1}(v) = h_{p+1}^{'}(v^{'})$ holds.
 3. $v \notin \mathrm{iars}_p$. In this case, there must exist a transaction U_i such that $v \in W(U_i) \cap \mathrm{vars_i}$, $l_p(alive_i) = true$ and $l_{p+1}(alive_i) = \mathrm{false}$. Again, the annotation $(\mathrm{vars}_i = \mathrm{vars}_i')$ holds in $\models M_{MC}$ ensures that $l_{p+1}(v) = h_{p+1}^{'}(v^{'})$ holds.

In the above three cases, absence of write-write conflicts and serializability holding ensure that there is a permutation of E_p that establishes the simulation. $\square$

4.4 Compositional Model Checking

In this section, we develop techniques that allow one to reason efficiently about a model by checking properties of a set of smaller, more abstract models. There are two basic ideas: abstraction and assume guarantee reasoning. Abstraction is a *conservative* approach to decomposing the original monolithic model. That is, we construct a set of abstract models from the monolithic model, and by verifying the abstract models, the original model is guaranteed to be correct with respect to its properties. Assume guarantee reasoning can refine (as in the classical counterexample guided refinement approach [44]) the abstract models constructed by abstraction. Assume guarantee reasoning used together with abstraction can help reduce the verification complexity of the abstract models. In the following, we will first present our abstraction and then in Section 4.4.2 describe assume guarantee reasoning used in our approach. We will describe a very simple example in Section 4.4.1 and 4.4.2 illustrating the basic idea of our abstraction and assume guarantee reasoning.

Consider the problem of making an abstract model that conserves correctness of the annotations on a transition t in a model M. We would like to remove as many transitions from the model as possible, while maintaining the property that if the annotations hold in the abstract model, they hold in the original model.

To form an abstraction, we analyze the sources that supply values for variables that are read by a transition. To make a model more abstract, we can convert some sources into free input variables and remove certain transitions from the model. We will develop a condition that ensures that input sources for a transition in an abstract model are at least as general as the sources in the original model. One distinctive feature of our approach is that we allow abstractions in which a variable is abstracted to an input variable in some transitions, but the original variable is retained in other transitions. This approach allows us to control the level of detail contained in an abstract model. We use this control of abstraction to reason compositionally about transactions.

To form abstract models, we introduce *input variable*s, and models with input variables. A model with input variables has the form $M = (V, \tilde{V}, I, T, A)$, where V is the set of state variables and $\tilde{V}$ is the set of input variables. For each variable v in V, there are two corresponding input variables, $\tilde{v}$ and $\tilde{v}^n$. The variable $\tilde{v}$ is used to provide an unconstrained input on the current state. The variable $\tilde{v}^n$ is used in annotated transitions, to provide an unconstrained input for postcondition annotations, which are evaluated in the "next" state of the model.

Semantically, a model with input variables is similar to a model without inputs. The states of the model map $V \cup \tilde{V}$ to values. Transitions can read but not write the input variables. The action function of a transition defines the next state of the ordinary state variables in V only. The next-state relation for input variables is unconstrained: an input variable can take on any value in the next state of a computation.

Without loss of generality, we can assume that all models have input variables, since a model that does not mention input variables can be represented as a model with input variables.

4.4.1 Abstraction

Let $t = (g \longrightarrow a)$ be a transition of a model M. We define a *read variant* of t as the result of replacing reads of zero or more variables in $R(t)$ with their corresponding input variables. More formally, let $V_t = \{v_1, \ldots, v_n\} \subseteq V$, $n \geq 0$. We define a substitution function σ, to replace each read of $v \in V_t$ in t with the input variable $\tilde{v}$:

$$\sigma(t, V_t) = (\sigma(g, V_t) \longrightarrow \sigma(a, V_t))$$

where

$$\sigma(g, V_t) = \lambda s.g(s[v_1 \leftarrow s(\tilde{v}_1)] \ldots [v_n \leftarrow s(\tilde{v}_n)])$$

$$\sigma(a, V_t) = \lambda s.a(s[v_1 \leftarrow s(\tilde{v}_1)] \ldots [v_n \leftarrow s(\tilde{v}_n)])$$

We write $\tilde{t}$ for read variant of t. A transition t is a read variant of itself in the case that no variables are renamed. If T is a set of transitions, then we define $variants(T)$ to be the set of all read variants of transitions in T.

We now discuss our approach to abstraction in more detail. Let $M = (V, \tilde{V}, I, T, A)$ be a model. We are interested in models built from a subset of $variants(T)$ that contain no more than one read variant of any transition in T. For the restriction of *no more than one*, we mean that because for any transition t in T, $variants(T)$ can contain multiple read variants of t. Therefore, for each abstract model built from M, we want it to contain at most one invariant of t. Let us call such a set *consistent*. Intuitively, a model defined by a consistent set of read variants of T can execute the behavior of M on a subset of the variables, and we want that the model will not contain two transitions that will generate write-write conflicts at any reachable state.

Now we define a syntactic condition that says when a set of read variants can exhibit all behaviors of the original model over a subset of the state variables. The syntactic condition defined in the following is just one form of such conditions, and there can be many other forms as well. Let $\tilde{T}$ be a set of read variants of transitions in T, $\tilde{T} = \{\tilde{t}, t \in T\}$. Consider $t \in T$, $\tilde{t}$ which is a read variant of t, and a variable v that is read by $\tilde{t}$, $v \in R(\tilde{t})$. We say that the read of v in $\tilde{t}$ is ***sufficiently general*** with respect to M if one of the following holds:

1. v is an input variable, or
2. $\tilde{T}$ contains a read variant of every transition in T that writes to v, or
3. There is a transition $s \in T$ that writes v, $\tilde{s} \in \tilde{T}$, and in every concurrent execution of M, whenever t fires, s fires on the previous step.

We say that $\tilde{T}$ is sufficiently general with respect to M if every read of every variable in $\tilde{T}$ is sufficiently general with respect to M. In the above, we consider three options under which sufficiently general can be said to hold; there could be other options as well.

Our abstract models overapproximate an original model by reading the values of a combination of state and input variables. In order to compare executions of abstract models to the original models, we define the *effective input state* of a model with input variables at a given state. When a transition reads the value of an input variable $\tilde{v}$, the effective input state assigns the same value to the original variable v. Let $M = (V, \tilde{V}, I, T, A)$ be a model and s be a state of M. The effective input state of a transition t at a state s, $\mathit{eff}(t, s)$, is defined for $v \in R(t)$:

1. If t reads the state variable v at s, then $\mathit{eff}(t, s)(v) = s(v)$.
2. If t reads the input variable $\tilde{v}$ at s, then $\mathit{eff}(t, s)(v) = s(\tilde{v})$.

The effective read states of a transition t in a model M are defined by $\mathit{Eff}(t, M) =$

$$\{\mathit{eff}(t, s) \mid s \text{ is a reachable state of } M \text{ and } t \text{ is enabled at } s\}.$$

The following proposition describes how a model M' defined by a sufficiently general subset of read variants of transitions of a model M can overapproximate the behavior of M.

Proposition 2. Let $M = (V, \tilde{V}, I, T, A)$ and let T' be a subset of T. Let $M' = (V, \tilde{V}, I', \tilde{T}, A')$ be a model where $I \subseteq I'$, and $\tilde{T}$ contains a read variant of each transition in T', i.e., M' contains a consistent set of T. If $\tilde{T}$ is sufficiently general with respect to M, then for all transitions $t \in T'$ $\mathit{Eff}(t, M) \subseteq \mathit{Eff}(\tilde{t}, M')$.

Remark: From our definition of effective read states $\mathit{Eff}(t, M)$, it is clear that $\mathit{Eff}(t, M)$ considers all the state variables of M that are read by t in every reachable state of M. Now consider the case where $t \in T$ is an annotated transition with precondition P and $\tilde{t} \in \tilde{T}$. If $\mathit{Eff}(t, M) \subseteq \mathit{Eff}(\tilde{t}, M')$ holds, then the read variant of P holds in M' implies that P must hold in M. This means that if $\tilde{T}$ is sufficiently general with respect to M, then it is sufficient to simply verify the read variant of P in M' instead of verifying P in M, i.e., the abstraction that is used to construct M' is conservative.

Proof: Let $S_M = s_0 \xrightarrow{E_0} s_1 \xrightarrow{E_1} \ldots$ be a concurrent execution of M where $s_0 \in I$. Based on S_M, we will find out a concurrent execution for M'. In more detail, let $n_0, n_1, \ldots,$ be a set of numbers where n_0 is the smallest number j such that there exists t in E_j and a read variant of t is in $\tilde{T}$. Similarly for every $i > 0$, let n_i be the smallest number j such that $j > n_{i-1}$, and there exists t in E_j and $\tilde{t}$ is in $\tilde{T}$. For every E_{n_i} of S_M, $i \geq 0$, we define E'_i to be the set of all transitions $\tilde{t}$ of M' such that t is in E_{n_i}.

Let $s'_0 = s_0$, and for $i > 0$, we define s'_i to be the state such that $s'_{i-1} \xrightarrow{E'_{i-1}} s'_i$. Now we will prove that $S'_M = s'_0 \xrightarrow{E'_0} s'_1 \xrightarrow{E'_1} \ldots$ is a concurrent execution of M'. Moreover, for every transition t in E_{n_i}, $i \geq 0$, such that $\tilde{t}$ is in E'_i, $\mathit{eff}(t, s_{n_i})(v) = \mathit{eff}(\tilde{t}, s'_i)(v)$ holds for every variable v that is read by t. We will apply induction on n_i of S_M to prove the proposition.

- Base Case, $i = 0$: We consider every transition t in E_{n_0} where a read variant of t is in M', and every variable v that is read by t. Case analysis:

1. If the value of v that t reads at s_{n_0} is from s_0, then whether $\tilde{t}$ reads v or the input variable $\tilde{v}$, $\mathit{eff}(t, s_{n_0})(v) = \mathit{eff}(\tilde{t}, s'_0)(v)$ can hold. For the case when $\tilde{t}$ reads $\tilde{v}$, the formula holds when $\tilde{v}$ takes on the value $s_0(v)$.

2. If the value of v that t reads at s_{n_0} is from an update of a transition x in E_j, $0 \leq j < n_0$), according to the definition of n_0, no read variant of x is in $\tilde{T}$. From the definition of "sufficiently general," $\tilde{t}$ must read the input variable $\tilde{v}$. Let $\tilde{v}$ take on the value $s_{n_0}(v)$. Then $s_{n_0}(v) = s_{j+1}(v)$, and $\mathit{eff}(t, s_{n_0})(v) = \mathit{eff}(\tilde{t}, s'_0)(v)$ holds.

3. If t reads the input variable $\tilde{v}$ at s_{n_0}, let $\tilde{t}$ take on the same value as t takes at s_{n_0}. Then $\mathit{eff}(t, s_{n_0})(v) = \mathit{eff}(\tilde{t}, s'_0)(v)$ must hold.

- Induction Hypothesis: Suppose for $0 \leq i \leq p$, $p \geq 0$, $s'_0 \xrightarrow{E'_0} \ldots s'_p \xrightarrow{E'_p} s'_{p+1}$ is a concurrent execution of M'. Also, for every transition t in E_{n_i} and $\tilde{t}$ is in E'_i, $\mathit{eff}(t, s_{n_i})(v) = \mathit{eff}(\tilde{t}, s'_i)(v)$ holds for every variable v that is read by t.

- Induction Step, $i = p + 1$: We consider every transition t in $E_{n_{p+1}}$ where a read variant of t is in $\tilde{T}$, and every variable v that is read by t. Case analysis:

1. If the value of v that t reads at $s_{n_{p+1}}$ is from s_0, then v is not in the write set of any transition in $E_0, \ldots, E_{n_{p+1}-1}$. Thus, $s'_{p+1}(v) = s_0(v)$. Whether $\tilde{t}$ reads v or the input variable $\tilde{v}$ at s'_{p+1}, $\mathit{eff}(t, s_{n_{p+1}})(v) = \mathit{eff}(\tilde{t}, s'_{p+1})(v)$ can hold. When $\tilde{t}$ reads $\tilde{v}$, the formula holds when $\tilde{v}$ takes on the value $s_0(v)$.

