Structural Design and Proof of Hierarchical Cache-Coherence Protocols

by

Joonwon Choi

M.S., Massachusetts Institute of Technology (2016)
B.S., Seoul National University (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2021

Author .....
Department of Electrical Engineering and Computer Science
January 26, 2021

Certified by .....
Adam Chlipala
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Structural Design and Proof of Hierarchical Cache-Coherence Protocols

by

Joonwon Choi

Submitted to the Department of Electrical Engineering and Computer Science
on January 26, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Cache-coherence protocols have been one of the greatest correctness challenges of the hardware world. A memory subsystem usually consists of several caches and the main memory, and a cache-coherence protocol defined in such a system allows multiple memory-access transactions to execute in a distributed manner, across the levels of a cache hierarchy. This source of concurrency is the most challenging part in formal verification of cache coherence.

In this dissertation, we introduce Hemiola, a framework embedded in Coq to design, prove, and synthesize cache-coherence protocols in a structural way. The framework guides the user to design protocols that never experience inconsistent interleavings while handling transactions concurrently. Any protocol designed in Hemiola always satisfies the *serializability* property, allowing a user to prove the protocol assuming that *transactions are executed one-at-a-time*. The proof relies on conditions on the protocol topology and state-change rules, but we have designed a domain-specific protocol language that guides the user to design protocols that satisfy these properties by construction.

The framework also provides a novel way to design and prove invariants by adding predicates to messages in the system, called *predicate messages*. On top of serializability, it is much simpler to prove a predicate message, since it is guaranteed that the predicate is not spuriously broken by other messages.

We used Hemiola to design and prove hierarchical MSI and MESI protocols, in both inclusive and noninclusive variants, as case studies. We also demonstrated that the case-study protocols are indeed hardware-synthesizable, by using a compilation/synthesis toolchain in the framework.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Arvind
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

When I was in the first year of my PhD, I happened to read a very popular comic strip (probably only among graduate students) named "PHD Comics"; this particular comic strip[1] was composed only of four panels, with captions "Impressed! Oppressed. Depressed. and Mostly Pressed," summarizing the life of a graduate student. I wondered if my graduate-student life would be like this cartoon.

After 6.5 years of my PhD, now I recall this comic strip in this dissertation and think about it again. My student life was indeed like this cartoon. I was impressed by brilliant faculty and students who I worked with. I was oppressed by research challenges that I had to get over. I was depressed by a number of rejections I got from conferences and the unknown future. I have felt constant pressure that I should produce good research outcomes.

That being said, I have never thought it was a bad decision that I joined MIT as a graduate student. I am rather grateful that I could do meaningful research in a great environment. I am also grateful to myself for overcoming all the oppression, depression, and pressure that I have had during my entire PhD study.

I would definitely not have been able to complete my PhD successfully without help from the people around me. First of all, I am sincerely grateful to my research advisors Professor Adam Chlipala and Professor Arvind for their invaluable advice. Before joining MIT, I only had a research background in programming languages and compilers, not much in hardware verification. I especially thank Arvind for providing me a fast track to learn modern hardware design and verification. More importantly, I also learned from Arvind that it is really crucial to have a "core idea" in one's research career; I have no choice but to mention the one-rule-at-a-time semantics in Bluespec, for instance, that has dominated all my research projects. My PhD study can be summarized as seeking my own core idea; to a certain degree I am satisfied in that now I have my own philosophy in hardware verification.

I also thank Adam for discussing every small technical detail with me as well as keeping my research projects going in the right direction. During my entire PhD, I think I have been mostly impressed by Adam, astonished by his broad knowledge and insight. Adam has motivated me greatly with the goal that by the end of my graduate study I should know more than Adam about what I have studied for my PhD. (I am still not sure if I have achieved it, though.) One thing I am sure is that I will not be able to work with anyone who is more quick-witted than Adam, and I regard it as my best luck in my PhD.

I would also like to thank the other thesis committee member, Professor Nickolai Zeldovich. I have to say that the basic idea of the Hemiola framework, serializability, is "indirectly" from Nickolai. When I was in the second year, I took a seminar course by Nickolai and studied a research paper that used the commuting-reduction technique, which motivated me to use a similar technique to verify cache-coherence protocols. I thank Nickolai for opening such a wonderful course at that time.

Next, I would like to give my gratitude to the MIT PLV group members. The first

[1] PHD Comics: Professed? http://phdcomics.com/comics.php?f=1672

project that I was mainly involved with was the Kami hardware-verification framework. At that time, I had to learn advanced computer architecture as well as working on the framework. I especially thank Muralidaran Vijayaraghavan for working on Kami with me and giving me intuitions about hardware design and verification in the early phase of development. I also thank the other student author of Kami, Benjamin Sherman, for building basic Coq libraries for the framework. The finite-map library he built has been further developed and used in the Hemiola framework as well.

I would also like to thank the members of the Lightbulb project: Andres Erbsen for the broad knowledge of software and hardware, brilliant intuitions about the project, and most practically comprehensive knowledge of the Coq proof assistant. Also, Samuel Gruetter for great meticulousness in the entire project as well as broad knowledge of program logics. I was very happy to work with these students, convincing myself that theorem proving has a decent possibility to be used for verifying realistic computer systems.

My another big gratitude should go to hardware-verification experts in the PLV group: Clément Pit-Claudel, Stella Lau, and Thomas Bourgeat for listening to the practice talk for my PhD defense and giving me invaluable comments and suggestions. I want to especially thank Clément; I asked quite a lot of questions about Coq, and he kindly discussed with me every time, even though some questions looked a bit stupid. Particularly for the Hemiola framework, Clément explained to me how to reify shallowly embedded programs in a beautiful way. Also, Stella and Thomas for discussing various hardware topics with me and more importantly for checking in on me regularly to see if I am doing fine.

I am thankful for belonging to another research group and also want to give my gratitude to the CSG/Arvind group members. Very big gratitude to Chanwoo Chung, who has been my lunchmate for over 6 years, for teaching me everything about real "hardware" – from basic Bluespec knowledge to how to set up FPGAs on a server. I would not have been able to synthesize any of my Hemiola cache-coherence protocols without his help. I also thank the Riscy-processors developers – Andy Wright, Sizhuo Zhang, and Thomas Bourgeat – for kindly answering my questions about processor designs in Bluespec. Thanks to Shuotao Xu as well for empathizing with the pain and pleasure of parenting.

A couple of people gave me invaluable intuitions for the Hemiola project. I am very thankful to Professor Mengjia Yan for sharing knowledge about noninclusive caches and state-of-the-art cache-coherence designs. Also to Tej Chajed for discussing the relation between serializability and behavioral refinement, which was one of the main headaches building the framework.

There are some people who helped me a lot in terms of my mental health during my PhD journey. The first four years of my graduate study were with music; I was very lucky to receive the MIT Emerson scholarship for private music study and to perform various music performances. I am very grateful to the MIT Music & Theater Arts (MTA) department for giving me an opportunity to earn the scholarship. I want to especially thank Professor Marilyn Roth at New England Conservatory (NEC) for teaching me piano and precious "life lessons" for four years. I also thank all the people who came to my first-and-last solo piano recital at MIT.

Last but not least, I would like to give my special gratitude to my family. I would like to thank my parents, Kija Pyo and Seungil Choi, for having been very supportive for almost 26 years of my student life and having encouraged me to pursue academic careers. My biggest gratitude goes to my wife Jeaeun Shon, for having been with me, overcoming difficulties in our lives, and empathizing with all my oppression, depression, and pressure during my journey. We have learned (mostly when COVID-19 happened from the year 2020) that life is not that easy and does not go as planned most of the times, but I am very sure we will find and follow the path that we are happy enough. Lastly, I am very thankful to our lovely 2-month-old daughter Hannah Choi, for crying less than usual when I write these acknowledgments!

I dedicate this dissertation to my wife and daughter.

Contents

List of Figures

Chapter 1

Introduction: What Makes Hardware Verification Complex

Modern hardware designs are extremely complex. Such a statement has become so hackneyed that it does not draw any attention, but dealing with complex hardware is still a big problem. A single-core processor and a memory system were enough to run computer programs in the early days. Computer architecture has become much more complicated, equipped with multicores, various accelerators (e.g., GPUs, NPUs, etc.), hardware security modules (HSMs), hierarchical memory subsystems, and so on. Such hardware components form a large, complex hardware circuit often called a system on a chip (SoC).

Concurrency is one of the major challenges in hardware design. Concurrent execution has been very natural in hardware in terms of maximizing utility of hardware components. For instance, at a high level, processing units and a memory subsystem run concurrently; e.g., a processor core may execute other instructions after making a memory request while the memory handles it. Looking at each component, a processor core may be pipelined, which is very common these days, implying that (computer-architecture) instructions are executed concurrently as well. The memory subsystem may be composed of several caches and the main memory, and several requests from different cores are handled concurrently in a distributed way throughout the caches and the memory. Concurrency does not mean complete parallelism; for cor-

rectness, some ordering among execution units should be maintained, and managing this constraint is indeed a great challenge.

As hardware has become large with extreme concurrency, hardware verification has become much more challenging as well. Industry has most commonly used testing and bounded model checking (BMC) [18] for quality assurance of hardware circuits, but these techniques are not sound and may miss subtle bugs. Due to this shortcoming, both industry and academia have developed so-called *formal-verification* techniques. At least in hardware verification, there are two mainstream approaches to formally verifying hardware: model checking and theorem proving.

Model checking [17, 20] basically tries to verify a hardware module by exploring all reachable states, checking whether the desired properties hold in those states. The biggest benefit of model checking is that it is fully automated. That being said, state-space explosion is a typical problem in model checking; it often requires too much time and (memory) space to explore all reachable states.

Theorem proving, also called deductive verification, provides a mechanism to generate proofs based on a certain logic. Theorems are proven either manually by a user (called interactive theorem proving), automatically by various solvers (called automated theorem proving), or in a combined way (both interactive and automated). The biggest benefit of theorem proving is that once a theorem is proven, it can be used repeatedly for bigger proofs. Theorem provers usually provide rich logics and thus allow a user to state theorems general enough to be reused. That being said, a theorem prover usually requires a user to understand the target computer system deeply enough to state theorems and proofs in detail.

Problem statement

Among various hardware-verification challenges and formal-verification methodologies, this dissertation specifically deals with cache-coherence protocols with the Coq proof assistant [23], an interactive theorem prover.

Cache-coherence protocols have been regarded as one of the greatest correctness challenges of the hardware world. As per typical distributed-protocol verification, the

main challenge comes while dealing with interleavings among transactions. In cache-coherence protocols, a request from a processor may require multiple caches to change their local states, and we call a sequence of such state transitions a *transaction*. Thus it is natural that different cache objects concurrently handle different requests, which makes the transactions *interleaved*. Interleavings make the verification much more difficult, by significantly increasing the number of states to be explored.

Research on model checking has produced sound techniques to overcome state-space explosion by interleavings. One of the core techniques is to establish *noninterference* lemmas [47, 51, 16, 74, 58, 69], which ensure that concurrent executions of transactions are not spuriously interleaved. In addition to this resolution, model checkers have supported *parameterization* to verify cache-coherence protocols with unknown numbers of cores [79, 78, 3]. In order to cover hierarchical protocols, recent approaches [13, 14, 45, 46, 60] also employed *modularity* to verify protocols per hierarchy level.

While the same concern exists in theorem proving (by requiring additional proof effort to deal with interleavings), it also has employed similar techniques. Parameterization is a natural fit for theorem proving, thanks to rich logics, making it possible to prove a cache-coherence protocol with an arbitrary tree shape given as a parameter [76]. Modularity has been harnessed as well, e.g., with assume-guarantee reasoning [48].

The established style of noninterference reasoning, though, has an inherent weakness: it requires explicit lemma statements per-protocol. The lemmas are often described explicitly by the user (guided by intuition) or discovered automatically (typically guided by counterexamples). Another issue is that even though these lemmas are stated and proven, it is hard to be sure if the lemmas are enough to resolve all possible interleavings in the system.

Use of modularity has also imposed significant restrictions in protocol design, in order to obtain clear module boundaries among caches. A concrete limitation of modular design and verification comes when trying to verify cache-coherence protocols that are not *inclusive*. Noninclusive caches have been in common industrial use for

a decade: AMD Opteron servers are known to use exclusive caches [36], and Intel Skylake-X processors use noninclusive caches [2, 82, 77].

We would like to avoid all of these weaknesses via a new approach doing mechanized proof in an expressive logic, stating and proving once-and-for-all a general property and using it multiple times. Particularly, in proving cache-coherence protocols, we want to formalize *the most-general form of noninterference*, find minimal and abstract conditions *that are not coupled to any specific protocols* but still imply noninterference, state the implication as a theorem, prove it once, and use it repeatedly for various cache-coherence protocols.

The abstract conditions that imply noninterference can be found from the intuitions that protocol designers already have. In order to describe a case where the protocol deals with two different requests passing through the same cache, for instance, a designer often says "this cache acts like a *serialization point* so the two transactions can safely *interleave* with each other." While the notions of interleaving and serialization are already pervasive, we have found no past work trying to find a minimal set of conditions that ensure transactions are always serialized, formalizing them in a reusable framework.

We go one step further: rather than requiring the user to prove the conditions per-protocol, we establish a framework that guides the user to design protocols that automatically satisfy the conditions, without worrying about possible complex interleavings among different transactions. The framework does not require the user to write any noninterference properties or to design the protocol in a modular manner.

In this paper, we introduce the Hemiola framework, embedded in the Coq proof assistant, which eliminates the burden of designing and proving correct interleavings in cache-coherence protocols. Specifically, Hemiola formalizes a set of message-passing distributed protocols with tree hierarchies and particular topologies of channels between nodes, with associated locks. Hemiola exposes a domain-specific language of protocols, such that any expressible protocol is guaranteed to exhibit "general" noninterference, which will be called *serializability* and formalized later in the dissertation.

Furthermore, Hemiola provides a novel invariant proof method that only requires

consideration of execution histories that do not interleave memory accesses – so that almost all formal reasoning about concurrency is already done by the framework, not proofs of individual protocols.

To demonstrate usability of our framework, we provide complete correctness proofs for hierarchical cache-coherence protocols, in both inclusive and noninclusive variants, as case studies. We also demonstrate that the case-study protocols are indeed hardware-synthesizable, by using a compilation/synthesis toolchain in Hemiola.

All the code and proofs of the framework and the case studies are available as open source at the following link:

https://github.com/mit-plv/hemiola

Contributions

To sum up, the contributions of this dissertation include the following:

- We discover a minimal set of topology and lock conditions that ensures *serializability*, extracted from usual cache-coherence protocol designs. We then identify a domain-specific protocol language, where every protocol defined in this language ensures serializability by construction, backed up with mechanized Coq proof (chapter 5). Lastly, we formalize how serializability helps prove correctness (trace refinement), by using the novel notion of *predicate messages* in distributed protocols.

- Using the protocol language, we provide the complete correctness proofs of three hierarchical cache-coherence protocols: inclusive and noninclusive MSI protocols and a noninclusive MESI protocol (chapter 8). Ours are the first complete mechanized proofs for various hierarchical "noninclusive" cache-coherence protocols, and as a bonus they share a large segment of reusable proofs (including with the inclusive variant).

- We provide a toolchain to compile Hemiola protocols to hardware implementations and to synthesize them to run on FPGAs. We also evaluate our case-study

protocols, in order to demonstrate that they indeed run on hardware with reasonable performance.

Outline

Before introducing our proposed techniques to verify cache-coherence protocols, we will provide some background in chapter 2 to understand conventional protocol designs and typical correctness challenges. After the background chapter, this dissertation is composed of three parts.

The first part presents our underlying theory to reason about cache-coherence protocols. chapter 3 introduces protocol transition systems as underlying state-transition systems to describe behaviors of cache-coherent memory subsystems. On top of a protocol transition system, we provide the formal definition of serializability in chapter 4. This chapter also demonstrates how serializability helps state and prove invariants of a given system.

The second part introduces Hemiola, a framework embedded in Coq to design and verify cache-coherence protocols more structurally, by using the serializability guarantee. chapter 5 first presents the Hemiola domain-specific language for easier protocol design. chapter 6 describes the biggest contribution of the framework, a general once-and-for-all property claiming that any protocol defined with the DSL obtains serializability for free. We then discuss the first series of related work in chapter 7, focusing on how previous approaches dealt with interleavings among transactions.

The last part provides our case studies to demonstrate practicality of the framework. We present the designs and complete correctness proofs of hierarchical MSI and MESI protocols on top of Hemiola in chapter 8. chapter 9 presents the protocol compiler, which compiles a protocol described in Hemiola to a corresponding hardware implementation and discusses the correctness of the compiler. We then discuss the other series of related work in chapter 10, exploring some previous approaches to verifying cache-coherence protocols.

Chapter 2

Background and Overview

Before introducing our proposed method to design and prove cache-coherence protocols more structurally, in this chapter we provide some background to understand conventional cache-coherence protocol designs and typical correctness challenges. We first introduce a very simple cache-coherence protocol to explain the purpose of the protocol, basic design, and a number of nontrivial corner cases that require careful design. After that, we explore the design space of cache-coherence protocols for better understanding of the case-study protocols that we will provide in chapter 8.

2.1 Cache-Coherence Protocols In a Nutshell

In this section, we provide a simple yet motivating example to explain the typical challenges. For simplicity, throughout the section, we will consider a protocol protecting only *a single memory location*. We will see it is still nontrivial to design and verify a correct protocol.

The overall goal of cache coherence is, as the name suggests, to preserve coherence among multiple candidate values in a memory subsystem. In other words, if the system is coherent, then it should behave like an atomic memory. This behavior inclusion is formally defined as *refinement*, which will be introduced in section 3.2.2. The correctness of a cache-coherence protocol is sometimes shown by proving representative invariants, not talking about the relation to the desired spec. The single-

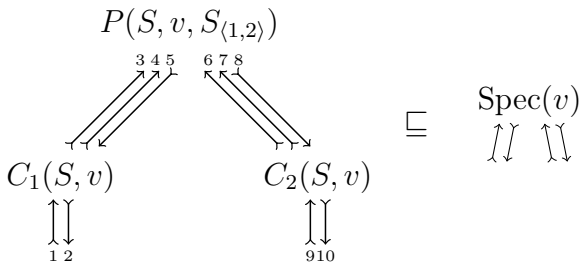


Figure 2-1: A simple MSI directory protocol and its spec

writer/multiple-reader (SWMR) invariant and the data-value invariant [55] are classic choices; in proving the refinement of a protocol, these invariants (or similar ones) must be proven.