2. If the value of v that t reads at $s_{n_{p+1}}$ is from an update of a transition x in E_{n_j}, $0 \leq j \leq p$, and $\tilde{x}$ is in E'_j, then $s_{n_j+1}(v) = s_{n_{p+1}}(v)$. By induction, it follows that for every variable u that is read by x, $\mathit{eff}(x, s_{n_j-1})(u) = \mathit{eff}(\tilde{x}, s'_{j-1})(u)$ holds. So $s_{n_j+1}(v) = s'_{j+1}(v)$ must hold. Whether $\tilde{t}$ reads v or $\tilde{v}$, $\mathit{eff}(t, s_{n_{p+1}})(v) = \mathit{eff}(\tilde{t}, s'_{p+1})(v)$ can hold. For the case when $\tilde{t}$ reads $\tilde{v}$, the formula holds when $\tilde{v}$ takes on the value $s_{n_j+1}(v)$.

3. If the value of v that t reads at $s_{n_{p+1}}$ is from an update of a transition y in E_k, $0 \leq k < n_{p+1}$, and no read variant of y is in $\tilde{T}$, then according to the definition of "sufficiently general," $\tilde{t}$ must read $\tilde{v}$. Also, we have

$s_{k+1}(v) = s_{n_{p+1}}(v)$. Let $\tilde{v}$ take on the value $s_{k+1}(v)$. Then $\mathit{eff}(t, s_{n_{p+1}})(v) = \mathit{eff}(\tilde{t}, s'_{p+1})(v)$ holds.

4. If t reads $\tilde{v}$ at $s_{n_{p+1}}$, let $\tilde{t}$ take on the same value of $\tilde{v}$ as t takes. Then $\mathit{eff}(t, s_{n_{p+1}})(v) = \mathit{eff}(\tilde{t}, s'_{p+1})(v)$ must hold.

In all the four cases, $\tilde{t}$ can be enabled at s'_{p+1} and $\mathit{eff}(t, s_{n_{p+1}})(v) = \mathit{eff}(\tilde{t}, s'_{p+1})(v)$ holds for every variable v that t reads. So the induction holds upto $i = p + 1$.

The above proof indicates that for every concurrent execution $S_M = s_0 \xrightarrow{E_0} s_1 \xrightarrow{E_1} \ldots$ of M, there exists a concurrent execution $S'_M = s'_0 \xrightarrow{E'_0} s'_1 \xrightarrow{E'_1} \ldots$ of M', such that for every transition t in E_j, $j \geq 0$, and a read variant of t is in M' (here j is an n_i in our proof),

$$\mathit{eff}(t, s_j)(v) = \mathit{eff}(\tilde{t}, s'_i)(v)$$

holds for every variable v that is read by t. Therefore $\mathit{Eff}(t, M) \subseteq \mathit{Eff}(\tilde{t}, M')$ holds. □

The notions of read variant and sufficiently general abstraction extend readily to models with annotations. A read variant of an annotated transition $g \rightarrow \{P\}a\{Q\}$ is formed by consistently replacing read variables v with the corresponding same-state input variables $\tilde{v}$ in g, P and a, and replacing variables v in Q with the corresponding next-state variables $\tilde{v}^n$.

Consider an annotated model M and a set $\tilde{T}$ of read variants of transitions in M. A variable instance v in the precondition of an annotated transition is sufficiently general with respect to M under the same conditions that apply for a read instance in a guard or action. A variable instance v in a postcondition of a transition $\tilde{t}$ is sufficiently general if one of the following holds:

1. v is a next-state input variable.

2. $\tilde{T}$ contains a read variant of every transition of M that writes to v.

3. $\tilde{t}$ writes to v.

4. There is a transition s of M that writes to v, $\tilde{s} \in \tilde{T}$, and in every concurrent execution of M, if t fires then s fires on the same step.

As before, we say that $\tilde{T}$ is sufficiently general with respect to M if every read of every variable in $\tilde{T}$ is sufficiently general with respect to M.

Proposition 3. Let $M = (V, \tilde{V}, I, T, A)$ and $M' = (V, \tilde{V}, I', \tilde{T}, A')$ be models where $I \subseteq I'$, and $\tilde{T}$ is a consistent set of T that is sufficiently general with respect to M. Let t be an annotated transition in T and $\tilde{t}$ be a read variant in $\tilde{T}$. Then $M' \models \tilde{t} \text{ implies } M \models t$.

Remark: Consider any annotated transition t in T with precondition P and postcondition Q. This proposition suggests that if $\tilde{T}$ in the abstract model M' is sufficiently general with respect to M, then it is sufficient for us to simply verify the read variants of P and Q in M', instead of verifying them in M. Again, this means that the abstraction that is used to construct M' is conservative.

Proof: Let $t = g \longrightarrow \{P\}a\{Q\}$ be an annotated transition in T where a read variant of t is in T'. For the pre-condition P, in order for $M \models t$ to hold, for any concurrent execution $s_0 \xrightarrow{E_0} s_1 \xrightarrow{E_1} \ldots$ of M, whenever $s_i \xrightarrow{E_i} s_{i+1} (i \geq 0)$ and E_i contains t, $P(s_i)$ must hold. From Proposition 2, we have $\mathit{Eff}(t, M) \subseteq \mathit{Eff}(\tilde{t}, M')$. So $M' \models \tilde{t}$ implies $M \models (g \longrightarrow \{P\}a)$. In the following, we will only consider the postcondition Q.

For the postcondition Q of t, we will prove the proposition by contradiction. Suppose $M' \models t'$ holds but $M \models t$ does not. Then there exists a concurrent execution of M, $s_0 \xrightarrow{E_0} s_1 \xrightarrow{E_1} \ldots$, such that there exists a number m, $m \geq 0$, where E_m contains t but Q fails. Here, we consider the smallest m which satisfies the above condition. Now we will find out a concurrent execution of M' in the same way as in the proof of Proposition 2.

Let $n_0, n_1, \ldots,$ be a set of numbers where n_0 is the smallest number j such that there exists a transition x in E_j and a read variant of x is in M'. Similarly for every $i > 0$, let n_i be the smallest number j such that $j > n_{i-1}$, and there exists x in E_j and $\tilde{x}$ is in M'. For every E_{n_i}, $i \geq 0$, we define E'_i to be the set of all transitions $\tilde{x}$ of M' such that x is in E_{n_i}.

Let $s'_0 = s_0$, and for $i > 0$, we define s'_i to be the state such that $s'_{i-1} \xrightarrow{E'_{i-1}} s'_i$. From the proof of Proposition 2, there exists a concurrent execution of M': $s'_0 \xrightarrow{E'_0} s'_1 \xrightarrow{E'_1} \ldots$, and for every transition x in E_{n_i} and $\tilde{x}$ is in M', $i \geq 0$, ,

$$\mathit{eff}(x, s_{n_i})(v) = \mathit{eff}(\tilde{x}, s'_i)(v)$$

holds for every variable v that is read by x.

Now we consider t with the postcondition Q. Because $\tilde{t}$ is in M', the number m where E_m contains t but Q fails must be some n_i, $i \geq 0$. Let m be n_p, $p \geq 0$. For every variable v which is in the postcondition Q, we do a case-analysis based on the definition of "sufficiently general" for reads of variables in postconditions. Our goal is to prove that for every such v, the value it is evaluated for Q of t in M, is the same for Q or its read invariant of $\tilde{t}$ in M'.

1. v is a next-state input variable. Let $\tilde{t}$ take on the value $s_{n_p+1}(v)$ for $\tilde{v}^n$.

2. T' contains a read variant of every transition of M that writes to v. From the induction of the proof of Proposition 2, it follows that $s'_{p+1}(v) = s_{n_p+1}(v)$ holds.

3. $\tilde{t}$ writes to v. Because $\mathit{eff}(t, s_{n_p})(u) = \mathit{eff}(\tilde{t}, s'_p)(u)$ holds for every variable u that is read by t, $s'_{p+1}(v) = s_{n_p+1}(v)$ holds.

4. There is a transition y of M that writes to v, $\tilde{y} \in T'$, and in every concurrent execution of M, if t fires then y fires on the same step. Similarly, because $\mathit{eff}(y, s_{n_p})(u) = \mathit{eff}(\tilde{y}, s'_p)(u)$ holds for every variable u that is read by y, $s'_{p+1}(v) = s_{n_p+1}(v)$ holds.

The above analysis indicates that for this execution of M', if Q or its read variant of $\tilde{t}$ holds at s'_p, Q of t must also hold at s_m. From $M' \models \tilde{t}$, it follows that Q of t holds at s_m. Contradiction. Therefore, $M \models t$ must hold. $\Box$

4.4.1.1 A Simple Example

Now we show a trivial example just to illustrate the basic idea of abstraction. Consider a model $M = (V, I, T, A)$ where V includes four variables in which v_1, v_2 are integers and v_3, v_4 are booleans. The initial state I is $v_1 = v_2 = 0$, $v_3 = \mathrm{true}$ and $v_4 = \mathrm{false}$. The set of assertions A includes two formulas: $v_3 = \mathrm{true}, v_4 = \mathrm{false}$.

Finally, the transition T contains two transactions U_1, U_2 as follows in which we use $:=$ for assignment:

$$U_1 = \text{transaction}[(v_1 \geq 0 \rightarrow v_3 := \text{true}); (\text{true} \rightarrow v_2 := v_2 - 1)]$$

$$U_2 = \text{transaction}[(v_2 \leq 0 \rightarrow v_4 := \text{false}); (\text{true} \rightarrow v_1 := v_1 + 1)]$$

With our abstraction, we can construct two models M_1, M_2 from M which overapproximates M. $M_1 = (V_1, \tilde{V_1}, I_1, T_1, A_1)$ where $V_1 = V$, I_1 is simply $v3 = \text{true}, v_2 = 0$, $A_1 = \{v_3 = \text{true}\}$, and T_1 only includes one transaction as follows:

$$U_1' = \text{transaction}[(\tilde{v_1} \geq 0 \rightarrow v_3 := \text{true}); (\text{true} \rightarrow v_2 := \tilde{v_2} - 1)]$$

Similarly, $M_2 = (V_2, \tilde{V_2}, I_2, T_2, A_2)$ where $V_2 = V$, I_2 is simply $v4 = \text{false}, v_1 = 0$, $A_2 = \{v_4 = \text{false}\}$, and T_2 only includes one transaction as follows:

$$U_2' = \text{transaction}[(\tilde{v_2} \leq 0 \rightarrow v_4 := \text{false}); (\text{true} \rightarrow v_1 := \tilde{v_1} - 1)]$$

In the above, we replace all the read of variables v_1, v_2 to the read of the input variables $\tilde{v_1}, \tilde{v_2}$ in the models M_1, M_2. From our abstraction, it is clear that $\models M_1 \wedge \models M_2$ holds implies that $\models M$ holds. However, for this example $\models M_1 \wedge \models M_2$ does not hold because the abstraction is too conservative, i.e., the input variables $\tilde{v_1}, \tilde{v_2}$ are completely unconstrained. In the following section, we will show how assume guarantee reasoning can help refine the models M_1, M_2 such that $\models M$ can hold.

4.4.2 Assume Guarantee Reasoning

In the previous section, we develop a set of sufficient conditions under which the annotations of a transition can be checked in an abstract model. That is, we propose a list of guidelines for conservative abstraction; of course there can be other guidelines as well. Abstract models constructed from such conservative abstraction usually overapproximates the concrete model, and assume guarantee reasoning can help refine the abstract models. More specifically, assume guarantee reasoning involving the input variables of an abstract model can reduce the size of the models that need to be checked below the size that is needed when abstraction is used alone.

Now we describe assume guarantee reasoning used in our approach in a formal way. Because assume guarantee reasoning is a form of induction in which we introduce a set of assumptions and at the same time justify them, we will first define the notions used to represent it. Let $P(\tilde{V})$ be a satisfiable formula over the input variables in $\tilde{V}$ and $Q(V)$ is an invariance property over the state variables V. We define $M; P(\tilde{V}) \models Q(V)$ to be true if $Q(V)$ holds on all reachable states of the model M, provided the input variables are constrained at each step to satisfy $P(\tilde{V})$.

In its simplest form, the assume guarantee reasoning that is used in our approach is a form of induction involving the input variables. Suppose $M = (V, \emptyset, I, T, A)$ is a model, and $M' = (V, \tilde{V}, I', \tilde{T}, A')$ is a model with input variables, where $\tilde{T}$ contains a read variant of every transition in T, and $I \subseteq I'$. Then the following scheme is valid. Note that any update of the variables V from the input variables $\tilde{V}$ requires at least one time step for a transition to fire.

$$\frac{M'; P(\tilde{V}) \models P(V)}{M \models P(V)}$$

Remark: The above scheme states that if the formula $P(V)$ holds on all the reachable states of the model M' provided that the input variables of M' are constrained at each step of M' to satisfy $P(\tilde{V})$, then $P(V)$ must hold in the model M without any constraint.

Proof: Let $S_M = s_0 \xrightarrow{E_0} s_1 \xrightarrow{E_1} \ldots$ be any concurrent execution of M. For $\forall i \geq 0$, let E'_i be the set of transitions in M' which contains the read variant of every transition in E_i. Let $s'_0 = s_0$, and for $i > 0$, we define s'_i to be the state such that $s'_{i-1} \xrightarrow{E'_{i-1}} s'_i$. We will prove by induction that $s_i = s'_i$ can hold for every $i \geq 0$.

- Base Case, $i = 0$: The fact that $M'; P(\tilde{V}) \models P(V)$ holds implies $I' \models P(V)$. Because $I \subseteq I'$, it follows that $I \models P(V)$. So $P(V)$ holds at s_0.

- Induction Hypothesis: Suppose for $s_0 \xrightarrow{E_0} s_1 \ldots \xrightarrow{E_{p-1}} s_p$ of S_M, $p \geq 0$, there exists a concurrent execution $s'_0 \xrightarrow{E'_0} s'_1 \ldots \xrightarrow{E'_{p-1}} s'_p$ of M', such that $s_i = s'_i$, $0 \leq i \leq p$, and E'_i includes the read variant of every transition of E_i. Also, $P(V)$ holds at s_i, $0 \leq i \leq p$ of S_M; thus $P(V)$ holds for every s'_i, $0 \leq i \leq p$. For every E'_i, $0 \leq i < p$, if some transitions in E'_i read input variables, then the values they take on satisfy $P(\tilde{V})$.

- Induction Step, $i = p+1$: For every transition t in E_p, we consider every variable v that is read by t. If the read variant $\tilde{t}$ of M' reads v, then by induction $s'_p(v) = s_p(v)$ holds. Otherwise, $\tilde{t}$ reads input variables in $\tilde{V}$. Because $P(\tilde{V})$ is satisfiable, and by induction $P(V)$ holds at s_p, for every v where $\tilde{t}$ reads $\tilde{v}$ at s'_p, let $\tilde{v}$ take on the value $s_p(v)$. By doing so, $s_{p+1} = s'_{p+1}$ holds, and $P(\tilde{V})$ is satisfied because $P(V)$ holds at s_p. From $M'; P(\tilde{V}) \models P(V)$, it follows that $P(V)$ holds at s'_{p+1}, so does s_{p+1}. The induction holds upto $p+1$.

Therefore, $M \models P(V)$ holds. $\square$

The actual form of assume guarantee reasoning that we use can be stated as follows. Suppose $M = (V, \emptyset, I, T, \emptyset)$ is a model, and $M_1 = (V, \tilde{V}, I_1, T_1, \emptyset)$, and $M_2 = (V, \tilde{V}, I_2, T_2, \emptyset)$ are models with input variables where T_1 and T_2 are consistent sets of T that are both sufficiently general with respect to M. We require that for each transition t in M, either M_1 or M_2 contains a read variant of t. Also suppose the initial states of M_1, M_2 both contain the initial states of M. These conditions assure that M_1 and M_2 are abstractions of M that preserve the correctness of annotations on transitions in M.

We can use compositional assume guarantee reasoning as follows. Let $P(X)$ be a formula over a set of state variables $X \subseteq V$. Suppose that every transition in M that writes to a variable in X is annotated with the postcondition $P(X)$. Then we can use the following scheme, which checks that $P(X)$ or a read variant is true in the initial state and whenever one of the variables in X is written. The scheme assumes $P(\tilde{X})$ is always true over the input variables, where $\tilde{X}$ is the result of replacing each variable in X with the corresponding input variable.

$$\frac{\begin{array}{cc} I_1 \models P(X) & M_1; P(\tilde{X}) \models T_1 \\ I_2 \models P(X) & M_2; P(\tilde{X}) \models T_2 \end{array}}{M \models T}$$

Remark: The above scheme states that under the conditions that (i) the initial states I_1, I_2 satisfy the formula $P(X)$, and (ii) for every transition t in M_1, M_2 which writes to a variable in X, the read variant of $P(X)$ as a postcondition of t holds provided that the input variables of M_1, M_2 are constrained to satisfy $P(\tilde{X})$ at every step of M_1, M_2, imply that – in the model M, the postcondition $P(X)$ of every transition that writes to a variable in X must hold without any constraint.

Proof: This proof is a composition of those of the previous three propositions. Let $S_M = s_0 \xrightarrow{E_0} s_1 \xrightarrow{E_1} \ldots$ be any concurrent execution of M, $s_0 \in I$. For M_1, let $n_0, n_1, \ldots$ be a set of numbers which satisfy the following properties: (i) n_0 is the smallest number j such that there exists a transition $t \in E_j$ and a read variant of t is in T_1; (ii) for each $i > 0$, n_i is the smallest number j such that $j > n_{i-1}$, and there exists $t \in T_j$ and $\tilde{t} \in T_1$. For every E_{n_i} of S_M, $i \geq 0$, we define $E_{1,i}$ to be the set of all transitions $\tilde{t}$ of T_1 where $t \in E_{n_i}$. Let $s_{1,0} = s_0$, and for $i > 0$, we define $s_{1,i}$ to be the state of M_1 such that $s_{1,i-1} \xrightarrow{E_{1,i-1}} s_{1,i}$. Similarly, for M_2 we define the numbers $m_0, m_1, \ldots$, and for every m_i, $i \geq 0$, we define $E_{2,i}$ and the states $s_{2,i}$.

Now we prove the proposition by induction on $i \geq 0$, for every s_i and E_i.

- Base Case, $i = 0$: For every $t = g \rightarrow \{P\}a\{Q\} \in E_0$, from the proposition "for each transition in M, either M_1 or M_2 contains a read variant", it follows that $\tilde{t}$ is either in M_1 or M_2. Suppose $\tilde{t} \in T_1$. Now we consider g, P, a and Q and their corresponding read variants.

  For every variable v that is read by g, P or a, if the input variable $\tilde{v}$ is read by any of the read variants $\tilde{g}, \tilde{P}$ or $\tilde{a}$, let $\tilde{v}$ take on the value $s_0(v)$. Because $I_1 \models P(X)$ and $I \subseteq I_1$, we have $I \models P(X)$. Thus, s_0 satisfy $P(X)$. So if there is any input variable $\tilde{v}$ read by $\tilde{g}, \tilde{P}$ or $\tilde{a}$, the values that the input variables take on can satisfy the constraints $P(\tilde{X})$.

  Because $s_{1,0} = s_0$, it is clear that $\mathit{eff}(t, s_0)(v) = \mathit{eff}(\tilde{t}, s_{1,0})(v)$ holds for every variable that is read by g, P or a. As a result, for each variable u that is in the write set of t, it follows that $s_1(u) = s_{1,1}(u)$ must hold.

  Now we consider the postcondition Q of t in E_0, based on the fact that M_1 is sufficiently general with respect to M. For every variable v in Q, we do a case analysis based on the definition of "sufficiently general" for v in $\tilde{Q}$ of $\tilde{t}$ in $E_{1,0}$.

  1. $\tilde{Q}$ reads $\tilde{v}^n$, i.e., the next-state input variable: Let $\tilde{v}^n$ take on the value $s_1(v)$.

2. T_1 contains a read variant of every transition of M that writes to v, or $\tilde{t}$ writes to v, or there is a transition $s \in T$ that writes to v, $\tilde{s} \in T_1$, and s fires in E_0 also: In all these cases, we have $s_1(v) = s_{1,1}(v)$ from the above analysis that for each variable u that is in the write set of t, $s_1(u) = s_{1,1}(u)$ holds.

So it is clear that the postcondition $\tilde{Q}$ of $\tilde{t}$ in $E_{1,0}$ must be evaluated to the same value as Q of t in E_0. The above holds similarly if $\tilde{t} \in T_2$.

From the fact that $M_1; P(\tilde{X}) \models T_1$ and $M_2; P(\tilde{X}) \models T_2$, it follows that $\tilde{Q}$ must hold, so does Q in E_0. Finally, if $\exists x \in X$ in the write set of t, from the proposition, we know that t is annotated with the postcondition $P(X)$. From the analysis for any postcondition Q, it follows that $P(X)$ must hold at s_1 of S_M.

- Induction Hypothesis: Suppose for $s_0 \xrightarrow{E_0} s_1 \ldots \xrightarrow{E_{p-1}} s_p$ of S_M, $p \geq 0$, there exists a concurrent execution

$$s_{1,0} \xrightarrow{E_{1,0}} \ldots \xrightarrow{E_{1,p_1-1}} s_{1,p_1}$$

$$s_{2,0} \xrightarrow{E_{2,0}} \ldots \xrightarrow{E_{2,p_2-1}} s_{2,p_2}$$

of M_1, M_2 respectively, which satisfies the following properties. First, p_1 is the largest number of n_i in its definition such that $n_i \leq p$. Second, $P(X)$ holds for every s_i, $0 \leq i \leq p$. For every $0 \leq i < p_1$, the values that the input variables (if any) take on, for every transition in $E_{1,i}$, satisfy the constraint $P(\tilde{X})$. Third, for every $t \in E_i$, $0 \leq i < p$, if $\tilde{t}$ is in T_1 of M_1, then $\mathit{eff}(t, s_i)(v) = \mathit{eff}(\tilde{t}, s_{1,j})(v)$ holds for every variable v that is read by t. Here $n_j = i$, $j \geq 0$. The above holds similarly if $\tilde{t}$ is in T_2 of M_2.

- Induction Step, i = p+1: For every transition $t = g \rightarrow \{P\}a\{Q\}$ in E_p of S_M, we consider its read variant in M_1 or M_2. Suppose $\tilde{t} \in T_1$. Now we consider g, P, q and Q and their read variants.

  For every variable v that is read by g, P or a, we do a case analysis based on the definition of "sufficiently general" for read of variables.

1. v is an input variable of $\tilde{g}, \tilde{P}$ or $\tilde{a}$: Let $\tilde{v}$ take on the value $s_p(v)$. Because $P(X)$ holds at s_p, the values that the input variables take on satisfy can the constraints $P(\tilde{X})$.

2. T_1 contains a read variant of every transition in M that writes to v: There will be two cases: (i) $s_p(v) = s_0(v)$, i.e., no transition has ever updated v yet, or (ii) let y be the transition that *last* writes to v, i.e., $y \in E_j$, $j < p$, and $\tilde{y} \in E_{1,j_1}, n_{j_1} = j$.

   For the first case, it must be that $s_{1,p_1}(v) = s_0(v)$. For the second case, by induction, it follows that $\mathit{eff}(y, s_j)(u) = \mathit{eff}(\tilde{y}, s_{1,j_1})(u)$ holds for every variable that is read by y. So $s_p(v) = s_{1,p_1}(v)$ must hold.

3. $\exists s \in T$ that writs to v, $\tilde{s} \in T_1$, and s fires on the previous step of t, i.e., $s \in E_{p-1}$: Similar to the previous case, we can prove that $s_p(v) = s_{1,p_1}(v)$ must hold.

From the above analysis, it follows that for every variable v in the write set of t, $s_{p+1}(v) = s_{1,p_1+1}(v)$ must hold.

Now we consider the postcondition Q of t in E_p. First of all, M_1 is sufficiently general with respect to M. For every variable v in Q, we do a case analysis based on the definition of "sufficiently general" for v in $\tilde{Q}$ of $\tilde{t}$ in E_{1,p_1}.

1. $\tilde{Q}$ reads $\tilde{v}^n$, i.e., the next-state input variable: Let $\tilde{v}^n$ take on the value $s_p(v)$.

2. T_1 contains a read variant of every transition of M that writes to v, or $\tilde{t}$ writes to v, or there is a transition $s \in T$ that writes to v, $\tilde{s} \in T_1$, and s fires in E_0 also: In these cases, let y be the transition that last writes to v at s_j, $0 \leq j \leq p$, and $\tilde{y} \in E_{1,j_1}$, $n_{j_1} = j$, which is also the last transition writes to v. From the above analysis, we have $\mathit{eff}(y, s_j)(u) = \mathit{eff}(\tilde{y}, s_{1,j_1})(u)$ holds for every variable that is read by y. So $s_p(v) = s_{1,p_1}(v)$ must hold.