Figure 2-1 shows caches and communication channels for a simple directory-based MSI protocol (LHS of $\sqsubseteq$). Since we deal with only a single memory location, the specification (RHS of $\sqsubseteq$) is a single-value (v) container with atomic read and write. There are three caches (P, C_1, and C_2) in the implementation, and each of them has its own status (M, S, or I) and data (v). The status of a cache represents a permission on its local replica. In this MSI protocol, an object can read/write the data with the M ("modified") status, only read with S ("shared"), and cannot read/write with I ("invalid"). The parent P additionally has a data structure called a *directory* to track the statuses of the children. For example, a directory might be $S_{\langle 1,2 \rangle}$, meaning that both C_1 and C_2 have S status, in some logical snapshot of state.

Objects communicate through ordered channels, shown as ($\rightarrowtail$) in the figure, where each has a unique index (shown as a natural number in the figure). C_1 and C_2 have channels to receive and respond to external requests (from processor cores). There are three types of channels between a parent and a child: a single channel for parent-to-child messages and two channels for child-to-parent requests and responses, respectively. It is natural to wonder why two separate channels are required from a child to a parent; we will see the reason very soon. Note that channels are depicted in a logical way; the actual hardware implementation may use different hardware components (e.g., finite-capacity FIFOs or buses) that can simulate ordered channels.

Figure 2-2 depicts some example state-transition cases depending on statuses of

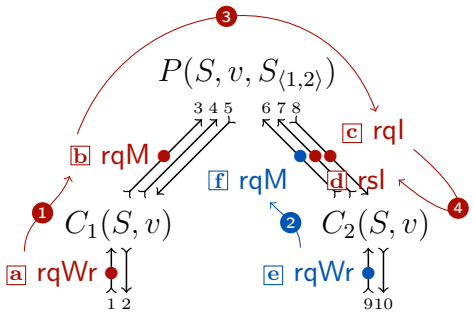


Figure 2-2: Rule-execution cases in the simple MSI protocol

the caches. All the caches run concurrently, repeatedly executing *rules* that define local state transitions. A rule may take some messages from input channels, perform a state transition, and put messages in output channels. A rule may also have a precondition, blocking use of that rule when the precondition does not hold. ❶ shows the case where a child C_1 takes an external request (a rqWr) to write data, but it does not have M status and thus further requests to the parent (b rqM) to get the permission. At this moment, in many cache-coherence protocol designs, C_1 changes its status to a *transient state* SM to record its current status (S) and the next expected status (M). This transient state also functions as a *lock* not to allow any further external requests for the value.

Due to the concurrent execution of the caches, we might have another rule executed at the same time. ❷ is executed concurrently with ❶, where C_2 also takes an external request (e rqWr) with the same purpose, thus requests f rqM to the parent as well. Since ❶ and ❷ happened at the same time, now the parent P needs to decide which request to deal with. Suppose that it decided to handle b rqM first.

❸ presents the next execution by P, taking the input message b rqM and making an *invalidation* request (c rql) to the other child C_2 to change its status to I. This invalidation request is required, since when a child has M the others should not be able to read/write the data. The parent, at this moment, changes its directory status to a transient state (named S^D in some textbooks) to disallow any other requests from the children. For instance, the parent should not handle f rqM, since otherwise it will handle two "rqM"s simultaneously, which might lead to an incoherent state –

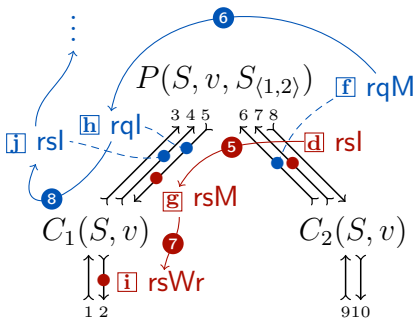


Figure 2-3: Rule-execution cases, continued

two M statuses in the caches.

Lastly, ❹ shows the case that C_2 handles the invalidation request (c rql). A number of corner cases should be handled carefully:

- Since C_2 requested f rqM, it has a transient state SM when c rql arrives. It should still be able to handle this invalidation request even in the transient state (while not allowing any external requests). In this case C_2 accepts c rql and changes its transient state to IM. We see that transient states should be fine-grained enough to distinguish which requests to handle.

- Due to the existence of f rqM, if we had a single channel from a child to a parent, a deadlock would occur. P cannot take f rqM since it has been locked (in a transient state) after making an invalidation request. It cannot take d rsl as well since the response is not the first one of the ordered channel. This case shows the necessity of having multiple channels between a child and a parent.

A so-called three-channel system has been widely used and regarded as a good choice to make the design correct and live [76, 75]. While there are other possible correct topology and network settings, the cases shown in Figure 2-2 at least demonstrate that it is nontrivial to construct one of them.

Figure 2-3 presents some additional rule-execution cases right after ❹ in Figure 2-2. After ❹ responds with d rsl, now P can take it to respond back to C_1. ❺ presents this step, taking d rsl and responding back to C_1 with g rsM. At this moment P changes its status to $P(I, \cdot, M_{(1)})$ (from the transient state S^D) to record that it no

longer has the coherent value and just granted C_1 the M status. Since the transient state also functions like a lock, this status change can be also regarded as a lock release, so that P can handle some other requests.

❻ is the next transition step by P to accept a new request. [f] rqM has been waiting for the transient state of P to be released, and ❻ finally takes it and sends an invalidation request to C_1.

Looking at the message channel 5, the one for parent-to-child messages (from P to C_1), there are two messages, [g] rsM and [h] rqI, residing in the channel. A single parent-to-child channel affects the correctness of the protocol in this case. If we had two separated downward channels, one for requests and the other for responses, C_1 could handle [h] rqI first, change its status to I, handle [g] rsM later, and change its status to M again. Since it sent the invalidation response to the parent, eventually C_2 will also get the M status, which leads to an incoherent state. Therefore, it is crucial to have a single parent-to-child channel so that [g] rsM *blocks* [h] rqI to be handled first. After C_1 takes [g] rsM and responds back to the processor core with [i] rsWr, presented as ❼, it can subsequently take the invalidation request [h] rqI and responds with [j] rsI, as shown in ❽.

As examined in Figure 2-2 and Figure 2-3, in a cache-coherence protocol, transient states and network channels play a crucial role in making *interleavings* correct. Regarding the sequence of rule executions (in red) [❶; ❸; ❹] as an execution flow in Figure 2-2 – we will call it a *transaction* in later sections – to handle the original external request [a] rqWr, we see that no further executions could happen after ❷, which is for the other request [e] rqWr. As explained above case-by-case, proper transient states and network channels block [f] rqM from further processing. Similarly, in Figure 2-3, the execution flow [❺; ❸] was not spuriously affected by ❻ or ❽, thanks to the correct channel setting.

Throughout the dissertation, we will use this example to convey various intuitions in formal definitions and proofs, referred to as the "interleaving example."

2.2 Overview: the Hemiola Framework and Its Formal Guarantee

In the previous section, we confirmed that proper locking (by transient states) and topology are crucial for designing a correct cache-coherence protocol. If those ingredients are that essential, can we craft a domain-specific language where only conformant protocols are expressible? That is exactly what the Hemiola framework provides: it provides *rule templates*, explained in section 5.3 in more detail, that employ proven-safe topologies and network structures and automatically set associated locks, so designers can design protocols without worrying about corner cases due to interleavings.

Before diving into the details of the theory, in this section, we provide a brief overview of the Hemiola framework by redefining the example protocol shown in section 2.1 using rule templates. Specifically, we will show the use of transient states discussed in section 2.1 directly in conventional descriptions and compare them with the alternative locking mechanism exposed by Hemiola. After the comparison, we will provide high-level intuitions about the formal meaning of safe interleaving and how the rule templates ensure this property.

The Hemiola rule templates

	Conventional description	*In Hemiola*

(1)	`rule :=`
	`msgIn = extToC1.first;`
	`extToC1.deq();`
	`assert (msgIn.id == rqWr);`
		`rule.rquu`
		`:from ext`
		`:me C1`
		`:accepts rqWr`

(2)	`assert (!mshr.in_transition);` `assert (line.status ≤ msiS);`	`:requires` `(fun line msgIn => line#[status] ≤ msiS)`

(3)	`mshr.in_transition <= msiSM;`
	`c1ToPRq.enq({id: rqM, val: 0});`
	`endrule`
		`:transition`
		`(fun line msgIn => <| rqM, 0 |>).`

Figure 2-4: A child requesting M from its parent

Figure 2-4 presents a conventional rule description (in pseudocode) used to request

the M status from a parent on the left side of the figure. This rule definition matches the rules ❶ and ❷ in the interleaving example shown in Figure 2-2. This figure also presents a corresponding rule in Hemiola using one of the rule templates. There are largely three benefits of using the Hemiola rule templates; we explain them by matching the number annotations in the figure.

(1) Input/output patterns: in a conventional description, a user has to find and designate a FIFO module to dequeue an input message and to check the purpose of the message by looking at its message ID. In Hemiola, each rule template is restricted in terms of input/output patterns; for instance, the `rquu` rule template (where `rquu` stands for request-up-up) always takes an input request from one of its children (`:from` the `external` world in this case) and generates an output request to the parent of `:me C1`. We can also annotate that this rule only `:accepts` an input message with an ID `rqWr`.

(2) In-transition checks: a conventional description uses a miss status holding register (`mshr`) to record whether a line is `in_transition` or not. Hemiola employs an abstract locking mechanism, and each rule template automatically sets/releases the locks. Hence, a user does not need to describe any in-transition checks or transient states manually, which is one of the major headaches in designing a protocol. More specifically, Hemiola provides two kinds of locks: *an uplock and a downlock*. The `rquu` rule template automatically sets an uplock, since it makes an upward request (to the parent). The uplock is released when handling the corresponding response from the parent. We will see some other use cases of the locks in the other rule templates.

(3) Use of transient states: in a conventional description, a user should manually update `mshr` to set a proper transient state and find a FIFO to enqueue an output message. Neither is required in Hemiola; in this `rquu` rule template we just need to construct an output message. All the rule templates assume the three-channel system introduced in section 2.1; e.g., the `rquu` rule template enqueues an output message to the upward-request channel.

	Conventional description	*In Hemiola*

(1)	`rule :=`
	`msgIn = c1ToPRq.first;`
	`c1ToPRq.deq();`
	`assert (msgIn.id == rqM);`
		`rule.rqud`
		`:from C1`
		`:me P`
		`:accepts rqM`

(2)	`assert (!mshr.in_transition);` `assert (line.dir.st == msiS);`	`:requires` `(fun line msgIn => line#[dir].st == msiS)`

(3)	`mshr.in_transition <= msiSD;`
	`pToC2.enq({id: rqI, val: 0});`
	`endrule`
		`:transition`
		`(fun line msgIn => (C2, <| rqI, 0 |>).`

Figure 2-5: A parent requesting invalidation from the other child sharer

Figure 2-5 shows another rule pattern to make a further request, used in the parent object to send an invalidation request to the other sharer. This rule matches the rules ❸ and ❻ in the interleaving example shown in Figure 2-2 and Figure 2-3, respectively. We have the same three benefits in the suggested rule template on the right side of the figure: automatically correct input/output channels, no in-transition checks, and no need to consider transient states. The only difference from the `rquu` template is that this one generates further requests to children, not to the parent. (`rqud` here stands for request-up-down.) It automatically sets a downlock, since it makes downward requests to children. Likewise, the downlock is released when handling the corresponding responses from the children.

	Conventional description	*In Hemiola*

(1)	`rule :=`
	`msgIn = pToC2.first;`
	`pToC2.deq();`
	`assert (msgIn.id == rqI);`
		`rule.immu`
		`:me C2`
		`:accepts rqI`

(2)	`assert (mshr.in_transition` `== msiSM);`	`:requires` `(fun line _ _ => line#[status] == msiS)`

(3)	`mshr.in_transition <= msiIM;`
	`line.status <= msiI;`
	`c2ToPRs.enq({id: rsI, val: 0});`
	`endrule`
		`:transition`
		`(fun line msgIn =>`
		`(line +#[status ← msiI], <| rsI, 0 |>).`

Figure 2-6: A child responding to invalidation immediately even in transition

The next pattern, shown in Figure 2-6, is used to respond immediately to a request from a parent. The rules ❹ and ❽ in the interleaving example use this pattern to handle an invalidation request. As already discussed in section 2.1, the child object

should be able to handle this invalidation request from its parent even though it is in transition. In a conventional description, it is required to change the transient state properly (from `msiSM` to `msiIM` in this example). In Hemiola, as seen in the figure, it is not required to manage such transient states; a user can just use the `immu` rule template (where `immu` stands for immediate-up). The locking mechanism using an uplock and a downlock is still enough to implement this case properly by prioritizing downward requests over the uplock.

	Conventional description	*In Hemiola*

(1)	`rule :=`
	`msgIn = c2ToPRs.first;`
	`c2ToPRs.deq();`
	`assert (msgIn.id == rsI);`
		`rule.rsud`
		`:accepts rsI`

(2)	`assert (mshr.in_transition);` `assert (mshr.rq.id == rqM);`	`:holding rqM` `:requires (fun _ _ _ => True)`

(3)	`mshr.in_transition <= None;`
	`line.dir.st <= msiM;`
	`line.dir.excl <= mshr.rsbTo;`
	`line.status <= msiI;`
	`pToC1.enq({id: rsM, val: 0});`
	`endrule`
		`:transition`
		`(fun line msgIn rq rsbTo =>`
		`(line +#[status ← msiI]`
		`+#[dir ← setDirM rsbTo],`
		`<| rsM, 0 |>).`

Figure 2-7: A parent responding back to the child requestor to grant M

In a usual cache-coherence protocol, when a cache object gets a response message from another object, it must have previously sent a corresponding request to that object. Figure 2-7 describes this case, where a parent cache object handles an invalidation response from a child, like the rule ❺ in the interleaving example. In a conventional description, we need to check that the line is currently `in_transition` and the original request held in the MSHR matches the input response. It is also required to release the MSHR slot by setting `in_transition` to none. As expected, in Hemiola we do not need to manage transient states at all; the `rsud` rule template (standing for response-up-down) just requires an additional annotation `:holding rqM` to declare that the rule handles a response message where the ID of the original request message is `rqM`. This rule template automatically releases the downlock previously acquired when sending a downward invalidation request.

Lastly, Figure 2-8 deals with the case where a child finally gets a response from its

	Conventional description	*In Hemiola*
(1)	`rule := msgIn = pToC1.first; pToC1.deq(); assert (msgIn.id == rsM);`	`rule.rsdd :accepts rsM`
(2)	`assert (mshr.in_transition); assert (mshr.rq.id == rqWr);`	`:holding rqWr :requires (fun _ _ _ => True)`
(3)	`mshr.in_transition <= None; line.status <= msiM; line.value <= mshr.rq.value; c1ToExt.enq({id: rsWr, val: 0}); endrule`	`:transition (fun line msgIn rq rsbTo => (line +#[status ← msiM] +#[value ← rq.(msg_value)], <| rsWr, 0 |>).`

Figure 2-8: A child responding to a write request

parent to update its status to M, like a rule ❼ in the interleaving example. The `rsdd` rule template (standing for response-down-down) is very similar to the `rsud` template but has a difference that it automatically releases the uplock that was previously acquired when sending the `rqM` request.

We will explore much more detail of the locking mechanism and the rule templates provided by Hemiola in chapter 5.

The serializability guarantee by Hemiola

We have emphasized that it is easier to describe cache-coherence protocols with the Hemiola rule templates, but it does not mean a lot if the rule templates are just an alternative representation of transient states. The biggest contribution of the Hemiola framework is that using the rule templates (implicitly with the locking mechanism) already implies safe interleavings among transactions.

Hemiola provides the formal definition of serializability as a criterion for safe interleavings. Informally, that property claims that for any legal history of a system we can always find a sequential history that reaches the same resulting state, where each transaction is performed atomically (without any interleavings).

Figure 2-9 presents a history brought from the example in section 2.1, where two histories (in red and blue) are interleaved to form a history h. We can find a corresponding sequential history h_{seq}, formed by concatenating two transactions in

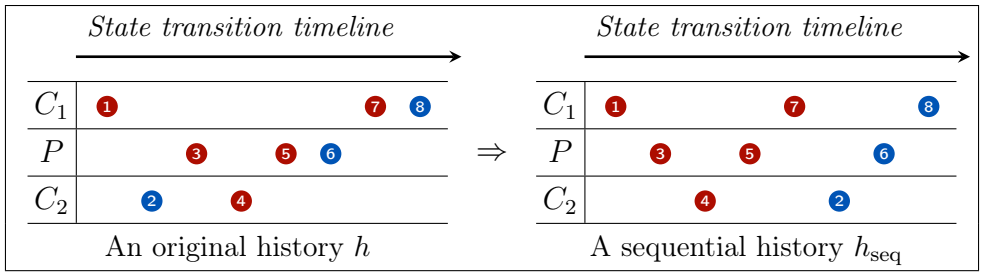


Figure 2-9: An example of a serializable history

red and blue. Following the rule executions in h_{seq}, we see that it indeed reaches the same state as the original history h does. We will introduce all the required formalization to define serializability in chapter 4.

The biggest contribution of the Hemiola framework is that use of the rule templates implies serializability. Figure 2-9 just presents a specific history and its corresponding sequential history. Hemiola provides a theorem guaranteeing that we can find a sequential history for any legal history in any protocol designed with the rule templates. In other words, the theorem says that good request/response patterns with proper locking mechanisms are enough to prove serializability.

How does this nontrivial theorem hold? The highest-level intuition behind this theorem is the *mutual exclusion* by uplocks and downlocks. For instance, when an object is uplocked, it cannot generate any further upward requests, indicating that only a single transaction can (upwardly) pass through this object by acquiring an uplock. Likewise, when an object is downlocked, it cannot generate any further downward requests, meaning that a single transaction can (downwardly) pass through this object by acquiring a downlock.

Mutual exclusion by a lock is one typical way to make an execution of a transaction (thread) atomic. By repeatedly applying the atomicity guarantee by each lock, we can derive a sequential history from a given interleaved one. Note again that the state transitions by a transaction (other than locking) are irrelevant to the locking mechanism, implying that the serializability proof indeed does not require any specific conditions on a protocol.

All the required proof techniques to reason about these mutual exclusions and the

actual serializability proof will be introduced in chapter 6. We will also present our suggested methodology to exploit serializability for the verification of cache-coherence protocols in section 4.3.

2.3 Design Space of Cache-Coherence Protocols

In section 2.1 and section 2.2, we examined a very specific protocol design. There are different ways to design a cache-coherence protocol, and each design decision affects performance and correctness of the protocol. In this section, we explore the major design space of conventional cache-coherence protocols.

Snooping vs. Directory

A typical classification of a cache-coherence protocol is whether the protocol employs a designated data structure to keep track of the statuses of child caches. There are two representative classes: *snooping* and *directory* protocols.

A snooping protocol does not use any data structure to monitor the statuses of child caches. Instead, when the parent communicates with its children, it broadcasts a message to them. Each child cache decides whether to respond to the message from the parent or just to ignore it, depending on its status. For example, suppose a scenario, already mentioned in section 2.1, that the parent wants to make an invalidation request to downgrade the status of the child who has the M status to I. Since the parent cache does not have any information which child currently has M, it has no choice but to broadcast invalidation-request messages to all of the children. The child cache with M will respond to the parent after invalidating its status, while the other caches will just ignore the request. In actual hardware implementations, snooping is usually implemented with an ordered wire bus shared by the parent and children.

A directory protocol, as its name says, uses a directory structure to record which child has which status. The example presented in section 2.1 is a directory protocol; we see that the parent P additionally has a directory data structure. Considering the same invalidation scenario, since the parent knows which child has the M status

exactly, it can make an invalidation request just to that child.

A snooping protocol is known to be easier to implement than a directory protocol, since an ordered bus enforces less interleavings among transactions. That being said, the snooping protocol is also known not to be scalable, since the cost of broadcasting significantly increases when more child caches are involved. A directory protocol, on the other hand, is known to be scalable and matches hierarchical protocols well, but design and verification are relatively more difficult than for the snooping protocol.

This dissertation only deals with directory protocols due to their inherent difficulties. We will build a framework that can ease the burden of designing and verifying directory protocols and demonstrate the practicality of the framework with various hierarchical directory protocols as case studies.

Write-update vs. Write-invalidate

A cache-coherence protocol can be also classified by the patterns of memory writes. Among various memory-write patterns, we classify protocols by when each legal write to a cache is propagated to the other caches. Then we have two classes: *write-update* and *write-invalidate* protocols.

In a write-update protocol, if a cache wants to write to a line, it requests to update the lines in the other caches first and updates its line after the updates by the others. This protocol is beneficial when a cache writes to a line and the other caches want to read it right after the write.

In a write-invalidate protocol, if a cache wants to write to a line, it first requests to invalidate the lines in the other caches so that they cannot read or write the line. Once all the other caches are invalidated, it can write a new value. The write-invalidate policy is used more generally in practical cache-coherence protocols, since the write-update policy requires to propagate new values for each update and is not efficient when some other caches actually do not need to read the up-to-date value anymore.

Write-through vs. Write-back

Another criterion about memory writes is when a write to a cache is applied to the main memory; it provides two classes as well: *write-through* and *write-back* protocols.

In a write-through protocol, when a cache writes to a line (already with enough permission), it further requests a write to the main memory as well. This protocol is easier to design, since the up-to-date value can always be found in the main memory.

On the other hand, in a write-back protocol, a cache does not request to the main memory to send the up-to-date value when writing to a line. The value is eventually written back to the main memory when the line is evicted from the cache side. Write-back protocols are harder to implement than write-through ones, since they should provide a way to find the up-to-date value through all the caches, e.g., by managing directories that point to caches with correct values. That being said, almost all practical cache-coherence protocols use the write-back policy to avoid unnecessary writeback to the main memory.

All the case-study protocols in this dissertation, which will be introduced in chapter 8, follow the write-invalidate and write-back policies.

Inclusive vs. Noninclusive vs. Exclusive

The protocol we introduced in section 2.1 is a flat (nonhierarchical) protocol in that child caches (C_1 and C_2) communicate directly with the parent P as the main memory with a directory. In other words, there are no intermediate caches between the lowest-level ones (called L1 caches) and the main memory. There is an essential tradeoff between caches and the memory: caches have much faster latency than the memory but lower hit rate. In order to mitigate hit rate of a cache, hierarchical cache-coherence protocols have been developed and used. In a hierarchical protocol, a number of L1 caches are clustered to have an L2 cache as a parent, and so on. A higher-level cache usually has a higher latency than lower-level caches but has a better hit rate due to its increased size.

Hierarchical cache-coherence protocols can be classified by so-called *cache-inclusion*

policies. Especially when designing a protocol with directories, it is practically easier to design caches to require that every line in the value cache has its corresponding directory entry in the directory cache, and vice versa. It is indeed easier to design with such a policy, since it allows to combine an information cache (one containing line statuses, etc.) and a directory cache into a single cache, which is more space-efficient by not having to store tags in each cache. This policy, in other words, means that whenever a line is in a child cache there is a corresponding line in the parent cache as well. We therefore call it *inclusive*, meaning that the parent cache includes the lines in child caches.

Inclusive caches have a benefit that certain requests from the parent can be answered immediately just by searching a directory status. For example, when a cache gets an invalidation request from the parent and the line entry does not exist in the directory, it does not need to forward the request to the children for complete invalidation, since by the inclusion policy absence of a directory status already implies that the line does not exist in any child caches. That said, inclusive caches have a drawback that some value entries are wasted; for example, when a directory status is M, meaning that a child has the up-to-date value, the parent still holds a stale value inside the value cache.

A *noninclusive* cache – also called a noninclusive nonexclusive (NINE) cache – does not require such inclusion. Since it is not inclusive, it has less chance to waste value lines. That said, the biggest drawback comes when a directory-status entry does not exist in the directory cache; the absence of a directory status does not imply that child caches do not have the line anymore. The same issue occurs in a snooping protocol as well, e.g., a cache should always broadcast an invalidation request to all the child caches in order to handle the one from the parent.

In order to resolve these issues, a number of practical cache-coherence protocols use noninclusive value caches and inclusive directories, called NCID [82], e.g., Intel Skylake-X processors are known to use the NCID structure [2, 77]. By having noninclusive value caches we have less chance to waste value lines, and by having inclusive directories we do not always need to visit child caches for invalidation.

As opposed to inclusive caches, an *exclusive* cache intends to maximize the utility between child and the parent caches, by requiring that the lines of a child and the parent are always disjoint to each other. AMD Opteron servers are known to use exclusive caches [36]. A similar optimization to NCID is possible in exclusive caches, by managing directories as inclusive while the value caches are exclusive.

Remarks

The classifications introduced so far are orthogonal to each other, i.e., every combination of the policies gives different protocol design. For instance, the case-study protocols that will be introduced in chapter 8 are directory-based and follow the write-invalidate and write-back policies. One of the protocols is inclusive, whereas the others are noninclusive.

Although not every combination of the policies is covered by the case studies, we would like to claim that our proposed framework Hemiola is general enough to design and verify cache-coherence protocols with any policies mentioned above, since the domain-specific language and verification methodology provided by the framework are not coupled to any specific protocol policies.

Part I

Protocol Transition Systems and Serializability

Chapter 3

Protocol Transition Systems

The very first task to verify cache-coherence protocols in a theorem prover is to define an underlying state-transition system formally. Formalization of such a system typically consists of its formal syntax and semantics. Additionally, if we want to verify a program defined in the system, a correctness criterion should be established as well. In this chapter, we provide the formal definition of state-transition systems for cache-coherence protocols, called *protocol transition systems*.

3.1 Cache-Coherence Protocols as Message-Passing Systems

In a cache-coherence protocol, cache objects in a system communicate with each other via *messages*. Such communication is asynchronous in that a sender (a cache object) sends a message to a channel first, and a receiver handles it later. Here the notion of a channel is logical, but the actual hardware implementation may use various hardware components (e.g., finite-capacity FIFOs or buses) that can simulate it. The hardware implementation of communication channels will be explained in detail later in chapter 9.

The correctness proof of a cache-coherence protocol heavily relies on the formalization of a message-passing system. An incorrect formalization can leave a protocol

unprovable. It is also a part of the trusted computing base (TCB); one must read through the formalization and be convinced that it is fair. The formalization of a message-passing system varies by its purpose. For instance, in order to reason about software distributed protocols, one may want to add Byzantine fault models [42] to the formalization.

A cache-coherence protocol also requires its specific assumptions on top of a message-passing system. First of all, communication channels should be ordered; practical cache-coherence protocols indeed use ordered channels, and the correctness of a protocol usually depends on this constraint. Also, each channel cannot be accessed twice in a single state-transition cycle, e.g., one cannot push two different messages to a channel in the same cycle. This constraint is also due to the practical implementations of an ordered channel; considering a channel as a black-box circuit, there is usually a single "wire" to push a new element to the channel, thus accessing it twice (through the wire) is not possible.

Last but foremost, a conventional message-passing system assumes that it is fair to reason a single state transition (by a single object) at a time. In other words, while multiple objects make their local state transitions by consuming and generating messages in actual execution, it is still fine to assume that a state-transition step is made by a single object. This assumption is powerful, especially in verification, since without it there would be an exponential number of cases to consider for a state transition. We certainly want to utilize the assumption in reasoning about cache-coherence protocols. Fortunately, a convenient concept has been developed in rule-based hardware description languages (RHDLs) such as Bluespec SystemVerilog [56], called "one-rule-at-a-time semantics." Our cache-coherence protocol descriptions will be rule-based, and thus our formalization of the underlying system will assume that rules are executed atomically, even when multiple rules are executed at the same time. It is worth noting that the one-rule-at-a-time semantics has been used in designing/verifying hardware components [76, 24, 25] and formalized [9, 15, 8].

ID	$\mathsf{id} \in \mathbb{I}$
Value	$\mathsf{val} \in \mathbb{V}$
Message	$m ::= \langle \mathsf{ty}, \mathsf{id}, \mathsf{val} \rangle \in \mathbb{M} \triangleq \mathbb{B} * \mathbb{I} * \mathbb{V}$
Index	$i \in \mathbb{I}$ (for channels, objects, etc.)
Channel Index & Message	$im ::= (i, m) \in \mathbb{IM} \triangleq \mathbb{I} * \mathbb{M}$
Object state	$o \in \mathbb{O}$
Rule precondition	$\mathsf{prec} \in \mathbb{O} \times \overline{\mathbb{IM}} \to \mathbb{P}$
Rule transition	$\mathsf{trs} \in \mathbb{O} \times \overline{\mathbb{IM}} \to \mathbb{O} \times \overline{\mathbb{IM}}$
Rule	$r ::= \langle i, \mathsf{prec}, \mathsf{trs} \rangle$
Object	$O ::= \langle i, o_{\mathsf{init}}, \overline{r} \rangle$
System	$S ::= \langle \overline{O}, \overline{i_{\mathrm{in}}}, \overline{i_{\mathrm{rq}}}, \overline{i_{\mathrm{rs}}} \rangle$

Figure 3-1: Protocol transition system

3.2 Formal Definition of Protocol Transition Systems

Now we provide the formal syntax and semantics for protocol transition systems.

Notations Throughout the dissertation, we will use several notations for lists (sequences) and finite maps. An overline (e.g., $\overline{l}$) denotes a list. $\overline{\overline{l}}$ denotes a list of lists. $[]$, $(\overline{l} + e)$, $(\overline{l_1} + \overline{l_2})$, $(\overline{l_1} - \overline{l_2})$, and $(\overline{l_1} \# \overline{l_2})$ denote nil, single-element append, general append, subtraction, and disjointness of lists, respectively. We use the same operation $(+)$ for the single-element and general append. $\oplus\overline{\overline{l}}$ flattens the list of lists $\overline{\overline{l}}$ with repeated concatenation. $|\overline{l}|$ is the length of a list.

Regarding a list of key-value pairs as a finite map, we override notations for lists. For example, $(M + \overline{l})$ updates multiple key-value pairs in a finite map M. Moreover, we overload the same operation $(M + (k, v))$ for a single update for simplicity.

We will use $\langle \cdot \rangle$ to denote a struct and use a name (e.g., $s.\mathsf{fd}$) to access a field value. $(\overline{s.\mathsf{fd}})$ will be used as a shorter notation for $(\mathsf{List.map}\ (\lambda s.\ s.\mathsf{fd})\ \overline{s})$.

3.2.1 Syntax

Hemiola uses formal protocol transition systems as an underlying basis for reasoning about cache-coherence protocols. Figure 3-1 explains what such systems are. A *message* m is a communication unit, consisting of a Boolean message type, a message ID, and a value. A message type is false (true) for a request (response). A message ID belongs to an enumeration of message kinds. We use *value* to refer to each line in a cache or a memory. Note that a struct sometimes has an *index* to distinguish it from the other components. A pair (i, m) is used to represent a message m residing in a channel with an index i.

Rules make local state transitions within an object. A rule r is a struct composed of its rule index, a precondition (prec), and a transition function (trs). Each rule has a unique index within an object. A precondition prec takes two arguments, a current object state and input messages (as a list of pairs (i, m)), and decides whether the rule can be executed or not with the current state. A transition function takes the same arguments but returns the next object state and output messages (also as a list of (i, m)). Note that our formalization of the protocol transition system is shallowly embedded in Coq, e.g., precondition and transition definitions use native Coq function types.

An object O contains its object index (unique within a system), an initial state (o_{init}), and rules ($\overline{r}$) that make local state transitions within the object. The highest-level component is a system S, which contains information about objects and channels. It consists of objects ($\overline{O}$) and channel indices for internal messages ($\overline{i_{\mathrm{in}}}$), external inputs ($\overline{i_{\mathrm{rq}}}$), and external outputs ($\overline{i_{\mathrm{rs}}}$). It is necessary to distinguish between internal and external channels, in order to define external behaviors of the system, i.e., the interface of the cache-coherence protocol with processor cores, which will be explained in section 3.2.2.

The definition is general in that any object can access any channel in the system, just by mentioning the channel index in a state transition. In practical cache-coherence protocols, however, only specific message-passing patterns are used. Re-

Types:

$$\text{Message States} \quad M \in \mathbb{I} \to \overline{\mathbb{M}} \quad \text{State} \quad s \in \mathbb{S} ::= \langle \overline{o}, M \rangle$$

$$\text{Label} \quad l ::= l_\epsilon \mid l_{\text{in}}(\overline{im}) \mid l_{\text{out}}(\overline{im}) \mid l_{\text{int}}(O.i, r.i, \overline{im}, \overline{im})$$

Step:

$$\text{StepSilent:} \frac{}{s \xrightarrow[S]{l_\epsilon} s}$$

$$\text{StepIns:} \frac{\overline{im} \neq [] \qquad \overline{im.i} \subseteq S.\overline{i_{\text{rq}}}}{\langle \overline{o}, M \rangle \xrightarrow[S]{l_{\text{in}}(\overline{im})} \langle \overline{o}, M + \overline{im} \rangle}$$

$$\text{StepOuts:} \frac{\overline{im} \neq [] \qquad \overline{im} \subseteq M.\mathsf{hds} \qquad \overline{im.i} \subseteq S.\overline{i_{\text{rs}}}}{\langle \overline{o}, M \rangle \xrightarrow[S]{l_{\text{out}}(\overline{im})} \langle \overline{o}, M - \overline{im} \rangle}$$

$$\text{StepInt:} \frac{\begin{array}{ccc} S = \langle \overline{O}, \overline{i_{\text{in}}}, \overline{i_{\text{rq}}}, \overline{i_{\text{rs}}} \rangle & O \in S.\overline{O} & r \in O.\overline{r} \\ \overline{im^{\text{ins}}.i} \subseteq S.\overline{i_{\text{in}}} \cup S.\overline{i_{\text{rq}}} & \overline{o}[O.i] = o_1 & \overline{im^{\text{ins}}} \subseteq M.\mathsf{hds} \\ r.\mathsf{prec}\,(o_1, \overline{im^{\text{ins}}}) & r.\mathsf{trs}\,(o_1, \overline{im^{\text{ins}}}) = (o_2, \overline{im^{\text{outs}}}) & \\ \overline{im^{\text{outs}}.i} \subseteq S.\overline{i_{\text{in}}} \cup S.\overline{i_{\text{rs}}} & \overline{im^{\text{ins}}.i} \# \overline{im^{\text{outs}}.i} & \end{array}}{\langle \overline{o}, M \rangle \xrightarrow[S]{l_{\text{int}}(O.i, r.i, \overline{im^{\text{ins}}}, \overline{im^{\text{outs}}})} \left\langle \begin{array}{l} \overline{o} + (O.i, o_2), \\ M - \overline{im^{\text{ins}}} + \overline{im^{\text{outs}}} \end{array} \right\rangle}$$

Figure 3-2: Step semantics in protocol transition systems

calling our interleaving example in section 2.1, we see that the three objects (P, C_1, and C_2) form a tree shape and communicate through the designated three channels between the parent and a child. One of the main intuitions of the Hemiola framework is that safe interleavings are ensured just by restricting the communication patterns among objects (with proper locking mechanisms). We will explore how such a restriction is enforced by defining a domain-specific language (DSL) in chapter 5 and how the DSL implies safe interleavings in chapter 6.