So it is clear that the postcondition $\tilde{Q}$ of $\tilde{t}$ in E_{1,p_1} must be evaluated to the same value as Q of t in E_p. The above holds similarly if $\tilde{t} \in T_2$.

From the fact that $M_1; P(\tilde{X}) \models T_1$ and $M_2; P(\tilde{X}) \models T_2$, it follows that $\tilde{Q}$ must hold, so does Q in E_p. Finally, if $\exists x \in X$ in the write set of t, from the proposition, we know that t is annotated with the postcondition $P(X)$. From the analysis for any postcondition Q, it follows that $P(X)$ must hold at s_{p+1} of S_M. The induction holds for $i = p + 1$.

Therefore, $M \models T$ holds. $\square$

4.4.2.1 A Simple Example

Now consider the example at the end of Section 4.4.1 again. In that example, we construct two models M_1, M_2 from the model M such that T_1 and T_2 are both sufficiently general with respect to M, and $\models M_1 \wedge \models M_2$ holds implies that $\models M$ holds. However, because the input variables $\tilde{v_1}, \tilde{v_2}$ are completely unconstrained, $\models M_1$ and $\models M_2$ do not hold. Now we show how assume guarantee reasoning developed in this section can help refine M_1, M_2.

We can apply the previous proposition to refine M_1, M_2. In more detail, consider a satisfiable formula $P(V) = (v_1 \geq 0 \wedge v_2 \leq 0)$ and its read variant $P(\tilde{V}) = (\tilde{v_1} \geq 0 \wedge \tilde{v_2} \leq 0)$. From the definition of M_1, M_2, it is clear that both $I_1 \models P(V)$ and $I_2 \models P(V)$ hold. Also, provided that $\tilde{v_1} \geq 0$ and $\tilde{v_2} \leq 0$ hold at each step of M_1, it is clear that $M_1; P(\tilde{V}) \models (v_2 \leq 0)$ holds. Similarly, for M_2, provided that $\tilde{v_1} \geq 0$ and $\tilde{v_2} \leq 0$ hold at each step of M_2, it is clear that $M_1; P(\tilde{V}) \models (v_1 \geq 0)$ holds. Now according to our proposition, $M \models P(V)$ holds. Therefore, the fact that $M_1; P(\tilde{V}) \models (v_3 = \mathrm{true})$ and $M_2; P(\tilde{V}) \models (v_4 = \mathrm{false})$ hold implies that $\models M$ holds.

As noted in the next section, in some cases the actual annotations to be checked can be simplified. For instance it may be the case that M consists of two transactions that can be shown to never overlap, and M_1 and M_2, respectively, contain read variants of these two transactions. In this case, only the final value produced by each transaction can be read by the other transaction. It is not necessary to annotate the nonfinal transitions in this case. Another case in which the invariant can be simplified is when no transition in one of the submodels, M_1 or M_2, writes to a certain variable.

4.5 Implementing the Compositional Approach

In the previous sections, we develop a set of conservative approaches to decomposing the model checking for the monolithic model M_{MC}, into a set of abstract models, using abstraction and assume guarantee reasoning. Now we describe an implementation of the compositional model checking for M_{MC}.

For each transaction U_i in M_{MC}, we build an abstract model $M_i = (V_{MC}, \tilde{V}_{MC}, I_i, T_i, A_i)$ containing the correctness conditions for U_i. In this model, I_i is highly unconstrained compared with I_{MC}. It covers all the states of M_{MC} where the trigger transition of U_i is enabled in M_{MC}. Essentially, I_i consists of all the states where the variables in $V_H \cap W(U_i) \cap R(U_i)$ match with their correspondents in V'_H. For transitions, T_i includes a set of read variants of all the transitions in U_i, including the annotations.

Now we discuss when the read of a variable v is replaced with the read of the input variable $\tilde{v}$ in an abstract model M_i. This is based on how the transaction U_i can interact with other transactions in M_{MC}.

- First, we consider variables $v \in V_L - V_H$. If U_i reads v, another transaction of M_{MC} writes v, and these transactions can overlap in M_{MC}, i.e., one starts before the other is completed, then we replace each read of v in M_i, with the read of the input variable $\tilde{v}$.

- Second, we consider variables $v \in V_H$ which U_i reads. If another transaction writes v, and they can overlap in M_{MC}, then we replace the reads of v with the reads of $\tilde{v}$ in M_i. Moreover, if v' is in the read set of $\mathrm{spec}(U_i)$, then we replace the reads of v' with the reads of $\tilde{v}'$ in M_i; also, we assume that v' and $\tilde{v}'$ always satisfy the constraint $\tilde{v} = \tilde{v}'$ in M_i. The reason to make this assumption is that reads of individual input variables in V_H and V'_H are too unconstrained to verify the specification enableness conditions etc. We will discuss how these assumptions will be guaranteed in the next.

- Third, we consider variables $v \in V_H$ that U_i writes. If another transaction reads v' or writes v, and they can overlap in M_{MC}, then we add $v = v'$ as a postcondition to every transition that writes to v or v' in M_i. That is, we request that in every reachable state of M_i, $v = v'$ holds. This is the guarantee part of the assume guarantee reasoning. In fact, the postcondition $vars_i = vars'_i$ which is checked at the end of the transaction U_i, is another part of guarantee in M_i.

The above process describes one implementation of how we can decompose the model checking of M_{MC} into the checks of each abstract protocol M_i. It is clear that this implementation is just a special case of the abstraction and assume guarantee reasoning presented in Section 4.4. So the abstract models built this way have the following property: $\wedge_i(M_i; P_i(\tilde{V_{MC}}, \tilde{V_{MC}}') \models T_i)$ implies $M_{MC} \models T_{MC}$. Here, $P_i(\tilde{V_{MC}}, \tilde{V_{MC}}')$ is the set of assumptions which are described in the second case.

For any i, if $M_i; P_i(\tilde{V_{MC}}, \tilde{V_{MC}}') \models T_i$ fails due to abstraction using input variables, we can undo the abstraction. In more detail, we can change every read of an input variable $\tilde{v}$ in M_i back to reads of the ordinary variable v. Then we transitively include all the transactions of M_{MC} that write v, to M_i. This is one heuristics of implementing the compositional approach.

Finally, if $M_i; P_i(\tilde{V_{MC}}, \tilde{V_{MC}}') \models T_i$ holds for each abstract model, we still need to check the write-write conflicts, the serializability condition and so on, in order for $\models M_{MC}$ to hold. Currently, we perform these checks monolithically on M_{MC}. Section 4.6.3 will describe the detailed implementation of these checks in Muv. Moreover, we need to find out whether two transactions can overlap in M_{MC}. We implement this by first statically computing the read and write sets of each transaction in M_{MC}, as will be described in Section 4.6.3. If the read set of one transaction overlaps with the write set of another transaction, an assertion will be added to M_{MC}, which asserts that these two transactions will not overlap. Finally, these assertions will be checked in M_{MC} in runtime. If necessary, more fine-grained checks can be performed among transitions, not transactions.

4.6 A Transaction Based Refinement Checker

To check that an RTL implementation of a hardware protocol including coherence protocols correctly implements a high level specification, we develop the following approach as shown in Figure 4.2. We employ Murphi to model the specifications and verify

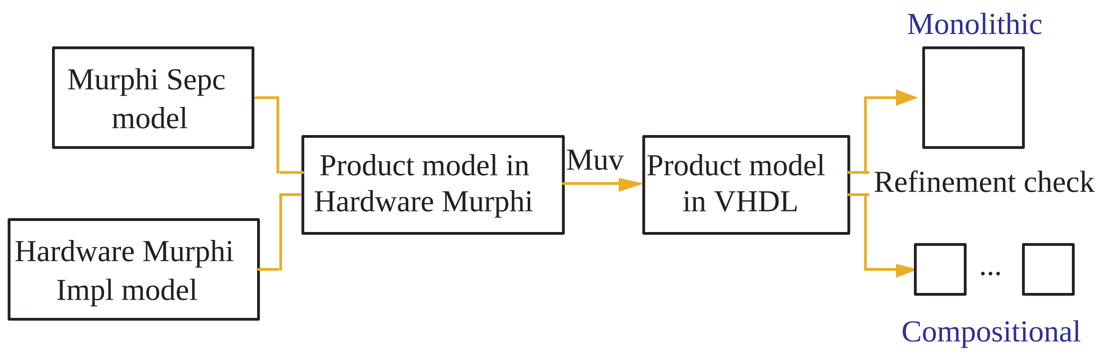


Figure 4.2. The workflow of the refinement check.

them, and employ *Hardware Murphi* to model implementations. Hardware Murphi extends the Murphi language of Dill et al. [50] into a hardware description language. S. German and G. Janssen defined Hardware Murphi and implemented a translator, called *Muv*, from Hardware Murphi into synthesizable VHDL [56, 58]. We extend Muv so that it is not just a translator from Hardware Murphi to VHDL; it can also generate assertions for the monolithic refinement check between the RTL implementations and high level specifications. Unlike the traditional hardware language, Hardware Murphi is more like a concurrent software language, e.g., a threaded language.

The refinement check works as follows. Given a Murphi specification model and an implementation model in Hardware Murphi, we first combine them into one cross product model in Hardware Murphi. We then use Muv to translate the combined model into a VHDL model. With the generated VHDL model, we can employ any existing VHDL verification tool to check the refinement properties.

4.6.1 Hardware Murphi

The Hardware Murphi language is a hardware-oriented extension of Murphi. It was proposed by German and Janssen [58]. Unlike the interleaving semantics of Murphi, Hardware Murphi is a concurrent shared variable language. Its features include the following:

- Each rule once enabled, is assumed to finish in one clock cycle.

- On each clock cycle, multiple rules can read a variable. By variable, we mean a global state variable, or it can be interpreted as a latch in hardware designs.
- On each clock cycle, at most one transition can write a variable.
- It supports *signal*s with modes *in*, *out*, and *internal*, and it supports signal assignments. Signals here are similar to wires in hardware designs.
- It supports *transactions*. The purpose of a transaction is to group a multiple implementation rules to relate to one specification rule. It strictly follows the formal definition of transactions in Section 4.1.
- It supports *chooserule* and *firstrule* to group a set of rules. The semantics of chooserule is that an arbitrary rule is chosen and fired provided its guard is enabled. If the chosen rule is not enabled, no rule will be fired. For firstrule, the first enabled rule in the group is chosen to fire.

Figure 4.3 shows a simple example of how a transaction looks in Hardware Murphi, and how a specification transition is implemented by a transaction. In this example, the specification transition transfers a message from a source node to the destination node. The guard condition under which the transition can be fired is that, (i) the valid bit of the output channel of the source node is true, i.e., there is a message available, and (ii) the valid bit of the input channel of the destination node is false, i.e., the input channel is available. Once the transition is enabled, the action part simply copies the message to the input channel of the destination node, sets the valid bit to true, and resets the output message which is a procedure.

The specification transition is implemented by two steps in the transaction. The first one is the trigger transition, where when the update signal of the buffer in the destination node is true, it copies the message to the buffer, and sets the valid bit to true. The second one is the closing transition, indicated by the statement "reset_transaction." It simply resets the output message in the source node. These two transitions could fire concurrently, or in different clock cycles, depending on the hardware design.

We extend the language of Hardware Murphi with the following features.

```
Rule "transfer msg src to dest"
   outchan.valid = true &
   inchan.valid = false
==>
   inchan.msg  := outchan.msg;
   inchan.valid := true;
   reset(outchan);
Endrule;
```

Specification transition

```
Transaction   "transfer msg src to dest"
Rule "r1. buffer receive input msg"
   buf.update_signal = true
==>
   buf.msg := buf.new_msg_signal;
   buf.valid := true;
Endrule;

Rule "r2. source reset output msg"
   source.reset_signal = true
==>
   reset(source);
   reset_transaction;
Endrule;
Endtransaction;
```

Implementation transaction

Figure 4.3. An example of transaction.

- A Hardware Murphi model can include another model by using "- -include". With this feature, a specification model can be easily included into an implementation model.

- A simple rule in Murphi and Hardware Murphi can now associate with a unique identity, by using rule:id guard $\longrightarrow$ action. We also introduce two new statements: $\langle\langle$ id.guard() $\rangle\rangle$ and $\langle\langle$ id.action() $\rangle\rangle$, where "id" is an identity of a simple rule. The first statement is meant to check that the guard of the rule with identifier id must hold, and the second is to execute the rule. With these extensions, a specification transition can be easily combined to an implementation transaction, so that whenever the transaction fires the specification transition also fires.