3.2.2 Semantics

State-transition steps

Figure 3-2 describes the semantics for state-transition steps in a protocol transition system. A state transition (step) happens by a rule that takes input messages, makes an object-state transition, and generates output messages. The semantics for a step is presented as a judgment $s_0 \xrightarrow[S]{l} s_1$, where S is the system to execute, s_0 is a prestate, s_1 is a poststate, and l is a label generated by the state transition. The state of a system (in domain $\mathbb{S}$) is a pair of object states and message states. Object states are represented in a finite map from object indices to object states. Message states are also represented in a finite map from channel indices to ordered queues of messages.

From now on, we assume that all the input and output messages used in the step definitions do not share the same channel, i.e., ($\mathsf{List.NoDup}\ \overline{im.i}$). In other words, while taking inputs and generating outputs, each step case never accesses a channel twice. As already mentioned in section 3.1, this assumption is fair and practical in hardware.

Rule [StepSilent] represents the case where no state transition happens in the current step; an empty label (l_ϵ) is generated in this case. A system may accept input messages from the external world. [StepIns] describes this case, where the external input messages ($\overline{im}$) should not be empty ($\overline{im} \neq []$), and channels of the messages are valid ($\overline{im.i} \subseteq S.\overline{i_{\text{rq}}}$), i.e., the input messages are all put to external-request channels. An external-inputs label ($l_{\text{in}}(\overline{im})$) is generated in this case. [StepOuts] describes the opposite case, for output messages being released to the external world. In this case, in addition to the [StepIns] case, each output message should be in the head (the first element) of its channel ($\overline{im} \subseteq M.\mathsf{hds}$).

Lastly, [StepInt] deals with a state transition by a rule (r) in an object (O). It nondeterministically chooses an object and a rule in the object, checks that the precondition holds ($r.p\,(o_1, \overline{im^{\text{ins}}})$), and applies the transition to update the state of the system ($r.t\,(o_1, \overline{im^{\text{ins}}}) = (o_2, \overline{im^{\text{outs}}})$). An internal label ($l_{\text{int}}(O.i, r.i, \overline{im^{\text{ins}}}, \overline{im^{\text{outs}}})$) is generated in this case, which records an object index, a rule index, input messages,

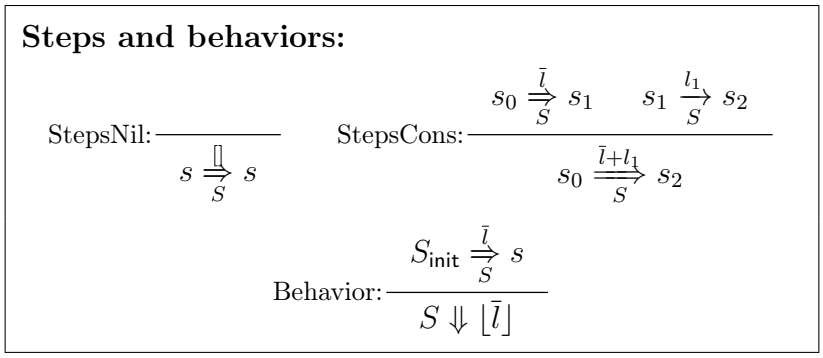


Figure 3-3: Multiple transition steps and behaviors in protocol transition systems

and output messages. Each input message should be from either an internal channel or an external-request one ($\overline{im^{\text{ins}}.i} \subseteq S.\overline{i_{\text{in}}} \cup S.\overline{i_{\text{rq}}}$) and should be the first element of the channel ($\overline{im^{\text{ins}}} \subseteq M.\mathsf{hds}$). In contrast to the input messages, each output message should be enqueued to either an internal channel or an external-response one ($\overline{im^{\text{outs}}.i} \subseteq S.\overline{i_{\text{in}}} \cup S.\overline{i_{\text{rs}}}$). Lastly, the channels of the input and output messages should be disjoint from each other ($\overline{im^{\text{ins}}.i} \# \overline{im^{\text{outs}}.i}$). Note that the semantics is based on ordered channels, so messages are *enqueued* and *dequeued* in each state-transition case. We use notations $M + \overline{im}$ and $M - \overline{im}$ for such operations.

The step semantics is naturally lifted to one for multiple steps, as shown in Figure 3-3. It is presented as a judgment $s_0 \overset{\bar{l}}{\underset{S}{\Rightarrow}} s_1$, where S is the system to execute, s_0 is a prestate, s_1 is a poststate, and $\bar{l}$ is a *sequence of labels* generated by executions of the steps. [StepsNil] serves the case where no state transitions happen, and no labels are generated in this case. [StepsCons] is a natural inductive constructor that combines previous steps ($s_0 \overset{\bar{l}}{\underset{S}{\Rightarrow}} s_1$) and a new one ($s_1 \overset{l_1}{\underset{S}{\rightarrow}} s_2$). The label by the new step is appended to the end of the label sequence of the previous steps.

Throughout the dissertation, we will now call a sequence of labels a *history*. We say that a state s is *reachable* iff there is a history $\bar{l}$ such that $S_{\mathsf{init}} \overset{\bar{l}}{\underset{S}{\Rightarrow}} s$ holds, where S_{init} is the initial state of the system S, constructed by composing all initial object states. We use a simpler notation $S \Rightarrow s$ for reachable states. We also say that a history $\bar{l}$ is *legal* iff there is a state s such that $S_{\mathsf{init}} \overset{\bar{l}}{\underset{S}{\Rightarrow}} s$ holds. We write $S \overset{\bar{l}}{\Rightarrow} \bullet$ to assert that a history is legal.

Behaviors and correctness

A system S has a behavior $\lfloor \bar{l} \rfloor$, denoted as $S \Downarrow \lfloor \bar{l} \rfloor$, if there exists an execution of steps that generates $\bar{l}$, starting with the initial state of S ([Behavior] in Figure 3-3). Here the $\lfloor \cdot \rfloor$ operation filters out silent (l_ϵ) and internal (l_{int}) labels so only the external parts remain. We call such a sequence of labels a *trace*. In other words, a trace only consists of external-inputs and external-outputs labels.

Finally we define trace refinement as a notion of correctness in protocol transition systems:

Definition 3.2.1 (Trace Refinement)**.** *A system I ("implementation") trace-refines another system S ("specification"), written as $I \sqsubseteq S$, iff every trace of I is also a trace of S:*

$$I \sqsubseteq S \triangleq \forall \bar{t}.\ I \Downarrow \bar{t} \rightarrow S \Downarrow \bar{t}.$$

Trace refinement is one of the well-known correctness criteria to claim that the external (observable) behavior of a given implementation is within the behavior boundary of the specification. In other words, by proving trace refinement, we can say that the implementation does not go wrong in terms of the specification.

How do we prove trace refinement for a given implementation and a spec? It is usually proven by establishing a *simulation* relation between the implementation and the spec states:

Definition 3.2.2 (Simulation)**.** *We call a relation between two states $(\sim) : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{P}$ a simulation between the systems I and S iff 1) the relation holds for the initial states and 2) a step in S exists for every step in I, which generates the same external label and preserves the relation:*

1) $I_{init} \sim S_{init}$,

2) $\forall s_0, s_1, l.\ s_0 \xrightarrow[I]{l} s_1 \rightarrow \forall t_0.\ s_0 \sim t_0 \rightarrow \exists t_1.\ t_0 \xrightarrow[S]{l} t_1 \wedge s_1 \sim t_1$.

It is also well-known that simulation directly implies trace refinement [11], which is proven simply by induction on state-transition steps:

Theorem 3.2.1 (Simulation implies trace refinement)**.** *If there is a simulation* ($\sim$) *between two systems* I *and* S*, then* $I \sqsubseteq S$*.*

Chapter 4

Serializability in Protocol Transition Systems

Serializability [64, 5] is a celebrated notion of concurrency correctness that originated with databases and distributed systems. While each transaction in a system affects multiple values, serializability guarantees that concurrent execution of such transactions is correct in that the effect (state change) is the same as if the transactions were executed separately in sequence. In this section, we provide the formal definition of serializability on top of a protocol transition system and explain how it eases the burden of designing and proving conventional invariants.

4.1 Cache-Coherence Protocols as Distributed Protocols

We already explored in section 2.1 with the interleaving example that the cache objects in a cache-coherent memory system handle multiple requests from the processor cores concurrently, in a distributed way. We also learned that transactions to handle such requests are interleaved, so it is crucial in designing and proving a cache-coherence protocol to ensure safe interleavings.

A very well-known approach to ensuring safe interleavings is to prove *noninter-*

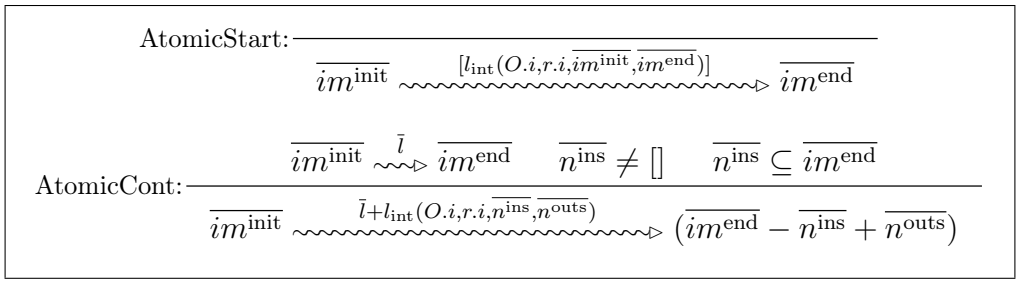
ference, which basically claims that no other state transitions spuriously affect state relevant to an ongoing transaction. Not constrained to the cache-coherence protocols, noninterference has been developed and employed in verifying other hardware components (e.g., processors) and concurrent software. We will discuss various uses of noninterference more in chapter 7.

Model checkers have employed noninterference lemmas for a long time. The lemmas are often described explicitly by the user (guided by intuition) or discovered automatically (typically guided by counterexamples). The established style of noninterference reasoning, though, has an inherent weakness: it requires explicit lemma statements per-protocol. Another issue is that even though these lemmas are stated and proven, it is hard to be sure if the lemmas are enough to resolve all possible interleavings in the system.

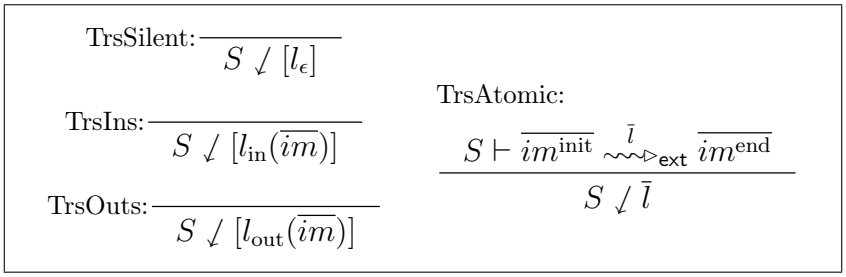
Another well-known approach is to prove *serializability*, which claims that the overall state transition of any interleaved transactions can be interpreted as if they are executed serially, i.e., *atomically in some order with no interleaving*. Serializability has been mainly developed and used in database systems and software distributed systems. Regarding a cache-coherence protocol as a distributed protocol, one of the main topics of this dissertation is to demonstrate that serializability can also be employed to prove the protocol. To our knowledge, the notion of serializability has not been used to provide a formal correctness proof of a cache-coherence protocol. Instead of stating noninterference lemmas coupled to each protocol, which is a conventional approach to the verification of cache-coherence protocols, in Hemiola we try to find general conditions that ensures serializability, *extracted and abstracted* from practical protocols, so the conditions can apply to various protocols.

4.2 From Atomic Histories to Serializability

In order to obtain the formal definition of serializability, we should define each basic term first – transactions, sequential executions, etc. In this section, we provide all the formal definitions required to define serializability, starting with atomic histories.



(a) Atomic histories



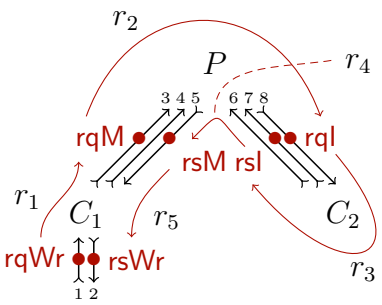
(b) Transactions

Figure 4-1: Atomic histories and transactions in protocol transition systems

4.2.1 Atomic histories

Figure 4-1 provides basic definitions of atomic histories and transactions. A history h is *atomic* iff it satisfies the predicate $(\overline{im^{\text{init}}} \overset{h}{\rightsquigarrow} \overline{im^{\text{end}}})$ with *initial messages* $\overline{im^{\text{init}}}$ and *live messages* $\overline{im^{\text{end}}}$. [AtomicStart] initializes a history with an internal label. [AtomicCont] inductively adds a label to an atomic history, where the input messages of the new label *should be from the live messages of the previous atomic history*. A shorter notation $(\overset{h}{\rightsquigarrow})$ will be used when $\overline{im^{\text{init}}}$ and $\overline{im^{\text{end}}}$ are clear from context.

Figure 4-2 presents an example of an atomic history from the interleaving example presented in section 2.1. h is generated by executions of five rules, $r_1 \in C_1.\overline{r}$, $r_2 \in P.\overline{r}$, $r_3 \in C_2.\overline{r}$, $r_4 \in P.\overline{r}$, and $r_5 \in C_1.\overline{r}$. r_1 takes an input message $(1, \mathsf{rqWr})$ (from the channel with index 1) as an initial message of the history. r_2 takes $(3, \mathsf{rqM})$, the output message from r_1. r_3 takes $(8, \mathsf{rql})$, the output message from r_2. r_4 takes $(7, \mathsf{rsl})$, the output message from r_3. Finally, r_5 takes $(5, \mathsf{rsM})$, the output message from r_4. Summing up all the rule executions, by the definition of an atomic history we get the predicate $[(1, \mathsf{rqWr})] \overset{h}{\rightsquigarrow} [(2, \mathsf{rsWr})]$. This example shows that an atomic history intuitively captures a *transaction flow* triggered by the initial messages.



$$
\begin{aligned}
h \triangleq\ & [l_{\text{int}}(C_1.i, r_1.i, [(1, \mathsf{rqWr})], [(3, \mathsf{rqM})]);\\
& l_{\text{int}}(P.i, r_2.i, [(3, \mathsf{rqM})], [(8, \mathsf{rqI})]);\\
& l_{\text{int}}(C_2.i, r_3.i, [(8, \mathsf{rqI})], [(7, \mathsf{rsI})]);\\
& l_{\text{int}}(P.i, r_4.i, [(7, \mathsf{rsI})], [(5, \mathsf{rsM})]);\\
& l_{\text{int}}(C_1.i, r_5.i, [(5, \mathsf{rsM})], [(2, \mathsf{rsWr})])]
\end{aligned}
$$

Figure 4-2: An example of an atomic history

Note that an atomic history does not need to be a completed transaction. For example, taking the first three rule executions (by r_1, r_2, and r_3) will generate the following history:

$$
\begin{aligned}
h_p \triangleq\ & [l_{\text{int}}(C_1.i, r_1.i, [(1, \mathsf{rqWr})], [(3, \mathsf{rqM})]);\\
& l_{\text{int}}(P.i, r_2.i, [(3, \mathsf{rqM})], [(8, \mathsf{rqI})]);\\
& l_{\text{int}}(C_2.i, r_3.i, [(8, \mathsf{rqI})], [(7, \mathsf{rsI})])].
\end{aligned}
$$

h_p is still an atomic history satisfying $[(1, \mathsf{rqWr})] \overset{h_p}{\rightsquigarrow\!\triangleright} [(7, \mathsf{rsI})]$ and is not completed in the sense that the live message (rsI) is not a response sent on an external channel.

We call a history *externally atomic* if initial messages are external requests ($\overline{m^{\text{init}}.i} \subseteq S.\overline{i_{\text{rq}}}$), denoted as ($S \vdash \overline{im^{\text{init}}} \overset{h}{\rightsquigarrow\!\triangleright_{\mathsf{ext}}} \overline{im^{\text{end}}}$). We sometimes use the shorter notation $S \overset{h}{\rightsquigarrow\!\triangleright_{\mathsf{ext}}}$ for some $\overline{im^{\text{init}}}$ and $\overline{im^{\text{end}}}$. The history h in the example is externally atomic, satisfying a predicate $S \vdash [(1, \mathsf{rqWr})] \overset{h}{\rightsquigarrow\!\triangleright_{\mathsf{ext}}} [(7, \mathsf{rsI})]$, since the initial message $(1, \mathsf{rqWr})$ is an external request (from a processor core attached to C_1). An externally atomic history records the way the system responded to some set of messages received from the outside world.

4.2.2 Transactions

Transactions are the next level of abstraction, presented in Figure 4-1b. A transaction is either a single silent label ([TrsSilent]), a single external inputs label ([TrsIns]), a single external outputs label ([TrsOuts]), or an externally atomic history ([TrsAtomic]). In other words, the permitted atomic execution steps, in this transactional semantics, are arrival of new messages from the outside world, release of messages to the outside world, or uninterrupted execution of an atomic history. We denote by $S \swarrow h$ that h is a transaction in S.

Note that an external-inputs label and an atomic history are regarded as separate transactions. In other words, accepting an external request (or releasing an external response) is regarded as a transaction separated from the atomic history handling the request (generating the response). This design choice gives more serializable histories in a given system, where the property is still meaningful enough to use it to verify distributed protocols. We will see why this relaxed definition is a better choice right in the next section (section 4.2.3).

4.2.3 Sequential histories and serializability

With a clear notion of transactions, we can easily define sequential histories and serializability.

Definition 4.2.1 (Sequential Histories and Serializability)**.**

1. *A history h is sequential iff the history is a concatenation of transactions:*

$$\mathsf{Sequential}\ S\ h \triangleq \exists\, \overline{t}.\ (\forall t \in \overline{t}.\ S \swarrow t) \wedge h = \oplus \overline{t}.$$

2. *A legal history h is serializable in the system S iff there exists a sequential history*

that reaches the same state:

$$\mathsf{Serializable}\ S\ h \triangleq \forall s.\ S_{\mathit{init}} \overset{h}{\underset{S}{\Rightarrow}}\ s \rightarrow \\ \exists\, h_{seq}.\ \mathsf{Sequential}\ S\ h_{seq} \wedge S_{\mathit{init}} \overset{h_{seq}}{\underset{S}{\Longrightarrow}}\ s.$$

3. A system S is <u>serializable</u> iff every legal history is serializable:

$$\mathsf{Serializable}\ S \triangleq \forall h.\ \mathsf{Serializable}\ S\ h.$$

It is quite natural to think of sequential histories only with atomic histories, e.g., like the one in Figure 2-9, but why do we have to separate external input/output labels from atomic histories? Suppose that an object C in a system S takes two external requests, say rq_1 and rq_2, in-order from the same channel and finishes handling rq_2 earlier than rq_1. Even if C should start handling rq_1 first because the channel is ordered, this case may happen when rq_1 requires further treatment by other objects while C can take rq_2 and respond immediately to the corresponding external channel. We may have the following legal history h in S for this kind of scenario:

$$h = l_{\text{in}}([(i_{\text{rq}}, \mathsf{rq}_1)]) + l_{\text{in}}([(i_{\text{rq}}, \mathsf{rq}_2)]) + h_a + l_{\text{out}}([(i_{\text{rs}}, \mathsf{rs}_2)]) + l_{\text{out}}([(i_{\text{rs}}, \mathsf{rs}_1)]),$$

where i_{rq} (and i_{rs}) is the external-request (external-response) channel. If we had not separated the external labels from an atomic history and regard the combined history as a transaction, then we cannot sequentialize h, i.e., separating the red and blue transactions, thus h is not serializable. However, it is still meaningful if we can just sequentialize the internal processes of rq_1 and rq_2 (presented as h_a) to have two separated atomic histories. The next section will explain how we benefit from this serializability definition. Lastly, note that this relaxed definition is still safe in that it does not require preserving the order of the trace.

4.3 Predicate Messages in Atomic Histories

Now we discuss how to exploit our notion of serializability: how does it help prove trace refinement between an implementation and its spec? There is a clear gap between two notions, serializability and trace refinement: the former only deals with *reachable states*, whereas the latter only deals with *behaviors*. However, we can bridge the gap by using serializability to help prove *invariants* (which only concern reachable states), since a simulation – which implies trace refinement – proof usually requires a number of invariants of the implementation.

We start by defining a conventional notion of an invariant:

Definition 4.3.1 (Invariants)**.** *We call* $\mathcal{I} : \mathbb{S} \to \mathbb{P}$ *an* <u>*invariant*</u> *over a system* S *if* $\mathcal{I}$ *holds for all reachable states, i.e.,* $\forall s.\ S \Rightarrow s \to \mathcal{I}(s)$.

In proving the correctness of a distributed protocol, it is quite common to state an invariant like "some important property holds whenever the system includes a certain message in a certain channel." We call such an invariant a *predicate message*, formally defined as following:

Definition 4.3.2 (Predicate Messages)**.** *A* <u>*predicate message*</u>*, denoted as* $S \vdash im\{P\}$*, is an invariant over a system* S *that additionally requires* im*, a pair of a channel and a message, to be in the system:*

$$S \vdash im\{\mathcal{I}\} \triangleq \forall s.\ S \Rightarrow s \wedge im \in s.M \to \mathcal{I}(s).$$

We will write just $im\{\mathcal{I}\}$ when the system S is clear from context. We will use an even-shorter version $m\{\mathcal{I}\}$ when the channel that contains the message is not ambiguous. The same notation $\mathsf{id}\{\mathcal{I}\}$ will be used most commonly to define a predicate message that holds for every message with a given ID (id) with the clear channel.

Figure 4-3 presents an example of a predicate message. Recalling the example protocol from section 2.1, when the system has a message of a kind rsM, we expect

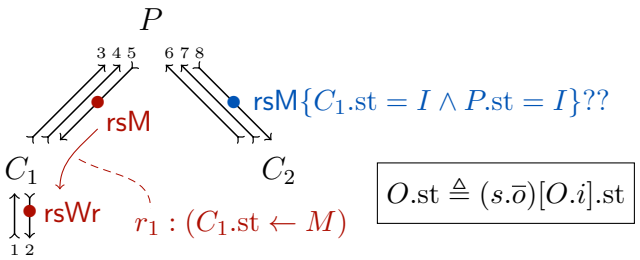


Figure 4-3: Interference breaks a predicate message

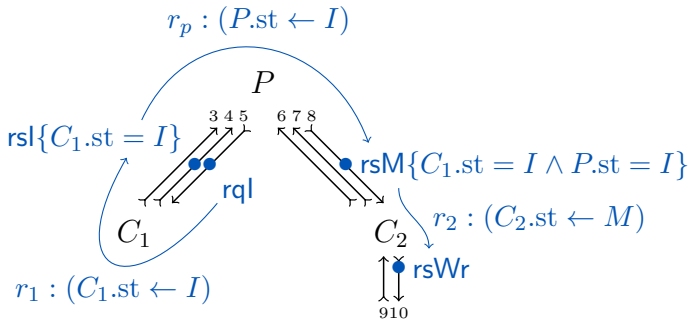
the parent and the other child (who is not the requestor) to have I status (like $\{C_1.\text{st} = I \wedge P.\text{st} = I\}$ in the figure). However, the predicate is broken when a state transition happens *by another transaction*; for instance, the predicate no longer holds if a state transition happens by $r_1 \in C_1$, which takes an input $(5, \mathsf{rsM})$ and updates the status of C_1 to M. From now on, we will use a brief notation like $r_1 : (C_1.\text{st} \leftarrow M)$ to denote a rule with some commands.

Investigating this corner case carefully, we can find that actually no two different rsM messages can be in the system at the same time. This finding implies that now the predicate message for rsM should have a much-more-complicated form, which considers *all possible noninterference cases*. The complete desired predicate message for $(8, \mathsf{rsM})$ will then look like:

$$(8, \mathsf{rsM}) \left\{ \begin{array}{l} \textit{// The original predicate} \\ C_1.\text{st} = I \ \wedge \ P.\text{st} = I \ \wedge \\ \textit{// Noninterference with another transaction to get } M \textit{ from } C_1 \\ (7, \mathsf{rsI}) \notin s.M \ \wedge \ (5, \mathsf{rsM}) \notin s.M \ \wedge \\ \textit{// More noninterference cases will be required} \\ \cdots \end{array} \right\}$$

It is indeed a burden to consider all possible interleavings per predicate message. We would not have faced such a complication if we could ensure that no other transactions interfere while handling a transaction. Serializability guarantees exactly that simplification, and Hemiola provides a way of designing and proving predicate messages in the simpler form, not taking any interleavings into account.

Our novel approach to employing predicate messages in atomic histories begins



$$\mathcal{I}_a\ s\ \overline{im_o} \triangleq \begin{array}{rcl} ((4, \mathsf{rsl}) \in \overline{im_o} & \rightarrow & (s.\bar{o})[C_1.i].\mathrm{st} = I) \wedge \\ ((8, \mathsf{rsM}) \in \overline{im_o} & \rightarrow & (s.\bar{o})[C_1.i].\mathrm{st} = I \wedge (s.\bar{o})[P.i].\mathrm{st} = I) \end{array}$$

Figure 4-4: Predicate messages defined as an atomic invariant

with formalizing the notion of *atomic invariants*:

Definition 4.3.3 (Atomic Invariants)**.** $\mathcal{I}_a : \overline{\mathbb{IM}} \times \mathbb{S} \rightarrow \mathbb{P}$ *is called an atomic invariant iff* $\mathcal{I}_a\ (\overline{im_o}, s_1)$ *holds for any atomic history* h *with* $s_0 \overset{h}{\underset{S}{\Rightarrow}} s_1$ *and* $\overline{im_i} \overset{h}{\rightsquigarrow} \overline{im_o}$.

Note that an atomic invariant takes the live messages ($\overline{im_o}$) from an atomic history as an additional argument, compared with ordinary invariants.

Figure 4-4 shows an example of predicate messages defined in an atomic history, formalized as an atomic invariant. $\mathcal{I}_a$ shows how predicate messages are formalized into an atomic invariant; it is basically a conjunction of predicates, where each predicate has a form of $(im \in \overline{im_o} \rightarrow P(s))$, claiming that the predicate P holds when im is in *the live messages* $\overline{im_o}$. Compare this form with the original definition of predicate messages, defined in Definition 4.3.2, where $(im \in s.M)$ is replaced with $(im \in \overline{im_o})$, thus avoiding possible interference by other messages.

We can prove that the atomic invariant $\mathcal{I}_a$ holds for the atomic history (in the figure) step-by-step:

- The initial step of the atomic history is the one by r_1. The live messages are $[(4, \mathsf{rsl})]$. Since r_1 changes the status of C_1 to I, it is straightforward to prove $\mathcal{I}_a$.
- The next step is by r_p, and at this point the live messages are $[(8, \mathsf{rsM})]$. By the induction hypothesis, we obtain the predicate message $(4, \mathsf{rsl})\{C_1.\mathrm{st} = I\}$. Since r_p changes the status of P to I, we can prove the predicate for $(8, \mathsf{rsM})$.

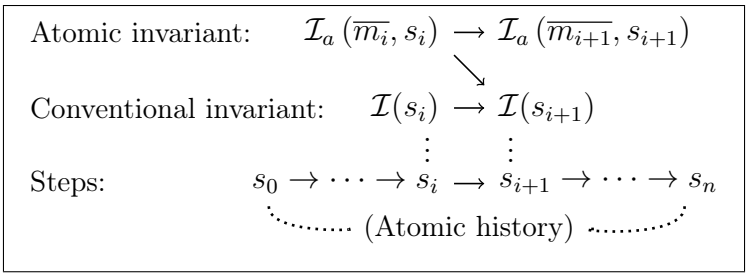


Figure 4-5: Atomic invariants in conventional invariant proof

- The last step is by r_2, and the live messages are $[(10, \mathsf{rsWr})]$. $\mathcal{I}_a$ trivially holds here since it does not contain any predicate for $(10, \mathsf{rsWr})$.

It is worth emphasizing again that the invariant proof was straightforward since no other state transitions interfere with an atomic history.

How do atomic invariants help prove conventional invariants? If the system S is serializable, by its definition, for every reachable state s with a history h ($S_{\mathsf{init}} \overset{h}{\Rightarrow}_S s$), there is a sequential history h_{seq} that reaches the same state ($S_{\mathsf{init}} \overset{h_{\mathrm{seq}}}{\Longrightarrow}_S s$). Since h_{seq} is a concatenation of transactions, an invariant can be proven *by showing that any transaction preserves it.*

The three induction cases ([TSilent], [TIns], and [TOuts]) do not need any special treatment, since each case induces just a single step. The [TAtomic] case, however, requires coordination with atomic invariants. In other words, we can employ both conventional/atomic invariants ($\mathcal{I}$ and $\mathcal{I}_a$) to prove the ones for the next state (s_{i+1}): Figure 4-5 explains how to use an atomic invariant $\mathcal{I}_a$ while proving a conventional invariant $\mathcal{I}$ in the [TrsAtomic] case. The point is that we can employ both $\mathcal{I}(s_i)$ and $\mathcal{I}_a(\overline{m_i}, s_i)$ to prove $\mathcal{I}(s_{i+1})$. For instance, we may want to have an invariant claiming that at most one node of the system has M status at a time. We will indeed use this invariant in our case studies, although the exact statement is a bit more complex. The predicate messages defined in Figure 4-4 will play a crucial role here, e.g., the one for $(8, \mathsf{rsM})$ says that C_1 and P both have I status, which means that the state transition by $(r_2 : C_2.\mathrm{st} \leftarrow M)$ preserves the invariant. We will see more comprehensive uses of predicate messages in our case studies (chapter 8).

Part II

A Framework with the Serializability Guarantee

Chapter 5

The Hemiola Domain-Specific Language

As explained in chapter 4, on top of serializability, it is much easier to design and prove invariants. That being said, it will still be a large burden if a user has to prove serializability per-protocol. In this chapter, we would like to provide *abstract conditions* to prove serializability, in order not to prove it directly for each protocol. Furthermore, we design a domain-specific language (DSL) that every protocol defined on top of this language automatically satisfies the conditions to guarantee serializability. The conditions have already been mentioned with the motivating example in section 2.1 – network topology and locking mechanisms, extracted from transient states of practical cache-coherence protocols.

5.1 Topology and Network Requirements

In order to employ the serializability guarantee in Hemiola, the objects in a given system should form a tree topology. Most cache-coherent memory subsystems follow this topology, where leaf nodes correspond to L1 caches, and the root corresponds to the main memory. A child and its parent in the tree communicate using the three channels shown in section 2.1: an upward-request channel, an upward-response channel, and a single downward channel. Note that the use of the three channels does

not mean the actual hardware implementation should use just the three channels; it can use any channel implementation that refines the three channels regarded as a specification. For example, we may want to have an additional queue that accepts all the requests from child caches (from the corresponding upward-request channels), so the parent can have a single entry point to handle child requests. We will indeed see in chapter 9 how logical channels in Hemiola are implemented at register-transfer level.

Hemiola as a Coq library generates the topology and network channels automatically from instances of a simple inductive type. For example, the following tree definition will generate topology and channels for four L1 caches, two L2 caches, the last-level cache (LLC), and the main memory:

```
Definition t: tree := Node [Node [Node [Leaf; Leaf]; Node [Leaf; Leaf]]].
```

5.2 Locking Mechanism

We saw in section 2.1 why fine-grained transient states are required to ensure safe interleavings in cache-coherence protocols. Revisiting the corner case described in Figure 2-2, a child should be able to handle an invalidation request from the parent even if it is in a transient state (SM), and after the response it changes its transient state to IM. By looking at these transient states carefully, we have discovered that another necessary condition, other than the tree-topology condition, is the locking perspective from the transient states.

Hemiola supports a general locking mechanism reflecting this discovery; instead of looking at the type of a message (e.g., invalidation), the framework provides *more-general* locking, just by looking at whether the message is from the parent or one of its children. This relaxed locking mechanism is then not coupled with any specific cache-coherence protocols but is still sufficient to prove serializability.

In particular, Hemiola provides two kinds of locks: *uplocks and downlocks*. We say an object is uplocked (downlocked) when it holds an uplock (downlock). Figure 5-1 depicts the locking mechanism in Hemiola. An uplock is set when an object makes

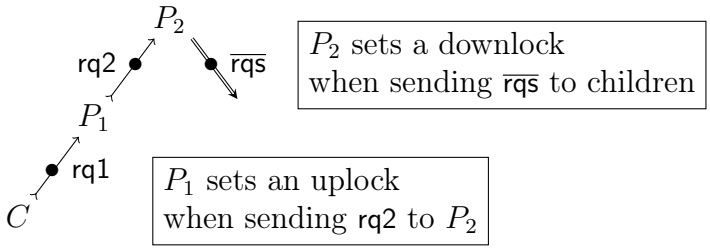


Figure 5-1: Locking mechanism in Hemiola

an upward request to its parent (P_1 in the figure); it is released when the object gets a corresponding response from the parent. On the contrary, a downlock is set when an object makes a downward request(s) to some of its children (P_2 in the figure).

	Uplocked	Downlocked
Acquiring an uplock	✗	✓
Releasing an uplock	✓	✗
Acquiring a downlock	✓	✗
Releasing a downlock	✓	✓

Figure 5-2: Locking conditions in Hemiola

Figure 5-2 shows the conditions to acquire or release the locks. We explore each case with examples in actual cache-coherence protocols:

An uplock cannot be set if the object is already uplocked. As already explained with the interleaving example in section 2.1, we do not want to make a further request to the parent (with the same line) when it already made an upward request. Requesting to the parent twice would result in spurious interleavings.

An uplock can be set even when the object is downlocked; instead, it cannot be released when downlocked. It means that an object can make a request to its parent even when it has made requests to its children. It is still safe, since the lock release will not be allowed when downlocked, i.e., the uplock release should always wait for the downlock to be released.

This case happens only when a protocol is defined with caches forming at least three levels, i.e., there is an intermediate cache object that has both a parent and

children. Suppose an intermediate cache is in an invalidation process, i.e., invalidation requests have been sent to children, and the cache has been downlocked since then. Also suppose that a child with the S status made an upward request to get the M status. In this case, the intermediate cache should be able to deal with this upward request and to make a further request to its parent, even when it is in the invalidation process. Granting the M status to the child, however, will happen after the invalidation process is finished, since the uplock can be released only after the downlock is released.

A downlock cannot be set if the object is already downlocked. This case is also presented in the example in section 2.1; we do not want to make another set of invalidation requests to children (with the same line) when a parent object is in the invalidation process. Handling two invalidation processes would lead to an incoherent state due to spurious interleavings.

A downlock can be set and released even when uplocked. This relaxation is very important to make the locking mechanism live. If a downlock could not be released when uplocked, we would have a deadlock case, where both locks wait for each other.

This case also happens when a protocol is at least 3-level. Similarly to the above case, suppose that an intermediate cache already forwarded a request from its child to the parent to get the M status, so it has been uplocked. When it receives an invalidation request from the parent at this moment, it should be able to deal with this request and possibly make further invalidation requests to some of its children. It sets a downlock for the invalidation process, even when it has been uplocked. We will see a couple of actual cases in chapter 8 with the case-study protocols, where both an uplock and a downlock are acquired to deal with two respective transactions.

Remarks on transient states

Now every cache object defined in Hemiola *does not need to set transient states* to consider all combinations among stable statuses; instead, the framework helps maintain the cache status and claim proper locks. For example, instead of setting a transient state SM, it is now desirable to maintain its status S and to set an uplock to record it just made the upward request. (The framework provides even more, actually, by guiding users to employ the *rule templates*, introduced in the next section, that automatically set proper locks.)

Each object defined in Hemiola has a semantic *lock state*, a finite map from a lock type (uplock or downlock) to lock information: the original request message, the original requestor index, and requestee indices from which we expect to get responses. In the common terminology of cache-coherence protocols, each lock corresponds to a miss status holding register (MSHR). An MSHR is set when a cache miss happens and the miss triggers further requests to obtain a proper status and up-to-date data. As with Hemiola lock holders, MSHRs play a crucial role both in deciding whether to handle certain requests and in storing information needed to handle them properly.