- A Hardware Murphi model can now have a section which describes the correspondence relationship on the joint variables between a specification and implementation models.

- The notion of *transactionset* is added. A transactionset represents a set of transactions which only differ in certain parameters. It is similar to the notion of ruleset in Murphi.

These extensions can help users ease the refinement check.

4.6.2 An Overview of Muv

Muv is a translator from Hardware Murphi to synthesizable VHDL, initially implemented by German and Janssen from IBM [58]. We extend it so that it can automatically generate assertions for refinement check. Internally, Muv collects all the global state variables declared in a Hardware Murphi model, to construct a state record data structure. The translator is implemented using a two-process template of VHDL – a *combinational* process collects all the updates happening in every transition in the model sequentially, and a *sequential* process simply copies these updates to the global state record.

The updates to state variables in Hardware Murphi are processed in Muv as follows. Updates within a rule are immediate, while they become visible to other rules only in the next cycle. In more detail, each rule has a local copy of the global state record. Updates within a rule only happen to the local copy. In each clock cycle, every local copy is first initialized to the value of the global state record. The local copy is then updated if the corresponding rule fires. Finally, the global copy collects all the updates happening in each local copy. Because multiple rules can fire concurrently in each clock cycle, we need to check the write-write conflicts, i.e., if multiple rules of the model can write to the same state variable concurrently.

We will describe how Muv can automatically generate the assertions of refinement check in the next section. These assertions will be contained in the resulting VHDL model, and they can be checked by any VHDL verifier.

4.6.3 Assertion Generation for Refinement Check

For refinement check, Muv will generate altogether three category of assertions: (i) assertions for checking write-write conflicts because of concurrent executions, (ii) assertions for checking the serializability condition, and (iii) assertions for checking the variable equivalence. In the following, we will illustrate how to generate these assertions in Muv.

4.6.3.1 Checking Write-Write Conflicts

For every global state variable in a Hardware Murphi model, we associate it with a subset of *update bits*. Update bits are an array of boolean variables, to track which global variables get updated in each clock cycle. Each global variable corresponds to a continuous range of the update bits, and the ranges of two different variables never overlap. The range for a global variable is obtained based on its data type. Currently, the data types that Hardware Murphi supports include

$$boolean \quad enum \quad subrange \quad record \quad array$$

Complex data types can be declared by nesting “record” or “array.” For each data type, we assign a range of update bits for it in the following way.

For the *boolean* type, we assign two bits. For *enum*, the number we assign to it is equal to the number of identifiers declared in the enumeration type, and this is similar for the *subrange* type. For *record*, the number of bits is the sum of all the bits that we assign for each field in the record. Finally for *array*, the number of bits is equal to the multiplication of the number of bits of the index, and that of the element of the array. For each global variable, Muv assigns the update bits to the variable according to the declaration order of the variable in the model. For example, for the first global variable, the range will start from 0 and end with the number of bits assigned to this variable.

With update bits, now we describe how write-write conflicts can be checked in Muv. For each global variable that appears on the left-hand side of an assignment, we add two more statements for the assignment: (i) immediately before the assignment, we assert that all the update bits associated to the variable must be true, and (ii) immediately after the assignment, all these update bits are set to true. For example, suppose v is a global state variable and $v := d$ is an assignment statement in the model. Also suppose the range of the update bits of v is $5..10$. Then the assignment will be transformed to:

In the above, the write-write conflicts check assumes that for every transition in a Hardware Murphi model, a variable can only be updated once. We think this is a

```
for i in 5..10 do  assert (update_bits[i] = false);   end;
v := d;
for i in 5..10 do  update_bits[i] := true;   end;
```

reasonable assumption for most RTL implementations, as each transition is supposed to finish in one clock cycle. Otherwise, the assertion failure is just a false alarm. To compute the range of the updates bits for a global state variable, we can do it either statically or dynamically. In more detail, if the variable is a variably indexed array access, then the range depends on the index itself so it can only be computed in runtime, at each clock cycle. For all the other cases, the range of the update bits can be computed statically.

For write-write conflicts check, two optimizations can be implemented. First, once the range of the update bits for a variable is computed, it can be cached so that there is no need to re-compute it for the following accesses to the variable. Similarly, this can be done for the size of update bits to be assigned to each data type. Second, the number of update bits can be reduced, and as a result, the number of assertions to be checked for write-write conflicts can be reduced. This is based on the observation that if a primitive data type is not used as the index type of an array, then we can assign just one bit for it. For example, suppose there is a global variable v in a Hardware Murphi model as follows

```
T1: 1 .. 100;
T2: array [T1] of T1;
v:  T2;
```

Here, $\mathrm{T1}$ and $\mathrm{T2}$ are type declarations. Without optimization, the number of update bits that v will be assigned is $10000 = 100 \times 100$. As a result, for every update of v in the model, 10000 assertions need to be checked. However applying our optimization, the number of bits for v can be reduced to 100. That is, we can assign just one bit for each element of $\mathrm{T2}$. This can remove many of the unnecessary assertions for write-write conflicts check.

4.6.3.2 Computing Read and Write Sets

As described in Section 4.3 and 4.4, for each rule and each transaction, we need to compute the read and write sets of variables, for refinement check. In Muv, the read and write sets are computed statically, with the overall process as follows:

1. Constant propagation for the whole model[1].

2. Pre-processing for all the *transactionset*'s and top-level *ruleset*'s.

3. Static computation of read and write sets.

In more detail, constant propagation is done on the abstract syntax tree (AST) of the monolithic model, with a bottom-up traversal [110]. After constant propagation, the preprocessor of Muv enumerates all the possible constant values for each iterator of transactionsets and top-level rulesets, to generate a list of transactions and simple rules.

Now we describe how the read and write sets of each simple rule are obtained. For every statement in the rule, we statically compute its read and write sets by tracking every expression in the statement. Basically, the designator on the left-hand side of an assignment will be added into the write set, while other expressions will be added into the read set. For function and procedure calls, the actual parameters will be added into the write set (which is clearly conservative). At the same time, the corresponding function or procedure declarations will be recursively analyzed for read and write sets. Finally, the read set of the rule is the union of those for every statement in the rule, and this is similar for the write set.

More specifically, for joint variables of array types, we differentiate variably and constantly indexed array access. For variably indexed array access, the variable of the array type as a whole is added into the read or write set, while for constantly indexed array access, the variable with the specific index will be added. Therefore, constant propagation and preprocessing can help make the read and write sets more precise.

Finally, for read sets, we need to process the reads of signals differently than that of state variables. In Hardware Murphi, signal assignments can be seen as functions over

[1]This was done mainly by G. Janssen from IBM.

other signals and state variables. So for each read of a signal, we need to add the set of state variables to the read set where the value of the signal depends on these variables.

This is implemented by first building a mapping for each signal assignment, with the first entity being the destination signal, and the second being the list of variables and signals which appear on the right-hand of the assignment. Then, the second entity of the mapping is propagated to other mappings, to replace the appearance of the first entity. This is done transitively until a fixed point is reached. For example, suppose in a Hardware Murphi model there exist three global variables *v1, v2, v3*, and three signal assignments for signals *s1, s2, s3* as follows:

$$\begin{array}{lll} \text{s1} & <= & \text{s2;} \\ \text{s2} & <= & \text{v1 \& v2;} \\ \text{s3} & <= & \text{s1} \mid \text{v3;} \end{array}$$

In the above, we use $<=$ to represent signal assignments. Then the value of signal *s2* depends on that of variables *v1* and *v2*. So the read of *s2* is in fact the read of *v1, v2*. After transitive propagation, the read set of *s1* becomes $\{$*v1, v2*$\}$. Similarly, the read set of *s3* is $\{$*v1, v2, v3*$\}$.

4.6.3.3 Checking Serializability

As defined in Section 4.3.1, serializability means that if multiple specification transitions can execute concurrently in the same clock cycle in the monolithic checking model, they can be converted to an interleaving execution, and still reach the same state. In Muv, we implement serializability by analyzing the set of read and write sets of specification global variables for each rule.

More specifically, for each rule in the monolithic model, we statically compute its read and write sets of specification variables. Let the read and write sets be $RS(r)$ and $WS(r)$, respectively, for a rule r. For each pair of rules ri and rj, we statically check if their read sets and write sets overlap. That is, we check

$$\begin{array}{ll} (1) & RS(ri) \cap WS(rj) = \emptyset \\ (2) & WS(ri) \cap RS(rj) = \emptyset \end{array}$$

We will use a directed graph G to store the above read-write dependent relationship for the monolithic model. In more detail, if (1) holds we add an edge $ri \rightarrow rj$ to G.

Similarly, we add $rj \rightarrow ri$ to G if (2) holds. In either case, we add an assertion to the resulting VHDL model, asserting that rules ri and rj cannot fire concurrently in any clock cycle.

Such assertions can be represented by adding an auxiliary boolean variable $rule\_active$ for each rule r in the monolithic model. At the beginning of each clock cycle, all the $rule\_active$'s are set to false; when a rule is enabled and fires, its corresponding $rule\_active$ is set to true. Now the assertion for two rules ri, rj with read-write dependency can be simply represented as $\neg(rule\_active(ri) \wedge rule\_active(rj))$.

Once the resulting VHDL model is model checked with these assertions, we can conclude serializability based on the directed graph G. In more detail, if the assertion $\neg(rule\_active(ri) \wedge rule\_active(rj))$ holds, we can remove the edges $ri \rightarrow rj$ and $rj \rightarrow ri$ from G. After all such edges are removed from G, we check if there exists a cycle in the graph. A classical depth-first search for backedges [135] will work here. Finally, if cycles do not exist, we can claim that serializability holds for the monolithic model, otherwise fails.

It is clear that our approach of checking serializability is conservative. This is because if multiple rules can fire concurrently in the same clock cycle, and there exists a cyclic read-write dependency among them, our approach must conclude that serializability fails. The conservativeness means that it is possible that serializability holds, while our approach concludes failure. For example, suppose there exist three rules ri, rj, rk in a monolithic model, and their static read-write dependencies are as follows:

$$ri \rightarrow rj \quad rj \rightarrow rk \quad rk \rightarrow ri$$

If in each clock cycle exactly two rules can fire concurrently, then our approach will conclude serializability failure, while it actually holds.

The conservativeness can be improved by checking cycles of read-write dependencies at runtime. That is, in each clock cycle, for all the rules that are enabled in the cycle, we check if there exists a read-write dependency cycle. By doing this, the serializability check can be made more precise. In fact, it happens that our serializability check is very similar to that in the database transactions [128].

4.6.3.4 Checking Variable Equivalence

As described in Section 4.3, in the monolithic model checking for refinement, we need to check that the postcondition $\neg alive_i \Rightarrow vars_i = vars'_i$ holds for every transaction U_i in the model. Here, $alive_i$ is the alive variable of U_i, and $vars_i$ is defined to be the vector of all variables written by either the transaction U_i, or by the corresponding specification transition. Since for all the variables $v \in V_L - V_H$, $v' = v$ holds, we only need to consider those variables in V_H or V'_H, i.e., the *joint* variables in both the specification and the implementation models.

In Muv, for every transaction U_i in the monolithic model, we first compute the joint variables in the write set of U_i and the corresponding specification transition. Let these variables be J. Then for all the variables in J, we construct a formula $\wedge_{v \in J}(v = v')$. This is exactly $vars_i = vars'_i$ in the definition of the postcondition. Finally, whenever the closing transition of U_i fires, the formula is checked immediately after the clock cycle. Currently in Muv, the correspondence relationship between V_H and V'_H is supposed to be provided by users and specified in the Hardware Murphi model.

4.6.4 Experimental Benchmarks

We have applied the refinement check on a simple 2-stage pipelined stack example, and a coherence protocol engine with certain realistic features. For the stack example, we model altogether three operations: push, pop and reset. Push and pop involve reading and writing the stack size and the stack data, and reset simply resets the stack to an empty one. The transaction of a push or a pop operation contains two transitions, and it takes two continuous clock cycles to finish in the implementation. The transaction of a reset operation simply sets the stack size to zero. In the specification, every operation is modeled by one rule, and two pushes can overlap also for two pops. The stack example as a whole is very simple and it passes the refinement check. In the following, we will mainly focus on the coherence protocol example.