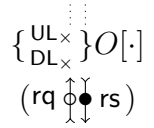
It is worth emphasizing that an uplock and a downlock are assigned for each line. In other words, the locking conditions presented in Figure 5-2 do not apply to the locks for different cache lines. This separation is safe, since in a cache-coherence protocol the transactions for different cache lines never interfere with each others. It is not practical, however, to have an uplock and a downlock for each line in the actual hardware implementation. We need to restrict the number of locks (MSHRs) to make the implementation synthesizable. A detailed discussion about this restriction will be provided in chapter 9.

5.3 Rule Templates

On top of the topology/network requirements and the locking mechanism, Hemiola provides rule templates to ensure that objects communicate within the topology and locks are properly set. In this section, we will explore what kinds of rule templates

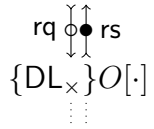
Hemiola provides and discuss their uses. Each rule template will be introduced with the form $\{P\}O[Q]$ and arrows with $\circ$ and $\bullet$, which means that the rule template is for an object O, requires input messages ($\circ$) and a precondition P, performs a state transition Q, and generates output messages ($\bullet$). UL, DL, $\mathsf{UL}_\times$, and $\mathsf{DL}_\times$ in a precondition indicate that the object is uplocked, downlocked, uplock-free, and downlock-free, respectively. $\mathsf{UL}\Uparrow$, $\mathsf{DL}\Uparrow$, $\mathsf{UL}\Downarrow$, and $\mathsf{DL}\Downarrow$ in a state transition indicate setting an uplock, setting a downlock, releasing an uplock, and releasing a downlock, respectively. SLT annotates that the rule template forbids any state modification beside locking.

Rule templates for immediate responses



(a) immd

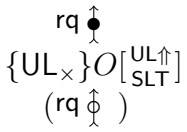
An immediate-down (**immd**) rule responds immediately to an upward request, requiring both locks be free. This rule allows not to take (generate) any input (output) messages. (The input/output messages are parenthesized in the form.) In other words, using this rule, an object can make a local state transition without any input/output messages, but in this case both locks should be free. This rule also shows that a transaction may start without any input messages as a trigger, which still matches the definition of externally atomic histories in Figure 4-1.



(b) immu

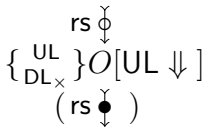
An immediate-up (**immu**) rule responds immediately to a downward request, and it only requires the downlock be free. In other words, even when uplocked, an object can make a state transition by taking a downward request and generating an upward response immediately.

Rule templates to communicate with the parent



(c) rquu

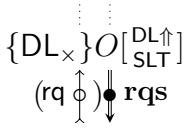
A request-up-up (**rquu**) rule (possibly) takes an upward request from a child and make another request to the parent. It requires the uplock to be free but does not care whether the object is downlocked or not. A state transition is not allowed when setting any lock (either an uplock or a downlock).



(d) rsdd

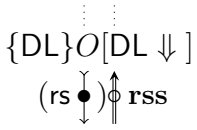
A response-down-down (**rsdd**) rule is dual to the **rquu** rule, which takes a downward response and (possibly) responds to the original child requestor. This rule releases the corresponding uplock. Note that in order to handle a response from the parent, the object should be downlock-free. This precondition is indeed required to ensure correctness (serializability), as explained in section 5.2.

Rule templates to communicate with children



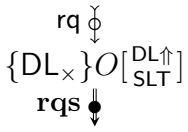
(d) rqud

A request-up-down (**rqud**) rule (possibly) takes an upward request and makes downward requests to some of the children except the child requestor. A downlock is set in this case, and thus no state transition is allowed. This rule does not have a precondition that the object should be uplock-free, i.e., the object can handle a request from the parent even when uplocked. This relaxation is still safe in terms of correctness (serializability) and necessary to avoid a deadlock.

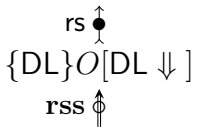


(e) rsud

A response-up-down (**rsud**) rule is dual to the **rqud** rule, which takes upward responses and (possibly) responds back to the original child requestor. The rule releases the corresponding downlock. As explained in section 5.2, it does not require any conditions on the uplock.

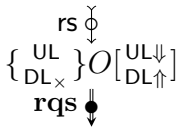


A request-down-down (**rqdd**) rule takes a downward request (from the parent) and makes downward requests to some of the children. Similar to **rqud**, a downlock is set, and no state transition is allowed. This rule also does not require the object to be uplock-free, which is to avoid a deadlock.



A response-up-up (**rsuu**) rule is dual to the **rqdd** rule, which takes upward responses and responds back to the parent. Similar to **rsud**, this rule also releases the corresponding downlock.

A rule template for whole-tree traversal



A response-down-request-down rule (**rsrq**) takes a downward response (from the parent) and makes new downward requests to some of the children. It makes a *transfer* from an uplock to a new downlock by releasing the uplock and setting a downlock, where the child-requestor information is moved from the uplock. This rule forces the order of a traversal, saying that the traversal for the outer objects must be done before traversing the inner objects. The forced order is important to avoid a deadlock. For example, if the object makes requests to its parent and children at the same time and sets both an uplock and a downlock, a deadlock may occur.

Example uses in cache-coherence protocols

In order to understand the rule templates better, we present how they can be used to describe an actual cache-coherence protocol. Figure 5-3 presents a number of rule templates used to describe the interleaving example described in section 2.1. We use an arrow (→) and a label (❶ rquu) to denote a state transition by a rule defined with

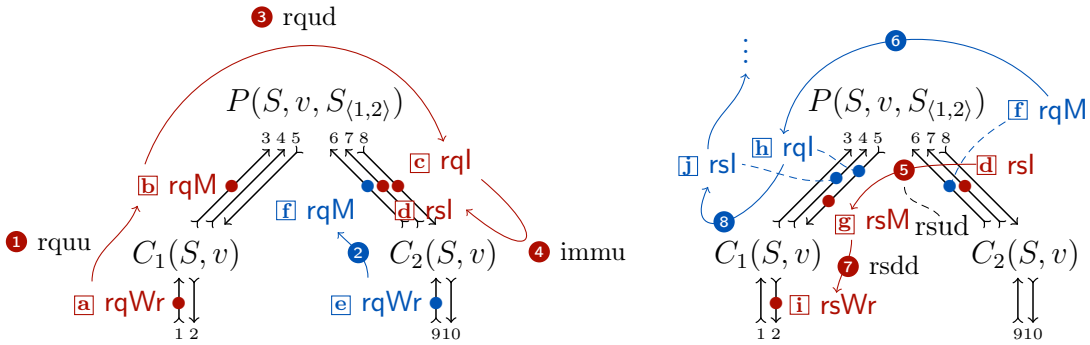


Figure 5-3: Use of the rule templates in a cache-coherence protocol

a certain rule template, e.g., 1 rquu is a rule in C_1 using the **rquu** rule template, taking an input message $(1, \mathsf{rqWr})$ and outputting $(3, \mathsf{rqM})$.

1 (or 2) uses the **rquu** rule template. It takes rqWr from the external-input channel of C_1 (or C_2) and makes a request rqM to the parent P to obtain the M status. By the definition of **rquu**, it implicitly checks if the object is not uplocked and acquires the uplock. As explained in section 2.1, this uplock is not to allow any further external requests.

3 (or 6) uses the **rqud** rule template. It takes rqM from the upward-request channel of C_1 (or C_2) and makes an invalidation request rql to the other child C_2 (or C_1). By the definition of **rqud**, it implicitly checks if the object is not downlocked and acquires the downlock. This downlock is not to allow any further upward requests to P. Indeed, after 3 is executed, f rqM has been stalled until the downlock is released by 5.

4 (or 8) uses the **immu** rule template. It takes rql from the downward channel to C_2 (or C_1) and responds to the parent with rsl. Note that **immu** does not require the uplock be free. Indeed, at the moment 4 is executed, C_2 has been uplocked by 2 rquu.

5 uses the **rsud** rule template. It takes rsl from the upward-response channel from C_2 and responds back to C_1 with rsM. **rsud** by definition releases the downlock at this moment so P can take another upward request (e.g., f rqM).

Lastly, 7 uses the **rsdd** rule template. It takes rsM from the downward channel

to C_1 and responds back with rsWr to the external world. The uplock is released at this moment. **rsdd** by definition also checks if the object is downlock-free, but in this case C_1 is a leaf node (an L1 cache) so it never sets the downlock.

Remarks

The rule templates are designed carefully to perform any practical transactions *safely* with serializable behavior. Consider an extreme case in a cache-coherence protocol. When an L1 cache wants to obtain a write permission, all the other caches should be invalidated (changing each cache status to Invalid). In order to perform such a transaction, it must be able to traverse all the other caches. This transaction kind is one of the longest-running in cache-coherence protocols, and the rule templates are designed with proper locking (UL and DL) and a state-change condition (SLT), not to create any incoherence while such a long transaction is interleaved with other transactions.

Chapter 6

Serializability in Hemiola

In this chapter, we provide the proof of serializability in Hemiola. As introduced in chapter 5, the proof largely requires two conditions: tree topology with the three channel kinds (introduced in section 5.1) and the locking mechanism (introduced in section 5.2). Since the Hemiola DSL with the rule templates faithfully follows such conditions, it is also fair to say that just using the DSL ensures serializability. While conventional approaches to verifying cache-coherence protocols deal with transient states directly and prove noninterference lemmas per-state, Hemiola provides the serializability guarantee as *the most general form of noninterference, obtained from conditions that are not coupled to any specific cache-coherence protocol*, e.g., the DSL does not mention any cache-coherence specifics.

In proving serializability we use a well-established technique called commuting reductions [44]. This reduction technique has been used before to prove correctness of concurrent software [32, 12] and distributed systems [31], but to our knowledge no past work has tried to discover serializability conditions for cache-coherence protocols and proved them using reductions. We will provide more detailed discussion of the use of commuting reductions in chapter 7.

6.1 The Intuition: Merging Atomic-History Fragments

We first present our intuition of how commuting reductions are used in the serializability proof; first, we need to formalize reduction. The most basic commuting reduction happens between two adjacent state-transition steps:

Definition 6.1.1 (Commutativity of steps)**.** *Two adjacent state-transition steps commute, denoted as* $(s_0 \xrightarrow[S]{l_0} s_1) \leftrightarrow (s_1 \xrightarrow[S]{l_1} s_2)$*, if the state transitions with the opposite order reach the same state:*

$$(s_0 \xrightarrow[S]{l_0} s_1) \leftrightarrow (s_1 \xrightarrow[S]{l_1} s_2) \triangleq \exists s_1'.\ s_0 \xrightarrow[S]{l_1} s_1' \wedge s_1' \xrightarrow[S]{l_0} s_2.$$

We will write just $l_0 \leftrightarrow l_1$ when the system S and the states involved with l_0 and l_1 are clear from context.

We can naturally lift the definition to one for histories to argue whether two histories commute or not:

Definition 6.1.2 (Commutativity of histories)**.** *Two adjacent histories commute, denoted as* $(s_0 \xRightarrow[S]{h_0} s_1) \leftrightarrow (s_1 \xRightarrow[S]{h_1} s_2)$*, if the state transitions with the opposite order reach the same state:*

$$(s_0 \xRightarrow[S]{h_0} s_1) \leftrightarrow (s_1 \xRightarrow[S]{h_1} s_2) \triangleq \exists s_1'.\ s_0 \xRightarrow[S]{h_1} s_1' \wedge s_1' \xRightarrow[S]{h_0} s_2.$$

We overload the same notation for the commutativity of histories. We will also use the shorter version $h_0 \leftrightarrow h_1$ with clear context.

Our high-level intuition for the serializability proof is that for a given interleaved history we can perform a finite number of reductions to get a sequential history that reaches the same state. More specifically, we will try to *merge* any two separated atomic-history fragments by performing reductions.

We elaborate this intuition more by using a concrete example. Figure 6-1 shows two interleaving transactions in red and blue, started from C_1 and C_2, respectively,

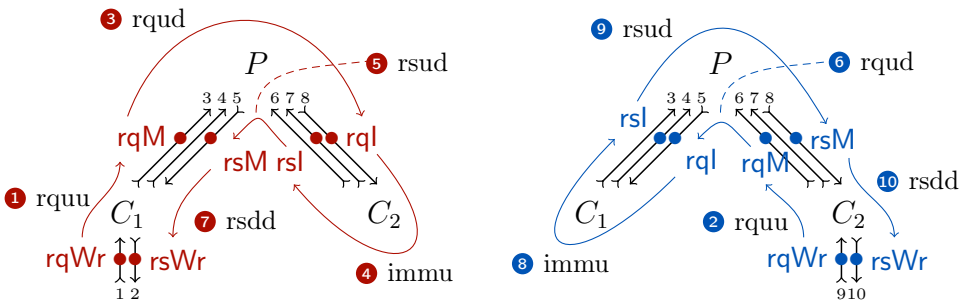


Figure 6-1: The two interleaving transactions

both seeking the M status in the simple MSI protocol presented in section 2.1. We introduce each state transition here to understand the interleaving better. We use the same annotations introduced in section 5.3 to indicate which rule template is used for each case:

- ❶ rquu and ❷ rquu: forward rqM to P since C_1 (or C_2) does not have the M status.
- ❸ rqud and ❻ rqud: P makes an invalidation request to the other child (C_2 and C_1, respectively).
- ❹ immu and ❽ immu: C_2 (or C_1) changes its status to I and responds with rsI immediately.
- ❺ rsud and ❾ rsud: the invalidation has finished so P responds with rsM to the original requestor.
- ❼ rsdd and ❿ rsdd: C_1 (or C_2) takes the response and upgrades its status to M.

Figure 6-2 shows the original legal interleaved history (top) and how reductions apply to it and make it sequential (bottom). In order to merge [❸;❹;❺] and [❼], either [❸;❹;❺] ⇿ [❻] or [❻] ⇿ [❼] should hold. We will say that [❻] is *pushed* to the left (right) when it commutes with [❸;❹;❺] ([❼]).

Two state transitions trivially commute if 1) the object state transitions happen in different objects and 2) input/output channels used for the transitions are all orthogonal to each other. (We will formalize this intuition as a lemma in the actual

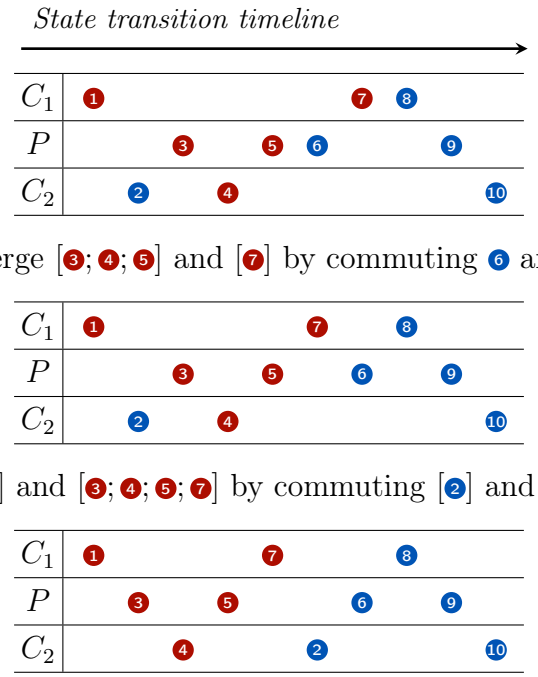
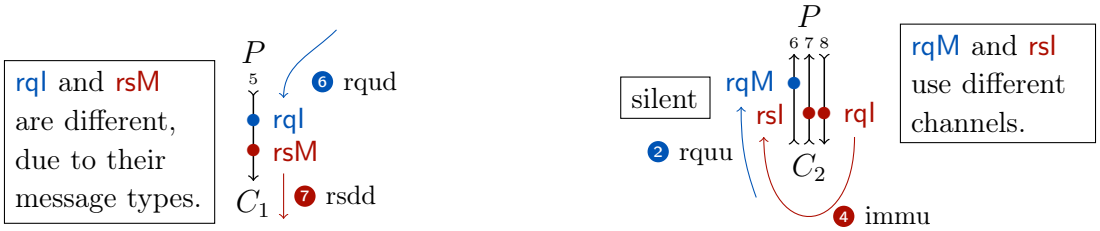


Figure 6-2: Serialization of an interleaved history by reductions



(a) Reduction by different message types **(b)** Reduction by the rule-template properties

Figure 6-3: Nontrivial reduction cases

serializability proof, shown in section 6.2.) If either of the conditions does not hold, we may need to analyze more.

In the first merge case, [❻] can be pushed to the right, i.e., (❻ ⇿ ❼) holds. The object state transitions are straightforward to deal with, since they are orthogonal (P for ❻ and C_1 for ❼). It is nontrivial to deal with the used channels, however, since the output of ❻ (rql) and the input of ❼ (rsM) share the same channel 5. Figure 6-3a depicts this case; we see ❻ and ❼ still commute, since ❼ rsdd should take a "response" (rsM) while ❻ rqud outputs a "request" (rql). It indicates that ❻ and ❼ are involved with different messages in the same channel, thus not affecting commutativity. After

the merge, we obtain a new (less-interleaved) history, shown in the middle of Figure 6-2.

Now we should merge [❶] and [❸;❹;❺;❼]; in this case, ❷ can be pushed to the right. While this reduction is trivial in the sense of orthogonal objects and channels, the only nontrivial case is between ❷ and ❹, where both rules are executed in C_2. These state transitions still commute, as shown in Figure 6-3b, since ❷ cannot make any object state transition, according to the definition of the "rquu" rule template. After this merge we finally obtain the sequential history.

How can we argue that the atomic-history fragments are always mergeable? As we start to see in the two cases in Figure 6-2, mergeability *follows from the rule templates*. While trying to merge two fragments, we especially look at the rule template used in the last state transition of the first fragment and figure out which state transitions can be performed between the two. For example, when checking mergeability between [❸;❹;❺] and [❼], we find that ❺ generates the output message rsM, and it blocks all the other transactions to go from P to C_1, e.g., ❻ was blocked until ❼ consumes rsM. We can thus commute ❻ and ❼ due to this blocking. In this sense, in proving serializability, we look at each rule template and characterize the allowed state transitions between two atomic-history fragments.