We use the German protocol [55] as the specification of our benchmark. This protocol is a simple invalidation-based MSI [48] protocol. The protocol models two addresses and two nodes, with each node being the home node for one address. The reason to model

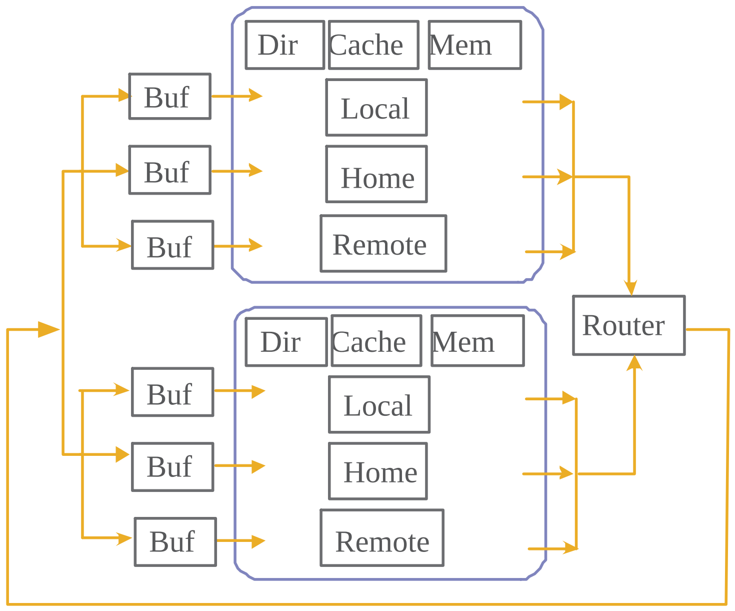


Figure 4.4. An overview of the implementation protocol.

two addresses instead of one as in coherence properties verification is that, the refinement check is used to check the RTL implementation meets the protocol specification. It will be more realistic to model multiple addresses for RTL implementations. In the model, each node contains a memory, a cache and a directory. To avoid communication deadlocks and allow messages received out of order, multiple communication channels are introduced for each pair of nodes. Altogether the specification model has about 500 LOC in Murphi.

The implementation model [56] is much more complex. Each node contains three controlling units: the local, home and remote units. It also contains a directory, a cache and a memory. Different units in one node can function simultaneously, so that one unit may have to wait for several clock cycles before it can access a shared resource. Figure 4.4 shows the overview of the implementation protocol.

In the implementation model, other than the two nodes there also exist one router and six buffer units. The router is used to transfer messages from source nodes to destination buffers, and it can select at most one message to transfer from the source node to the destination side buffer at each cycle. The buffers are used to store messages in the destination side, before the destination nodes pick the messages and process them. Altogether, the monolithic model has about 1500 LOC in VHDL.

4.6.5 Experimental Results

For the cache protocol example, we re-engineered the VHDL implementation model into a Hardware Murphi model, which results in about 2500 LOC. The heuristics that we used for selecting commit transitions for transactions are as follows. For each transaction, we select the first transition declared in the transaction whose write set overlaps with the joint variables, to be the commit transition. Commit transitions selected in this way are similar to the notion of "commit step" in the work of aggregation of distributed transactions [116].

Before the refinement check, the basic coherence properties have been verified in both the specification and the implementation models. However with refinement check, we still found three bugs in the implementation model. These bugs include:

1. In two continuous clock cycles, the router unit successfully transfer the first message, but it can lose the second message.

2. For one coherence request, the home unit of the home node can reply the request twice in two continuous cycles.

3. For one coherence reply, the cache unit can be updated twice in two continuous cycles.

We believe that these bugs are easy to miss. That is, they are hard to remember to check by writing assertions. However, our refinement check is an automatic way to construct such checks, to ensure we check enough properties.

Moreover, we have applied our compositional approach to the cache example after the bugs are fixed. Using an IBM verification tool called SixthSense [23], we showed that with the datapath of a cache line as one bit in width, both the monolithic and compositional approaches can finish refinement check within 30 minutes. However, with the datapath being 10 bits, the monolithic approach cannot finish the refinement check in over one day, while the compositional approach can finish it in about 30 minutes.

4.6.5.1 Transaction Details

After the bugs are fixed, there are altogether 13 transactions including transactionsets in the benchmark. In the protocol, a transaction can contain one, two, or three rules. Coincidentally all the rules in a transaction always fire concurrently in the same clock cycle. An example that a transaction contains three rules is a transaction in which a client node receives a grant reply from the home node. The three rules are: (i) the cache unit of the client updates the cache line, (ii) the local unit sets the request as completed, and (iii) the buffer unit resets the reply message. This transaction is modeled as one rule in the specification. That is, the client node receives the reply and the source node resets the message.

In our example, one rule can be in different transactions. In that case, each instance of the rule has a different *alive* variable and a trigger rule associated with it. It is worthwhile to mention that all the rules within a transaction do not necessarily need to fire concurrently, or in consecutive clock cycles. There could be gaps in between, and all our proof rules still work. For example, in hardware because all the processing units can be active at the same time, a unit may have to wait several clock cycles to access a shared resource.

In our benchmark, different transactions can fire concurrently. For example, one transaction is that a message is transferred from the source node to the destination side buffer. It contains two rules: (i) the destination buffer delivers message via an input channel, and (ii) the local unit of the source node resets the corresponding output channel. Another transaction is that the home node prepares an invalidate request for an address. These two transactions can fire concurrently in the same clock cycle, and our method is still able to check the refinement in this case.

4.7 Summary

In this chapter, we investigate techniques to check the consistency between an RTL implementation of a hardware protocol and a high level specification. Due to the differences in modeling and execution, we develop transactions to relate one step in the specification, to multiple step executions in the implementation. We then develop a formal theory to check the refinement relationship, i.e., under which conditions can we claim the consistency. The overall approach is as follows: the two models are combined into one such that whenever a transaction of the implementation starts executing, the corresponding specification transition also fires. In this combined model, for every *joint* variable, we require that its value must be equivalent, if there is no transaction actively updating it.

We also develop a compositional approach to reduce the verification complexity of refinement check. The basic idea is to allow the implementation transaction realizing a specification transition to be situated in sufficiently general environments, using abstraction and assume guarantee reasoning.

We employ Hardware Murphi which is a hardware language, to model RTL implementations of hardware protocols. Based on this language, we have mechanized the process to generate properties for refinement check, for the monolithic approach. That is, given a Hardware Murphi model of the RTL implementation which includes the specification model, assertions can be automatically generated for refinement check. This is implemented based on an initial effort from IBM which translates Hardware Murphi models to synthesizable VHDL models.

Illustrated on a cache coherence protocol engine with realistic features, we show that our approach is able to generate enough assertions to catch the bugs that are easy to miss. Also, we show that our compositional approach can effectively reduce the verification complexity of the refinement check. For future work, we plan to apply our approaches to other hardware designs such as complex pipelined systems. We also plan to explore how to mechanize the compositional refinement check approach.

CHAPTER 5

CONCLUSIONS AND FUTURE DIRECTIONS

In this dissertation, we develop *scalable* verification techniques for hierarchical cache coherence protocols in their high level specifications, on several multicore cache coherence protocols with certain realistic features. We also develop a *refinement* verification approach to checking whether a hardware implementation of a coherence protocol meets the high level specification, for nonhierarchical coherence protocols. Of course, we will also need to check refinement for hierarchical coherence protocol implementations as well, which will be the future work. For the refinement check, we further develop scalable verification techniques to reduce the verification complexity. Compared with traditional approaches, our approach can effectively reduce the verification complexity, and generate enough properties for the verification of hardware implementations.

In our work, we first employ metacircular assume guarantee reasoning to reduce the verification complexity of finite instances of protocols for safety, using an explicit state model checker. That is, given a hierarchical protocol, we first construct a set of abstract protocols. Each abstract protocol simulates the original protocol but with smaller state space. The abstract protocols depend on each other both to define the abstractions as well as to justify them. We show that in case of our hierarchical coherence protocols, the designers can easily construct each abstract protocol in a counterexample guided manner. The approach is practical, and it has been successfully applied to several hierarchical multicore cache coherence protocols with realistic features.

Furthermore, we decompose hierarchical protocols one level at a time through assume guarantee. To achieve this, we propose that hierarchical protocols be designed and implemented in a loosely coupled manner. Also, we use history variables in an assume guarantee manner, to verify noninclusive and snooping hierarchical protocols.

We have partly mechanized our compositional approach, in the aspects that (i) given a hierarchical protocol and the protocol details to be removed, how to automatically generate the abstract protocol, and (ii) given an error trace from an abstract protocol, how to automatically identify whether it corresponds to a genuine error in the original hierarchical protocol. For three multicore hierarchical protocol benchmarks, we show that our approach can bring about 95% reduction in the total state space during any single explicit state enumeration.

Moreover, we develop a refinement theory which defines under which conditions can we claim that an RTL implementation correctly implements a specification. We also develop a compositional approach to reducing the verification complexity of the refinement check, using abstraction and assume guarantee reasoning. Besides, we have mechanized the process to generate assertions for refinement check, based on the Hardware Murphi language and Muv, a translator from Hardware Murphi to VHDL models. The experimental results show that our approach is very effective to find out bugs that are easy to miss in a hardware protocol implementation with realistic features. Finally, we show that our compositional approach can effectively reduce the refinement check runtime from over a day to about 30 minutes on the protocol.

Overall, we have conducted some of the first research on the verification of hierarchical cache coherence protocols in the high level specifications, and refinement check for hardware protocol implementations. We have demonstrated the benefits of our approaches on several hierarchical coherence protocols, and a hardware implementation of coherence protocol with realistic features.

In retrospect, my PhD work has been a long journey and it involves a few more projects on cache coherence protocol verification. These projects include predicate abstraction for Murphi [38] and bounded transaction based testing for cache coherence protocol specifications [37]. The first project was motivated to check whether predicate abstraction is an efficient alternative to the explicit state enumeration approach as provided by Murphi. The second one was intended to develop an efficient approach to testing nonhierarchical cache coherence protocol specifications. In both projects, we achieved acceptable results but also encountered several problems. In the following, I will briefly describe each project and my learning from it.

5.1 Predicate Abstraction for Murphi

Predicate abstraction is another technique other than explicit state enumeration to prove properties in a finite or infinite state system. The basic idea of predicate abstraction was first described by Graf and Saïdi [63]. It is a technique trying to abstract large scale finite or infinite state systems into tractable finite state systems. The states in the abstract systems correspond to the truth values of a set of predicates in the original concrete systems.

Our work is derived from Das and Dill [49] who use overapproximation based abstraction. Overapproximation based abstraction is conservative, meaning that if a property is shown to hold on the abstract system, the concrete version of the property must also hold on the original system. However, if the property fails to hold on the abstract system, it is not clear whether the concrete version holds on the original system. Because the original implementation of [49] was not publicly available and several technical details cannot be inferred from their paper, we implemented the predicate abstraction for Murphi (called *PAM*) using a decision procedure CVC Lite (CVCL) [18], to verify cache coherence protocols. PAM first converts a coherence protocol described in the Murphi modeling language into a symbolic representative in the CVCL language, and then uses the decision procedure to compute the validity of predicates.

For PAM, the concrete state system is the explicit state system of a coherence protocol in Murphi. The set of predicates of interest is the set of properties to be verified in the Murphi model. For N properties to be verified in the concrete system, the abstract system will consist of N boolean state variables, with the value of each variable representing whether the property (or the predicate) holds in the concrete system. To construct the abstract system, the initial abstract states and the abstract state transition relation need to be built. In the following, we will describe each of them briefly.

To obtain the initial states of the abstract system, PAM first computes the initial states of the concrete system. For each initial concrete state, PAM will then evaluate using

CVCL whether it holds for individual predicates, thus obtaining an initial abstract state. For example, suppose a concrete system is described using variables $\{pc,\ y_1,\ y_2\}$ and there are two predicates $\phi_1 \equiv (pc = 1),\ \phi_2 \equiv \neg(pc = 1) \vee (y2 = 2)$. Also, suppose an initial concrete state has the value $\{pc = 1, y_1 = 1, y_2 = 0\}$. Then the corresponding initial abstract state will be $\{b_1 = \text{true}, b_2 = \text{false}\}$ where b_1, b_2 are the boolean variables correspond to ϕ_1, ϕ_2 individually.

For the abstract transition relation, PAM computes it in the following manner. For each abstract state, PAM first employs a procedure called *concretization* to obtain the set of concrete states which correspond to the abstract state. Concretization is the reverse process of abstraction; it computes the set of concrete states which after abstraction is equal to the abstract state. PAM then applies the concrete transition relation on the set of concrete states, to obtain the set of concrete successor states. After that, abstraction is applied to obtain the set of abstract successor states. More detail can be found in our technical report [38].

In the above process, if the set of all abstract successor states is equal to the set of all abstract current states, i.e., a fixed point is reached, then all the abstract state space has been explored. At this time, if a boolean state variable which corresponds to a predicate in the concrete system always has the value true in the abstract state space, then the predicate must hold in the concrete system, i.e., the corresponding property must hold. Otherwise, it is not clear whether the property holds as PAM employs a conservative abstraction.

We have implemented the above algorithm in PAM. Given a coherence protocol model in Murphi with user provided predicates, PAM first converts the data types described in Murphi into CVCL data types, and also the global state variables. After that, it will convert each expression, statement and rule of the Murphi model into that in the CVCL language. The output of the conversion is a C++ file which represents the abstract system in CVCL. With this file, we can employ CVCL as the decision procedure to compute the fixed point of the abstract system as describe in the above, to check the validity of the properties to be verified in the concrete system.

One optimization we made for PAM is to split large expressions in CVCL into a set of smaller ones, to improve the efficiency of the decision procedure. In more detail, according to the Murphi syntax, for every concrete state in Murphi, at most one enabled rule can be chosen to generate the successor concrete state. Thus, the transition relation in the concrete system is in fact the disjunction of all the rules in the protocol model. We found out that this disjunction can introduce very large expressions in CVCL and the decision procedure can run very slowly on such expressions. Therefore, we split such large disjunctive expressions into a set of smaller expressions each of which representing the transition relation of one rule. More detail of the optimization can be found in our technical report [38].

We have applied PAM to two widely studied coherence protocol models: the German [55] and the FLASH [81] protocols. Experiments showed that after optimization, the runtime on both models is acceptable: about 150 seconds and 45 minutes individually.

5.1.1 Learning from PAM

During the process of building PAM, I learned the basic techniques of predicate abstraction, abstraction, concretization, and the basics of the decision procedure CVCL. Although we achieved acceptable experimental results, there still exist several problems which we have not solved, due to the main interests of our funding agency. The following are several lessons I learned from the project to make PAM more efficient.

The first lesson is that in order for PAM to work efficiently, we need to be very familiar with the internal of decision procedures. That is, we should be able to directly locate the performance bottleneck and fix the problem of the decision procedure if there is any. Otherwise, the overhead of waiting for others to fix the problem can be big. The second lesson is that the transition relation that we used in the abstract system is not efficient. The reason is that for every abstract state, after it is concretized, the *concrete transition relation* is applied to obtain the set of successor concrete states. Using a decision procedure to precisely represent every detail of the concrete transition relation can be very inefficient. I think since we use overapproximation for the abstraction, we can also use overapproximation to represent the concrete transition relation as well. The details of the solution are worth investigating. The third lesson is that refinement is necessary for an abstract system that employs overapproximated abstraction. Otherwise, in most cases we cannot tell whether the properties hold in the concrete system.

5.2 Bounded Transaction Based Testing

As said earlier, the basic motivation of this work is to develop efficient debugging techniques for nonhierarchical cache coherence protocol specifications, as sometimes the state space of such protocols can be very large. To achieve this goal, we propose bounded transaction based testing based on explicit state space enumeration techniques. The notion of transaction here is different from that used in our refinement check between protocol implementations and their specifications. Instead, we use transactions to represent a causal sequence of transitions which starts with a coherence request and ends with a coherence reply. For example, the following sequence of transitions form a simple transaction: (i) a cache agent requests a shared copy, (ii) the local directory receives the request, (iii) the directory grants a shared copy, and (iv) the requester receives the reply.

The basic idea of our approach comes from the observation that traditional bounded model checking (BMC) is often ineffective for debugging cache coherence protocols, because when the breadth first search goes up to a certain depth, it cannot guarantee that significant behaviors are covered. In our approach, we propose bounded search by limiting the *number* and the *variety* of transactions which can exist concurrently. In more detail, we associate each transaction with an attribute as its type. There are altogether two types of transactions we consider: *shared* and *exclusive*. The type of shared means that the transaction is about a coherence request for a shared copy, and the type of exclusive is similar. For shared transactions, we restrict that before the transaction finishes, only exclusive transactions can start, while for an exclusive transaction, we allow both types of transactions to start before it finishes. The motivation behind this is that usually two shared transactions will not introduce interesting scenarios (or corner cases), while a shared and an exclusive, or two exclusive transactions can.

Other than restricting the types of transactions which can exist concurrently, we also restrict the number of transactions. More specifically, we associate each transaction with a *quota* which represents how many more transactions can start in its lifetime. Suppose a transaction has a quota of $N(N > 0)$. Then after a new transaction is started before the first one finishes, the quota of the first one will decrease by 1. By doing so, we limit the total number of transactions which can exist concurrently, to avoid exploring uninteresting behaviors.

We have implemented the basic idea of bounded transaction based testing in Murphi, and we call it *BT*. The following heuristics are employed to obtain the transaction information from a coherence protocol model. First, we perform a classical breadth first search on the coherence protocol model with smaller instances of cache agents, e.g., two cache agents for one address with 1-bit of cache line data. Such a breadth first search can generate a causal graph of which transition enables which other. We then perform filtering on the graph by dropping duplicate edges and pruning backward edges. For the German and FLASH protocols, we showed that such heuristics can generate the basic information of transactions. This information is recorded by associating each rule in the transaction with a number called *weight*. With the transaction information, we modify the BFS algorithm in the standard Murphi to implement the bounded transaction based testing.

Experiments showed that for the two errors that we injected to the German and FLASH protocols, BT was much more efficient than the classical BMC to find out the errors, especially when the number of nodes and the datapath of cache lines increase. This is reasonable, as increasing the number of nodes will of course increase the number of states to be explored in each level of the BFS, while our approach still selectively chooses only a subset of the interesting transactions to explore. More details can be found in the technical report [37].

5.2.1 Possible Future Directions for BT

I think bounded transaction based testing is a natural idea to debug cache coherence protocols, and future directions can be investigated in the following directions.

First, currently we obtain the information of transactions by concretely performing BFS on the protocol model with smaller protocol instances. Recent work in [136] suggests that we can probably obtain this information directly from the message flows available in most protocol descriptions. To me, the message flows are exactly the kind of information we want for transactions.

Second, other than the shared and exclusive types of transactions, we can identify more types of transactions from the message flows. The question of what types of transactions can exist concurrently, i.e., what kind of behaviors we want to explore, is another topic to be investigated. Finally, more protocol benchmarks with realistic features will help us evaluate our approaches.

5.3 Future Work for the Dissertation

For the verification of hierarchical cache coherence protocols for futuristic processors, I think the following work can be investigated with respect to the two aspects that we have worked on.

5.3.1 Verification of Hierarchical Protocol Specifications

For the verification of hierarchical coherence protocols in their high level specifications, our work and related work [27, 136] have mechanized most part of the approach. I think integrating them together into one framework is important. During the process, I would expect many subtle issues will come up which require new ideas and solutions. It will also be good to provide simple graphical user interfaces of the framework with click buttons. By doing so, our compositional approaches can be used routinely and automatically in the verification of hierarchical protocol specifications.

Another topic is that in practice writing a coherence protocol in its high level specification usually requires deep protocol awareness. The overhead lies in that protocol designers are not motivated to write them and it is difficult for verification engineers to write a correct one. Recently there has been some work from industry which starts to address this problem. For example, in [73] the authors propose an automatic approach to extracting models from design documents, e.g., block diagrams, timing diagrams, pipeline diagrams, message flows, etc. I think this is a promising step to automatically extract protocol specifications for verification, although the practice has different varieties and subtle issues. Future work in this area are worth trying.

Another topic which I think is interesting to investigate is to combine our compositional verification approach with the parameterized verification techniques proposed by Chou et al. [42]. From my understanding, both approaches share the same underlying theory. Thus, it seems straightforward to combine both the approaches to verifying hierarchical parameterized coherence protocols. Basically, we can first apply our compositional approach to decomposing a hierarchical protocol into a set of abstract protocols. After that, we can apply the approach in [42] for each abstract protocol.

5.3.2 Refinement Check

For the refinement check between protocol specifications and their RTL implementations, we have just started this work with basic theories and experiments. A lot more work can be done in this direction.

First, I think we need more benchmarks of nonhierarchical protocol implementations with *realistic* features. These features can include e.g., pipeline details, out of order executions, cache controllers with write buffers and read buffers, and so on. Protocol implementations with these details will force us to consider corner cases in our basic refinement theory, the monolithic checking approach and the compositional approach as well. I think this is the first priority to be investigated in this area.

Second, we have mechanized most of the monolithic checking approach but have not worked on the compositional approach. Basically, the compositional approach consists of a conservative abstraction and assume guarantee reasoning based refinement. In the very high level, they are similar to those used in our compositional approach for the verification of hierarchical protocol specifications. I think probably similar ideas can be applied for the mechanization; of course, new problems will come up as the details of the compositional approaches are all different.

Third, we have not applied our refinement check approach to hierarchical coherence protocol implementations yet. It would be good to see how to combine our techniques developed for the verification of hierarchical protocol specifications, with the refinement check approaches. Intuitively, such a combination is achievable at the very high level. That is, we first apply our compositional approach for hierarchical protocol specifications to building a set of abstract protocol specifications. We then decompose the protocol implementation accordingly to build the set of abstract protocol implementations. After that, we can apply our refinement checking approach to individually verify whether each abstract protocol implementation meets the corresponding abstract protocol specification. Again, hierarchical protocol implementations with realistic features are needed to evaluate the approaches.

To obtain protocol benchmarks with realistic features, I think one way is to work closely with architectural groups in academia in cache coherence protocols. Usually their benchmarks have certain implementation details which also achieve reasonable performance goals. I have been very lucky to benefit a lot from collaboration with industrial researchers. However, in most cases intellectual property protection is very strict in industry. I hope such healthy and productive collaboration can continue for future research in this area.

REFERENCES

[1] http://www.bluespec.com/.

[2] http://www.cs.utah.edu/formal_verification/hldvt07.

[3] http://verify.stanford.edu/dill/murphi.html.

[4] http://www.cs.utah.edu/formal_verification/software/murphi/murphi.64bits/.

[5] http://www.cs.utah.edu/formal_verification/fmcad08submission.

[6] AMD Athlon^{TM} 64 Processor Product Data Sheet. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24659.PDF.

[7] IEEE Std. 1666^{TM}-2005, SystemC^{TM} Language Reference Manual. http://www.systemc.org.

[8] Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/design/processor/manuals/248966.pdf.

[9] M. Abadi and L. Lamport. Conjoining Specifications. In *ACM Transactions on Programming Languages and Systems*, 1995.

[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] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Karanz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Int'l Symposium on Computer Architecture*, pages 2–13, 1995.

[12] R. Alur and T. A. Henzinger. Modularity for Timed and Hybrid Systems. In *Concurrency Theory*, pages 74–88, 1997.

[13] R. Alur and T. A. Henzinger. Reactive Modules. In *Formal Methods in System Design*, 1999.

[14] Arvind, R. Nikhil, D. Rosenband, and N. Dave. High-Level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *International Conference on Computer Aided Design*, 2004.

[15] Arvind and X. Shen. Using Term Rewriting Systems to Design and Verify Processors. In *IEEE Micro*, 1999.

[16] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[17] M. Azizi, O. A. Mohamed, and X. Song. Cache Coherence Protocol Verification of a Multiprocessor System with Shared Memory. In *Int'l Conference on Microelectronics*, pages 99–102, 1998.

[18] C. W. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Computer Aided Verification*, pages 515–518, 2004.

[19] 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.

[20] F. Baskett, T. A. Jermoluk, and D. 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.

[21] B. Batson and L. Lamport. High-Level Specifications: Lessons from Industry. In *Formal Methods for Components and Objects*, 2002.

[22] K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 305–308, 2002.

[23] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable Sequential Equivalence Checking across Arbitrary Design Transformations. In *Intl. Conference on Computer Design*, 2006.