6.2 The Formal Proof of Serializability

Based on the intuitions from section 6.1, in this section we provide the formal proof of serializability. The biggest contribution of the Hemiola framework includes precise formalization of the intuitions – merging atomic fragments, commutativity among rule templates, left/right pushes, etc. – and the serializability proof employing those intuitions. The highest-level theorem will simply claim that use of good topology and the rule templates automatically guarantees serializability.

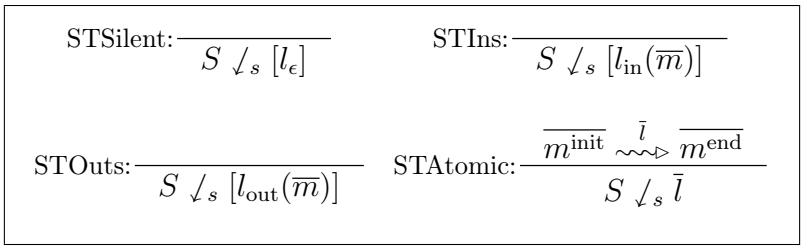


Figure 6-4: Semi-transactions

6.2.1 Semi-sequential histories

In section 6.1 we briefly provided the intuition of making a history sequential by merging atomic-history fragments repeatedly. In order to employ the intuition in the actual proof, we need to show that *finitely many* reductions suffice. For example, while merging two atomic-history fragments, if we happen to split any fragment between the two (so the number of intermediate fragments increases), we will never be sure whether the merging process will finish with a sequential history or not.

What would be the way to say *how much* a given history is sequentialized? Semi-transactions are defined in order to have a quantitative progress measure for reductions. The definition, given in Figure 6-4, is almost the same as for (ordinary) transactions, except that an atomic history does not need to be external. Thanks to this relaxation, any atomic-history fragment is a semi-transaction.

Semi-sequential histories are also defined in a similar way:

Definition 6.2.1 (Semi-Sequential Histories)**.** *A history h is semi-sequential with degree n iff the history is a concatenation of n semi-transactions.*

$$\mathsf{Sequential}_s\ S\ h\ n \triangleq \exists \bar{t}.\ |\bar{t}| = n \wedge (\forall t \in \bar{t}.\ S \not\downarrow_s t) \wedge h = \oplus \bar{t}.$$

The only difference from sequential histories is that we *count* the number of semi-transactions. In our interleaving example, looking at the original interleaved history shown in Figure 6-2, we can find the following semi-transactions in-order: [❶], [❷], [❸;❹;❺], [❻], [❼], and [❽;❾;❿]. In this case, the degree of a semi-sequential history is 6, the number of semi-transactions. Note that the degree of a semi-sequential

history is not uniquely determined. In the example, when splitting [❸; ❹; ❺] into the three separate single-label atomic histories – [❸], [❹], and [❺] – we obtain another semi-sequential history with degree 8.

This definition indicates that in order to obtain a serialized history, we should perform reductions to *decrease the count as much as possible*. Hemiola employs theorems that reflect this intuition:

Theorem 6.2.1. *Given a system S, a legal history is always semi-sequential:*

$$\forall h.\ S \overset{h}{\Rightarrow} \bullet \rightarrow \exists n.\ \mathsf{Sequential}_s\ S\ h\ n.$$

Proof. The proof is straightforward, since by definition any single-label history is a semi-transaction, and thus any history is semi-sequential, i.e., $\forall h.\ \mathsf{Sequential}_s\ S\ h\ |h|$. ■

Theorem 6.2.2 (Semi-sequentiality and serializability)**.** *If S satisfies the following property, then S is serializable:*

$$\begin{aligned}\forall h, n, s.\ S_{\mathit{init}} \overset{h}{\underset{S}{\Rightarrow}} s \wedge \mathsf{Sequential}_s\ S\ h\ n \rightarrow \\ \mathsf{Sequential}\ S\ h \vee \\ \exists h_r, m.\ S_{\mathit{init}} \overset{h_r}{\underset{S}{\Rightarrow}} s \wedge \mathsf{Sequential}_s\ S\ h_r\ m \wedge m < n.\end{aligned}$$

Proof. The proof employs the well-foundedness of the less-than operator ($<$) for natural numbers. For a given history h in a system S, we can find an initial number n that satisfies $\mathsf{Sequential}_s\ S\ h\ n$ (by Theorem 6.2.1). Now by the given property, h is either already sequential or reduces to h_r with degree $m < n$. Therefore, finite iteration of the property will eventually reduce the history to a sequential history. ■

6.2.2 Reduction of external input/output labels

For a given legal history in a system, the very first step to reduce it to a sequential history is to push all external input and output labels to the leftmost and rightmost

regions of the history, respectively. Recall that serializability (Definition 4.2.1) does not require the sequential history to preserve the behavior of a history but to reach the same state.

The following two dual theorems are enough to push all external labels:

Theorem 6.2.3. *In any legal history in the system S, the first external-input (the last external-output) label can be pushed to the beginning (the end) of the history.*

$$\forall s_0, s_1, \overline{h_0}, im, \overline{h_1}.\ \mathsf{NoExtIns}(\overline{h_0}) \rightarrow s_0 \xRightarrow[S]{\overline{h_0}+l_{in}(im)+\overline{h_1}} s_1 \rightarrow s_0 \xRightarrow[S]{l_{in}(im)+\overline{h_0}+\overline{h_1}} s_1 \text{ and}$$

$$\forall s_0, s_1, \overline{h_0}, im, \overline{h_1}.\ \mathsf{NoExtOuts}(\overline{h_1}) \rightarrow s_0 \xRightarrow[S]{\overline{h_0}+l_{out}(im)+\overline{h_1}} s_1 \rightarrow s_0 \xRightarrow[S]{\overline{h_0}+\overline{h_1}+l_{out}(im)} s_1,$$

where NoExtIns *and* NoExtOuts *are defined as follows:*

$$\mathsf{NoExtIns}(\overline{h}) \triangleq \forall l.\ l \in \overline{h} \rightarrow \forall im.\ l \neq l_{in}(im).$$

$$\mathsf{NoExtOuts}(\overline{h}) \triangleq \forall l.\ l \in \overline{h} \rightarrow \forall im.\ l \neq l_{out}(im).$$

Proof. By definition it suffices to prove $\overline{h_0} \leftrightarrow [l_{\text{in}}(im)]$. Since $\mathsf{NoExtIns}(\overline{h_0})$, each label in $\overline{h_0}$ is a silent, external-output, or internal label. It is trivial that a silent label and an external-input label commute, since the silent label makes no state transition. An external-output label and an external-input label also commute, since they use different channels. Commutativity between an internal label and an external-input label is relatively nontrivial, since an internal label may dequeue some external inputs. They still commute, however, since dequeue of an external-input label means there is already an external input message in the channel, and the external-input label adds a different message to the system. The proof for the external-output label is almost same as the one for the external-input label. ■

Figure 6-5 shows an original example history and the resulting history after pushing all the external input and output labels in the original history. We can generate such a history by performing reductions:

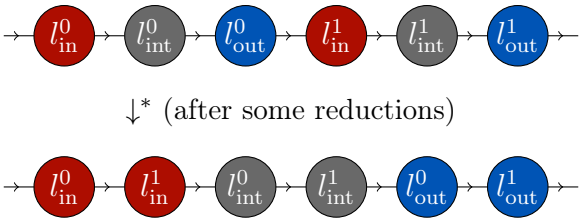


Figure 6-5: Pushing all external input/output labels

- Try to pick the first external-input label that is *not in the leftmost region of the history yet*. If it exists, it can always be pushed to the left, right after the external-input labels that are already in the leftmost region, by applying Theorem 6.2.3.

- Try to pick the last external output label that is *not at the rightmost region of the history yet*. If it exists, it can always be pushed to the right, right before the external-output labels that are already in the rightmost region, by applying Theorem 6.2.3.

Note that pushing the external input (output) labels to the leftmost (rightmost) positions is possible since the semantics of the protocol transition system is based on infinite-sized buffers, allowing the system to accept all the external input messages as early as possible and to postpone releasing all the external output messages as late as possible.

6.2.3 Interleaving and nonconfluent histories

Now we formalize the atomic-history fragments mentioned in section 6.1 – whether the two fragments belong to the same transaction or not. From this section, we will just deal with histories only consisting of internal labels, to focus on how to serialize atomic histories, assuming external labels are already pushed to the edges by the method described in section 6.2.2.

The below notion of continuity lets us figure out whether two histories are related as adjacent parts of a whole atomic history:

Definition 6.2.2 (Continuity)**.**

1. *Two atomic histories h_1 and h_2 are <u>continuous</u> ($\succ$) iff the initial messages of h_2 are in the live messages of h_1:*

$$h_1 \succ h_2 \triangleq \overline{im_1^{init}} \overset{h_1}{\rightsquigarrow\!\triangleright} \overline{im_1^{end}} \wedge \overline{im_2^{init}} \overset{h_2}{\rightsquigarrow\!\triangleright} \overline{im_2^{end}} \wedge \overline{im_2^{init}} \subseteq \overline{im_1^{end}}.$$

2. *We say h_1 and h_2 are <u>externally continuous</u> (denoted as $S \vdash h_1 \succ_{\mathsf{ext}} h_2$) if they are continuous and h_1 is externally atomic.*

3. *Two atomic histories h_1 and h_2 are <u>discontinuous</u> iff the live messages of h_1 are disjoint from the initial messages of h_2:*

$$h_1 \not\succ h_2 \triangleq \overline{im_1^{init}} \overset{h_1}{\rightsquigarrow\!\triangleright} \overline{im_1^{end}} \wedge \overline{im_2^{init}} \overset{h_2}{\rightsquigarrow\!\triangleright} \overline{im_2^{end}} \wedge \overline{im_1^{end}} \# \overline{im_2^{init}}.$$

Note that discontinuity requires that the live messages of the previous history are *disjoint* to the initial messages of the next history. Therefore, it is not true that two atomic histories are always either continuous or discontinuous. We will very soon explore how to deal with the other case, where the next history takes multiple sets of live messages from the previous histories.

From now on, throughout this chapter, we will have a number of definitions that use a sequence of atomic histories. Such a sequence $\overline{h}$ will be predicated as $\overline{\mathcal{A}}\,(\overline{h})$, with the following formal definition: $\overline{\mathcal{A}}\,(\overline{h}) \triangleq \forall h \in \overline{h}.\ (\overset{h}{\rightsquigarrow\!\triangleright})$.

Using the notion of continuity, we can also formalize interleaving of atomic histories:

Definition 6.2.3 (Interleaved Histories)**.** *In a given system S, a sequence of atomic histories $\overline{h}$ is <u>interleaved</u> iff there exist two histories h_1 and h_2 in the sequence that*

are externally continuous and any history between the two is discontinuous to h_1*:*

$$\begin{aligned}\mathsf{Interleaved}\ S\ \overline{h} \triangleq\ & \overline{\mathcal{A}}\,(\overline{h}) \wedge \\ & \exists h_1, h_2, \overline{h_1}, \overline{h_2}, \overline{h_3}. \\ & \overline{h} = \overline{h_1} + h_1 + \overline{h_2} + h_2 + \overline{h_3} \wedge \\ & S \vdash h_1 \succ_{ext} h_2 \wedge (\forall h' \in \overline{h_2}.\ h_1 \not\succ h').\end{aligned}$$

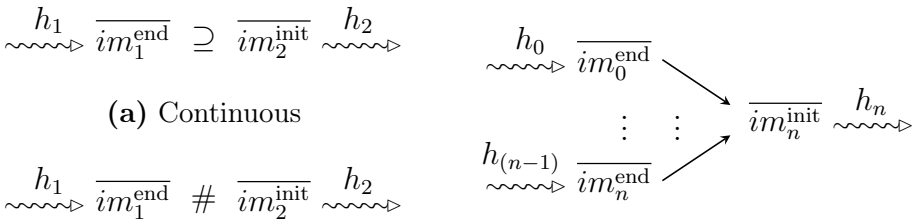
We also call a history h *interleaved, written* ($\mathsf{Interleaved}\ S\ h$)*, iff we can find an interleaved sequence of histories* $\overline{h_s}$ *that satisfies* $h = \oplus\overline{h_s}$*.*

There are two subtleties in defining the notion of interleaving precisely. First, we look for *externally continuous* histories, not just continuous ones, to define whether the whole history is interleaved or not. It does not restrict the notion since when continuous histories are found in a legal history, either they are already externally continuous or others are found at the beginning of the atomic-history chain that contains them. Second, we require that the earlier history (h_1) is *discontinuous* to each history ($\forall h' \in \overline{h_2}$) between the two externally continuous histories (h_1 and h_2). This is to ensure that the live messages generated by h_1 are preserved – i.e., not consumed by some other atomic histories – until h_2 takes its initial messages from the live messages.

Note that our interleaving example is indeed interleaved: we can find externally continuous histories [❶] and [❸;❹;❺], and [❶] is discontinuous with the only intermediate atomic history [❷].

One of the main intuitions to prove serializability is to categorize a given history. If the history is already sequential, we do not need to proceed further. If the history is interleaved, we will need to merge the externally continuous histories, which will decrease the degree of semi-sequentiality and lead to a sequential history. This categorization, however, implicitly assumes any history is either sequential or interleaved.

An additional condition is required, unfortunately, just to reason with sequential and interleaved histories. In other words, there is a third type of a history, happening when at least two different external atomic histories come together via a local state



(a) Continuous

(b) Discontinuous

(c) Confluent

Figure 6-6: Three relation types between atomic histories

transition consuming some live messages of these histories. Figure 6-6 depicts the three types of relations between two atomic histories. While atomic-history fragments in an interleaved history are only expected to be continuous or discontinuous to each other, there is another relation type, which is called *confluent* histories. This confluence case does not happen in any practical cache-coherence protocols, but we still need a formal predicate to ensure that the target system never generates such cases.

The *nonconfluence* predicate removes this case, by claiming that any atomic history is either the start of a new transaction or a continuation of a previous transaction:

Definition 6.2.4 (Nonconfluence)**.** *A system S is nonconfluent iff any first non-external history (if it exists) is interleaved (with an external atomic history):*

$$\begin{aligned}\mathsf{Nonconfluent}\ S \triangleq\ \forall \overline{h}, h_0, s.\ \ & \overline{\mathcal{A}}\,(\overline{h} + h_0) \wedge \\ & S_{\mathit{init}} \xRightarrow[S]{\oplus \overline{h} + h_0} s \wedge\ (\forall h \in \overline{h}.\ S \overset{h}{\rightsquigarrow}_{\mathsf{ext}}) \wedge\ \neg(S \overset{h_0}{\rightsquigarrow}_{\mathsf{ext}}) \rightarrow \\ & \mathsf{Interleaved}\ S\ (\overline{h} + h_0).\end{aligned}$$

6.2.4 Merging interleaved histories

In the last section, we examined the three types of histories and showed that the nonconfluence property rules out all confluent histories, so it is safe to reason only with interleaved histories. (Sequential histories are already good in terms of serializability.) In section 6.1 we provided our intuition that an interleaved history will be reduced to

a sequential history by repeatedly merging atomic-history fragments. In this section, we provide the formal definition of such merges and show that the nonconfluence and mergeability properties imply serializability.

As mentioned in section 6.2.3, the main purpose of merging two continuous histories is to decrease the degree of semi-sequentiality, so we have a less-interleaved history. Focusing solely on this purpose, we formally define the notion of mergeability as follows:

Definition 6.2.5 (Mergeability)**.** *For a given system S, two histories h_1 and h_2 are mergeable iff any history containing the two histories can be reduced to the one where the two histories are merged and the number of atomic-history fragments for the other histories is preserved:*

$$
\begin{aligned}
S \vdash h_1 \bowtie h_2 \triangleq \quad & \forall s_1.\ \ S \Rightarrow s_1 \rightarrow \\
& \forall \overline{h}, s_2.\ \ \overline{\mathcal{A}}\,(\overline{h}) \ \wedge\ s_1 \xRightarrow[S]{h_1 + \oplus\overline{h} + h_2} s_2 \rightarrow \\
& \exists \overline{h_l}, \overline{h_r}.\ \ \overline{\mathcal{A}}\,(\overline{h_l}) \ \wedge\ \overline{\mathcal{A}}\,(\overline{h_r}) \wedge \\
& \qquad s_1 \xRightarrow[S]{\oplus\overline{h_l} + h_1 + h_2 + \oplus\overline{h_r}} s_2 \ \wedge\ |\overline{h}| = |\overline{h_l}| + |\overline{h_r}|.
\end{aligned}
$$

Furthermore, we say the system S is mergeable *if any externally continuous histories in a legal history are mergeable:*

$$\mathsf{Mergeable}\ S \triangleq \forall h_1, h_2.\ S \vdash h_1 \succ_{ext} h_2 \rightarrow S \vdash h_1 \bowtie h_2.$$

The mergeability definition is abstract in that it relates the intermediate histories ($\overline{h}$) and the pushed histories ($\overline{h_{\mathrm{l}}}$ and $\overline{h_{\mathrm{r}}}$) not by mentioning they are indeed pushed from $\overline{h}$, but just by requiring *the number of atomic-history fragments are preserved* ($|\overline{h}| = |\overline{h_{\mathrm{l}}}| + |\overline{h_{\mathrm{r}}}|$). This condition implies that the number of atomic-history fragments is preserved after the merge. When h_1 and h_2 are continuous histories, however, by merging h_1 and h_2 we get the entire number of fragments decreased by 1, which means that the history after the merge has a smaller semi-sequentiality degree.