[24] R. Bhattacharya, S. German, and G. Gopalakrishnan. Symbolic Partial Order Reduction for Rule-based Transition Systems. In *Correct Hardware Design and Verification Method*, pages 332–335, 2005.

[25] R. Bhattacharya, S. M. German, and G. Gopalakrishnan. Exploiting Symmetry and Transactions for Partial Order Reduction of Rule Based Specifications. In *SPIN Workshop on Model Checking of Software*, pages 252–270, 2006.

[26] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Milticast Snooping: A New Coherence Method Using a Multicast Address Network. In *Int'l Symposium on Computer Architecture*, page 1999, 294-304.

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

[28] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Int'l Symposium Computer Architecture*, 2007.

[29] J. M. Borkenhagen, R. D. Hoover, and K. M. Valk. EXA Cache/Scalability Controllers. In *IBM Enterprise X-Architecture Technology: Reaching the Summit*, 2002.

[30] F. Briggs, M. Cekleov, K. Creta, M. Khare, S. Kulick, L. P. Looi, C. Natarajan, S. Radhakrishnan, and L. Rankin. Intel 870: A Building Block for Cost-Effective, Scalable Servers. In *IEEE Micro*, pages 36–47, 2002.

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

[32] J. Burch and D. Dill. Automatic Verification of Pipelined Microprocessor Control. In *Intl. Conference on Computer-Aided Verification*, 1994.

[33] M. Cekleov, D. Yen, P. Sindhu, J.-M. Frailong, J. Gastinel, M. Splain, J. Price, G. Beck, B. Liencres, F. Cerauskis, C. Coffin, D. Bassett, D. Broniarczyk, S. Fosth, T. Nguyen, R. Ng, J. Hoel, D. Curry, L. Yuan, R. Lee, A. Kwok, A. Singhal, C. Cheng, G. Dykema, S. York, B. Gunning, B. Jackson, A. Kasuya, D. Angelico, M. Levitt, M. Mothashemi, D. Lemenski, L. Bland, and T. Pham. SPARCcenter 2000: Multiprocessing for the 90's. In *Compcon Spring, Digest of Papers*, pages 345–353, 1993.

[34] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[35] A. Charlesworth. Starfire: Extending the SMP Envelop. In *IEEE Micro*, 1998.

[36] A. Charlesworth. The Sun Fireplane SMP Interconnect in the Sun Fire 3800-6800. In *High Performance Interconnects*, 2001.

[37] X. Chen and G. Gopalakrishnan. BT: A Bounded Transaction Model Checking for Cache Coherence Protocols. Technical report, School of Computing, Univ of Utah, 2006.

[38] X. Chen and G. Gopalakrishnan. Predicate Abstraction for Murphi Using CVC Lite. Technical report, School of Computing, Univ of Utah, 2006.

[39] L. Cheng. *Context-Aware Coherence Protocols for Future Processors*. PhD thesis, School of Computing, University of Utah, 2007.

[40] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. Carter. Interconnect-Aware Coherence Protocols for Chip Multiprocessors. In *Int'l Symp. on Computer Architecture*, 2006.

[41] 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.

[42] 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.

[43] E. M. Clarke. Proving the Correctness of Coroutines without History Variables. In *ACM Southeast Regional Conference*, 1978.

[44] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, 2000.

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

[46] A. Clarlesworth. The Sun Fireplane Interconnect. In *IEEE Micro*.

[47] M. Clint. Program Proving: Coroutines. In *Acta Informatica*, 1973.

[48] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998.

[49] S. Das, D. L. Dill, and S. Park. Experience with Predicate Abstraction. In *Computer Aided Verification*, pages 160–171, 1999.

[50] 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.

[51] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed Explicit-state Model Checking with HSF-SPIN. In *SPIN Workshop*, pages 57–79, 2001.

[52] EE Times, Feb 2007. http://www.eetimes.com/showArticle.jhtml?articleID=197005268.

[53] E. A. Emerson and V. Kahlon. Exact and Efficient Verification of Parameterized Cache Coherence Protocols. In *Correct Hardware Design and Verification Methods (CHARME)*, 2003.

[54] E. A. Emerson and V. Kahlon. Rapid Parameterized Model Checking of Snoopy Cache Protocols. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 144–159, 2003.

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

[56] S. German and G. Janssen. A Tutorial Example of a Cache Memory Protocol and RTL Implementation. Technical report, IBM Research Report, RC23958, 2006.

[57] S. M. German. Formal Design of Cache Memory Protocols in IBM. In *Formal Methods in System Design*, 2003.

[58] S. M. German and G. Janssen. Personal Communication.

[59] 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.

[60] D. I. Good and R. M. Cohen. Principles of Proving Concurrent Programs in Gypsy. In *Principles of Programming Languages*, pages 42–52, 1979.

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

[62] G. Gostin, J.-F. Collard, and K. Collins. The Architecture of the HP Superdome Shared-memory Multiprocessor. In *Int'l Conference on Supercomputing*, pages 239–245, 2005.

[63] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Computer Aided Verification*, 1997.

[64] A. Grbic. *Assessment of Cache Coherence Protocols in Shared-memory Multiprocessors*. PhD thesis, University of Toronto, 2003.

[65] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *High-Performance Computer Architecture (HPCA)*, 1999.

[66] B. T. Hailpern and S. S. Owicki. Modular Verification of Computer Communication Protocols. In *IEEE Transactions on Communications*, 1983.

[67] S. Hazelhurst and C. H. Seger. Symbolic Trajectory Evaluation. In *Formal Hardware Verification – Methods and Systems in Comparison*, 1996.

[68] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[69] M. D. Hill. What is Scalability? In *Computer Architecture News*, 1990.

[70] J. C. Hoe and Arvind. Hardware Synthesis from Term ReWriting Systems. In *International Conference on Very Large Scale Integration: Systems on a Chip*, 1999.

[71] J. H. Howard. Proving Monitors. In *Communications of the ACM*, 1976.

[72] C. N. Ip. *State Reduction Methods for Automatic Formal Verification*. PhD thesis, Stanford University, 1996.

[73] D. James, T. Leonard, J. O'Leary, M. Talupur, and M. R. Tuttle. Personal Communication.

[74] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, 1990.

[75] C. B. Jones. Tentative Steps towards a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems*, pages 596–616, 1983.

[76] R. Kane, P. Manolios, and S. K. Srinivasan. Monolithic Verification of Deep Pipelines with Collapsed Flushing. In *Design, Automation, and Test in Europe*, pages 1234–1239, 2006.

[77] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron Processor for Multprocessor Servers. In *IEEE Micro*, 2003.

[78] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. In *IEEE Micro*, pages 21–29, 2005.

[79] S. Krstić. Parameterized System Verification with Guard Strengthening and Parameter Abstraction. In *Automated Verification of Infinite-State Systems*, 2005.

[80] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[81] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. G. J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Intl. Symposium on Computer Architecture*, pages 302–313, 1994.

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

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

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

[85] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Int'l Symposium on Computer Architecture*, pages 241–251, 1997.

[86] 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.

[87] 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.

[88] T. D. Lovett and R. M. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Int'l symposium on Computer Architecture*, 1996.

[89] D. C. Luckham and J. Vera. An Event-based Architecture Definition Language. In *IEEE Transactions on Software Engineering*, 1995.

[90] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Pringer, 1995.

[91] M. M. K. Martin. *Token Coherence*. PhD thesis, University of Wisconsin-Madison, 2003.

[92] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. In *Int'l Symposium on Computer Architecture*, pages 206–217, 2003.

[93] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Int'l Symposium on Computer Architecture*, 2003.

[94] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth Adaptive Snooping. In *Int'l Symposium on High-Performance Computer Architecture*, pages 251–262, 2002.

[95] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. Martin, and D. A. Wood. Improving Multiple-CMP Systems Using Token Coherence. In *Intl. Symposium on High Performance Computer Architecture*, 2005.

[96] K. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In *Correct Hardware Design and Verification Methods*, pages 219–234, 1999.

[97] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1992.

[98] K. L. McMillan. A Compositional Rule for Hardware Design Refinement. In *Computer Aided Verification*, 1997.

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

[100] 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.

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

[102] K. L. McMillan and J. Schwalbe. Formal Verification of the Gigamx Cache-Consistency Protocol. In *Int'l Symposium on Shared Memory Multiprocessing*, 1991.

[103] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Foundations of Software Engineering (FSE)*, pages 24–32, 1996.

[104] N. Mellergaard and J. Staunstrup. Tutorial on Design Verification with Synchronized Transitions. In *International Conference on Theorem Provers in Circuit Design*, 1994.

[105] J. Misra and K. M. Chandy. Proofs of Networks of Processes. In *IEEE Transactions on Software Engineering*, pages 417–426, 1981.

[106] 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.

[107] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. In *IEEE Transactions on Software Engineering*, 1995.

[108] N. Muralimanohar and R. Balasubramonian. Interconnect Design Considerations for Large NUCA Caches. In *Int'l Symp. on Computer Architecture*, 2007.

[109] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri. Theorem Proving Guided Development of Formal Assertions in a Resource-Constrained Scheduler for High-Level Synthesis. In *Formal Methods in System Design*, 2001.

[110] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2004.

[111] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Int'l Conf. on Parallel Processing (ICPP)*, 1995.

[112] S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–284, 1994.

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

[114] S. Park. *Computer Assisted Analysis of Multiprocessor Memory Systems*. PhD thesis, Stanford University, 1996.

[115] S. Park, S. Das, and D. L. Dill. Automatic Checking of Aggregation Abstractions Through State Enumeration. In *IFIP TC6/WG6.1 Int'l Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, 1997.

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

[117] A. Pnueli, S. Ruah, and L. Zuck. Automatic Decuctive Verification with Invisible Invariants. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 82–97, 2001.

[118] F. Pong and M. Dubois. Verification Techniques for Cache Coherence Protocols. In *ACM Computing Surveys*, pages 82–126, 1997.

[119] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study. In *The First International Euro-Par Conference on Parallel Processing*, 1995.

[120] W. Qin. *Modeling and Description of Embedded Processors for the Development of Software Tools*. PhD thesis, Princeton University, 2004.

[121] W. Qin and S. Malik. A Study of Architecture Description Languages from a Model-based Perspective. In *International Workshop on Microprocessor Test and Verification*, 2005.

[122] A. Rose, S. Swan, J. Pierce, and J. Fernandez. Transaction Level Modeling in SystemC. In *TLM Whilte Paper*, 2005.

[123] H. Rotithor. Postsilicon Validation Methodology for Microprocessors. In *IEEE Design and Test*, 2000.

[124] C. Scheurich and M. Dubois. Correct Memory Operation of Cache-based Multiprocessors. In *Int'l Symposium on Computer Architecture*, pages 234–243, 1987.

[125] X. Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. Technical report, MIT, 1997.

[126] X. Shen, Arvind, and L. Rudolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. In *Int'l Conference on Supercomputing*, 1999.

[127] X. shen, Arvind, and L. Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *Int'l Symposium on Computer Architecture*, 1999.

[128] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2005.

[129] J. Silva and K. Sakallah. GRASP – A New Search Algorithm for Satisfiability. In *Int'l Conference on Computer Aided Design*, pages 220–227, 1996.

[130] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. In *Hot Interconnects*, pages 41–52, 1996.

[131] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. K. Martin, and D. A. ”Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. In *IEEE Transactions on Parallel and Distributed Systems*, 2002.

[132] J. Staunstrup. *A Formal Approach to Hardware Design*. Kluwer Academic Publishers, 1994.

[133] U. Stern and D. L. Dill. Automatic Verification of the SCI Cache Coherence Protocol. In *Correct Hardware Design and Verification Methods*, 1995.

[134] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and Their Supprot by the IEEE Futurebus. In *Int'l Symposium on Computer Architecture*, pages 414–423, 1986.

[135] T. H. Cormen and C. E. Leiserson and R. L. Rivest and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

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

[137] S. Tasiran, R. Alur, R. P. Kurshan, and R. K. Brayton. Verifying Abstractions of Timed Systems. In *Concurrency Theory*, pages 546–562, 1996.

[138] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. In *IBM Journal of Research and Development*, 2002.

[139] M. Thapar and B. Delagi. Stanford Distributed-Directory Protocol. *IEEE Computer*, pages 78–80, 1990.

[140] K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey. Cost-Efficient Block Verification for a UMTS Up-link Chip-rate Coprocessor. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2004.

[141] L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein. System Design Methodology of UltraSPARC-I. In *Design Automation Conference*, 1995.

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