We are finally equipped with enough definitions to introduce a convenient general way to prove serializability:

Theorem 6.2.4. *If a system S is nonconfluent and mergeable, then it is serializable.*

$$\forall S.\ \mathsf{Nonconfluent}\ S \wedge \mathsf{Mergeable}\ S \rightarrow \mathsf{Serializable}\ S.$$

Proof. For a given history h, by Theorem 6.2.2 there exists a sequence of atomic histories $\overline{h_s}$ such that $\oplus\overline{h_s} = h$. In the sequence we try to find the first externally continuous histories. If we cannot find any, then by definition h is already sequential. If there exist such histories, by the definition of nonconfluence (Definition 6.2.4) they are interleaved. Now by mergeability (Definition 6.2.5) we can find a new sequence of histories $\overline{h_r}$ that reaches the same state, where the externally continuous histories are merged, thus $|\overline{h_r}| < |\overline{h}|$. Applying Theorem 6.2.2, we will eventually obtain a sequential history reaching the same state by repeating this process. ■

6.2.5 Reduction and pushes by separation

Theorem 6.2.4 presents a good intuition on how to prove serializability using nonconfluence and mergeability. The mergeability definition (Definition 6.2.5), however, is rather abstract; it does not provide any actual techniques to merge continuous histories. In this section, we explore a method to merge continuous histories, by using *state-transition separation* and *pushability*.

Reduction by separation

We begin with introducing the trivial reduction between two state transitions:

Theorem 6.2.5. *Two state transitions commute if they affect different objects and*

different input/output channels:

$$\begin{aligned}&\forall o_1, r_1, \overline{\mathrm{ins}}_1, \overline{\mathrm{outs}}_1, o_2, r_2, \overline{\mathrm{ins}}_2, \overline{\mathrm{outs}}_2.\\&\quad o_1 \neq o_2 \ \wedge\ (\overline{\mathrm{ins}_1.i} \cup \overline{\mathrm{outs}_1.i}) \# (\overline{\mathrm{ins}_2.i} \cup \overline{\mathrm{outs}_2.i}) \rightarrow\\&\quad l_{int}(o_1, r_1, \overline{\mathrm{ins}}_1, \overline{\mathrm{outs}}_1) \leftrightarrow l_{int}(o_2, r_2, \overline{\mathrm{ins}}_2, \overline{\mathrm{outs}}_2).\end{aligned}$$

For a given history h, we denote by $\mathsf{objs}\,(h)$ ($\mathsf{chns}\,(h)$) the set of object (channel) indices whose state is *used* by h, obtained by collecting the object index (the input/output channel indices) in each label in h. We can then lift the trivial reduction theorem to one for histories, using $\mathsf{objs}\,(h)$ and $\mathsf{chns}\,(h)$:

Theorem 6.2.6 (Reduction by trivial separation)**.** *Two histories commute if they affect different objects and different input/output channels:*

$$\forall h_1, h_2.\ \mathsf{objs}\,(h_1) \# \mathsf{objs}\,(h_2) \ \wedge\ \mathsf{chns}\,(h_1) \# \mathsf{chns}\,(h_2) \rightarrow h_1 \leftrightarrow h_2.$$

Theorem 6.2.6 is too trivial to be used in actual atomic histories generated by the Hemiola DSL. We thus relax this theorem with respect to object and message states, so it can be used in the actual serializability proof.

Firstly, instead of comparing object indices, $\mathsf{objs}\,(h_1)$ and $\mathsf{objs}\,(h_2)$, sometimes we would like to prove the commutativity of state transitions *within an object* directly. Given two internal labels $l_{\mathrm{int}}(O.i, r_1.i, \overline{\mathrm{ins}}_1, \overline{\mathrm{outs}}_1)$ and $l_{\mathrm{int}}(O.i, r_2.i, \overline{\mathrm{ins}}_2, \overline{\mathrm{outs}}_2)$ (created by the same object O), if we prove the following property then the two labels are commutative in terms of object state transitions:

$$\begin{aligned}\mathsf{ObjSep}\ r_1\ r_2 \triangleq\ &\forall o_0, \overline{\mathrm{ins}}_1, \overline{\mathrm{ins}}_2.\\&\mathsf{let}\ (o_1, \overline{\mathrm{outs}}_1) := r_1.\mathsf{trs}\ (o_0, \overline{\mathrm{ins}}_1)\ \mathsf{in}\\&\mathsf{let}\ (o_2, \overline{\mathrm{outs}}_2) := r_2.\mathsf{trs}\ (o_1, \overline{\mathrm{ins}}_2)\ \mathsf{in}\\&\mathsf{let}\ (o_1', \overline{\mathrm{outs}}_2') := r_2.\mathsf{trs}\ (o_0, \overline{\mathrm{ins}}_2)\ \mathsf{in}\\&\overline{\mathrm{outs}}_2 = \overline{\mathrm{outs}}_2' \ \wedge\ r_1.\mathsf{trs}\ (o_1', \overline{\mathrm{ins}}_1) = (o_2, \overline{\mathrm{outs}}_1).\end{aligned}$$

For instance, while using the Hemiola DSL, when an object forwards an upward request it sets an uplock. The object should be able to handle a downward request to it even after setting the uplock. These two state transitions are always commutative, since there is no state transition while forwarding the upward request (except locking), and handling the downward request does not affect the uplock.

This object separation is naturally extended to one for atomic histories:

Definition 6.2.6. *For given atomic histories h_1 and h_2, we say h_1 and h_2 are object-separated, denoted as* $(\mathsf{ObjSepHst}\ h_1\ h_2)$*, if the following conditions are satisfied:*

$$\begin{aligned}\mathsf{ObjSepHst}\ h_1\ h_2 \triangleq\ \ \forall l_1, l_2.\ \ & l_1 \in h_1\ \wedge\ l_2 \in h_2 \rightarrow \\ & \mathsf{let}\ l_{int}(O_1.i, r_1.i, \overline{\mathrm{ins}}_1, \overline{\mathrm{outs}}_1) := l_1\ \mathsf{in} \\ & \mathsf{let}\ l_{int}(O_2.i, r_2.i, \overline{\mathrm{ins}}_2, \overline{\mathrm{outs}}_2) := l_2\ \mathsf{in} \\ & O_1.i \neq O_2.i\ \vee\ \mathsf{ObjSep}\ r_1\ r_2.\end{aligned}$$

Note that each label l in h_1 (or h_2) is always an internal label, since it is an atomic history. The intuition here is that h_1 and h_2 are commutative in terms of object state transitions when each pair of respective label – representing object state transitions – in h_1 and h_2 are commutative either because they use different objects (separation) or by the commutativity conditions within the same object.

Next we consider message states. Instead of comparing channel indices regardless of their input/output types, we relax the notion to allow the live messages of h_1 and the initial messages of h_2 to be in the same channel but still different. We first define a couple of new notions for atomic histories:

Definition 6.2.7. *For given an atomic history h, we define the input and output messages of h, denoted as* $\mathsf{ins}\,(h)$ *and* $\mathsf{outs}\,(h)$*, respectively:*

$$\mathsf{ins}\,(h) \triangleq \oplus(\mathsf{List.map}\ (\lambda l.\ \mathsf{let}\ l_{int}(o, r, \overline{\mathrm{ins}}, \overline{\mathrm{outs}}) := l\ \mathsf{in}\ \overline{\mathrm{ins}.i})\ h).$$

$$\mathsf{outs}\,(h) \triangleq \oplus(\mathsf{List.map}\ (\lambda l.\ \mathsf{let}\ l_{int}(o, r, \overline{\mathrm{ins}}, \overline{\mathrm{outs}}) := l\ \mathsf{in}\ \overline{\mathrm{outs}.i})\ h).$$

As seen in the definition, $\mathsf{ins}\,(h)$ ($\mathsf{outs}\,(h)$) collects all the input (output) message

channels of the internal labels. We will also use $\mathsf{inits}\,(h)$ and $\mathsf{lives}\,(h)$ to refer to the initial messages and live messages of h, respectively.

Definition 6.2.8. *Given atomic histories h_1 and h_2, we say h_1 and h_2 are message-separated, denoted as* $(\mathsf{MsgSepHst}\ h_1\ h_2)$*, if the following conditions are satisfied:*

$$\begin{aligned}\mathsf{MsgSepHst}\ h_1\ h_2 \triangleq\ & \forall\, \overline{\mathrm{inits}}_1, \overline{\mathrm{lives}}_1, \overline{\mathrm{inits}}_2, \overline{\mathrm{lives}}_2. \\ & \overline{\mathrm{inits}}_1 \overset{h_1}{\rightsquigarrow\!\triangleright} \overline{\mathrm{lives}}_1\ \wedge\ \overline{\mathrm{inits}}_2 \overset{h_2}{\rightsquigarrow\!\triangleright} \overline{\mathrm{lives}}_2 \rightarrow \\ & \mathsf{ins}\,(h_1)\ \#\ \mathsf{ins}\,(h_2)\ \wedge\ \mathsf{ins}\,(h_1)\ \#\ \mathsf{outs}\,(h_2)\ \wedge \\ & \overline{\mathrm{lives}}_1\ \#\ \overline{\mathrm{inits}}_2\ \wedge\ \mathsf{outs}\,(h_1)\ \#\ \mathsf{outs}\,(h_2).\end{aligned}$$

Instead of having $\mathsf{chns}\,(h_1)\ \#\ \mathsf{chns}\,(h_2)$ as a disjointness condition, we see a relaxation between $\overline{\mathrm{lives}}_1$ and $\overline{\mathrm{inits}}_2$ to require disjointness not only by channel indices, but also by message values.

This relaxation is indeed required to prove serializability in Hemiola. We can find an instance from our interleaving example, again, specifically from Figure 6-3a. It is the case where the atomic history [6] happens first, generating rql to the channel 5 as the live message, and the atomic history [7] happens next, consuming rsM from the same channel. These two histories commute, but if we just use Theorem 6.2.6 with the trivial conditions, we cannot prove commutativity. We can employ the relaxed condition (Definition 6.2.8), though, since rql and rsM are different messages, thus the relaxed condition $\overline{\mathrm{lives}}_1\ \#\ \overline{\mathrm{inits}}_2$ holds.

Using the new, relaxed object and message separations, we present the following reduction theorem:

Theorem 6.2.7 (Reduction by separation)**.**

$$\forall h_1, h_2.\ (\overset{h_1}{\rightsquigarrow\!\triangleright})\ \wedge\ (\overset{h_2}{\rightsquigarrow\!\triangleright})\ \wedge\ \mathsf{SepHst}\ h_1\ h_2 \rightarrow h_1 \leftrightarrow h_2,$$

where $\mathsf{SepHst}\ h_1\ h_2 \triangleq \mathsf{ObjSepHst}\ h_1\ h_2\ \wedge\ \mathsf{MsgSepHst}\ h_1\ h_2$.

Proof. The proof is quite straightforward by nested induction on $(\overset{h_1}{\rightsquigarrow\!\triangleright})$ and $(\overset{h_2}{\rightsquigarrow\!\triangleright})$. The base case is to prove $l_1 \leftrightarrow l_2$ when $h_1 = [l_1]$ and $h_2 = [l_2]$. Suppose $l_1 =$

$l_{\text{int}}(O_1.i, r_1.i, \overline{\text{ins}_1}, \overline{\text{outs}_1})$ and $l_2 = l_{\text{int}}(O_2.i, r_2.i, \overline{\text{ins}_2}, \overline{\text{outs}_2})$. Then we obtain the object separation (ObjSep r_1 r_2) from ($\mathsf{ObjSepHst}$ h_1 h_2). We also obtain $\overline{\text{ins}_1.i} \# \overline{\text{ins}_2.i}$, $\overline{\text{ins}_1.i} \# \overline{\text{outs}_2.i}$, $\overline{\text{outs}_1} \# \overline{\text{ins}_2}$, and $\overline{\text{outs}_1.i} \# \overline{\text{outs}_2.i}$ from ($\mathsf{MsgSepHst}$ h_1 h_2). These conditions are enough to prove commutativity between l_1 and l_2 just by construction. Induction cases are straightforward as well, e.g., if $h_1 = h_1' + l_1$ we perform reduction between l_1 and h_2 first and do another reduction between h_1' and h_2 by the induction hypothesis to perform the whole reduction between h_1 and h_2. ■

Pushes by separation

Now we provide our method to merge continuous histories, by pushing all intermediate atomic-history fragments outside of the continuous histories. Pushing an intermediate atomic-history fragment will require a finite number of history reductions, and we will use Theorem 6.2.7 for each one.

Suppose that we want to merge continuous histories h_1 and h_2 through the intermediate atomic histories $\overline{h}$. We first categorize each atomic history into two groups: *left-pushable* histories and *right-pushable* histories, with the following requirements:

1. h_1 and a left-pushable history are always commutative:

$$\forall h \in \overline{h}.\ \mathsf{LeftPushable}\ h \to h_1 \leftrightarrow h.$$

2. A right-pushable history and h_2 are always commutative:

$$\forall h \in \overline{h}.\ \mathsf{RightPushable}\ h \to h \leftrightarrow h_2.$$

3. A right-pushable history and a left-pushable one are always commutative:

$$\forall h_r, h_l \in \overline{h}.\ \mathsf{RightPushable}\ h_r \to \mathsf{LeftPushable}\ h_l \to h_r \leftrightarrow h_l.$$

These three conditions are enough to merge h_1 and h_2.

Theorem 6.2.8. *If each $h \in \overline{h}$ can be categorized in terms of left/right pushability, then $S \vdash h_1 \bowtie h_2$ holds.*

Proof. We first try to find the first left-pushable history in $\overline{h}$. If it does not exist, then all the atomic-history fragments in $\overline{h}$ are right-pushable. In this case $\oplus\overline{h} \leftrightarrow h_2$ holds by repeatedly commuting the last fragment of $\overline{h}$ and h_2. If the first left-pushable history exists, suppose $\overline{h} = \overline{h_r} + h_l + \overline{h_e}$, where h_l is the first left-pushable history. By its definition, each history in $\overline{h_r}$ is right-pushable, thus we can prove $(h_1 + \oplus\overline{h_r}) \leftrightarrow h_l$, i.e., h_l can be pushed before h_1. We will eventually have intermediate histories that are all right-pushable, by repeatedly pushing the first left-pushable history before h_1. Once we obtain such histories, we can push them after h_2 finally to merge h_1 and h_2. ■

How can we establish such pushability conditions? A naive approach is to use the object and message separation conditions introduced in Definition 6.2.6 and Definition 6.2.8:

$$\mathsf{LeftPushable}\ h \triangleq \mathsf{SepHst}\ h_1\ h.$$

$$\mathsf{RightPushable}\ h \triangleq \mathsf{SepHst}\ h\ h_2.$$

We get the first and second conditions of pushability with these definitions by simply applying Theorem 6.2.7. The third condition, however, is nontrivial, requiring the following property:

$$\forall h_l, h_r.\ \mathsf{SepHst}\ h_1\ h_l \rightarrow \mathsf{SepHst}\ h_r\ h_2 \rightarrow h_l \leftrightarrow h_r.$$

This is not provable by itself, since there are no conditions relating h_1 and h_2. Thus in our serializability proof we define left/right pushability in a more restricted way. We will introduce such restricted pushability definitions right in the next section.

6.2.6 All together: the serializability proof

We are now equipped with enough formalization and ready to prove the serializability guarantee in Hemiola. In this section, we provide the serializability proof in Hemi-

ola. Recalling the requirements from chapter 5, a protocol should be defined on a tree topology ($\mathsf{OnTree}\ S\ t$), and each rule should be defined using the rule templates ($\mathsf{GoodRules}\ S\ t$). Throughout the section, we will deal with a system S with these conditions. In order to prove serializability, we will use Theorem 6.2.4, i.e., nonconfluence and mergeability will be proven for the protocol defined with the Hemiola DSL.

Both the nonconfluence and mergeability proofs require a number of invariants that apply to any protocol satisfying ($\mathsf{OnTree}\ S\ t$) and ($\mathsf{GoodRules}\ S\ t$). All the invariants mentioned in the proof should be provable by inducting on state-transition steps, where each transition step requires a case analysis by rule template.

The very first invariant is to categorize the input messages of each internal label:

Invariant 6.1. *Input messages of each internal label always belong to one of four categories: 1) single request to the parent, 2) single response to a child, 3) single request to a child, or 4) multiple responses from children.*

Nonconfluence is easier to prove than mergeability, simply by looking at the initial messages of the first non-external atomic history:

Theorem 6.2.9 (Nonconfluence in Hemiola)**.**

$$\forall S, t.\ \mathsf{OnTree}\ S\ t \wedge \mathsf{GoodRules}\ S\ t \rightarrow \mathsf{Nonconfluent}\ S.$$

Proof of nonconfluence Following the definition of nonconfluence, let $\overline{h_e}$ be the sequence of externally atomic histories before the first non-external atomic history h_i. Since each history in $\overline{h_e}$ takes external input request messages, it is trivial that any two histories in $\overline{h_e}$ are discontinuous to each other. Now considering the first non-external atomic history in the definition of nonconfluence, the initial messages are internal and thus belong to one of the four categories presented in Invariant 6.1.

For the first three, the initial message is just a singleton. Since the history $(\oplus\overline{h_e} + h_i)$ is legal, at the time the initial message is consumed by h_i it is in the semantic

message state. It is straightforward that the semantically live messages before h_i is executed are all generated by $\overline{h_e}$; furthermore, we can prove that any live message before h_i belongs to the live messages of an external atomic history in $\overline{h_e}$. Thus we should be able to find an external atomic history, say $h_e^0 \in \overline{h_e}$, whose live messages contain the initial message. $(S \vdash h_e^0 \succ_{\mathsf{ext}} h_i)$ trivially holds by h_e^0 generating the initial message of h_i. Therefore (Interleaved S $(\oplus\overline{h_e} + h_i)$) holds by definition.

The last category (multiple responses from children) is nontrivial, since there is a chance that one part of the messages is generated by an external atomic history and the other part is from another history. If we prove that the messages are from a single external atomic history, nonconfluence is proven similarly to the first three cases. The idea comes from looking at the downlock associated with the responses.

Suppose that an external atomic history h_e outputted requests to children. It is trivial that h_e is continuous to any (non-external) atomic history that takes one of the children requests as the initial message. Now according to what is specified by the designated rule template for making requests to children (by GoodRules S t), the parent *sets a downlock and does not accept any other atomic histories* to make another set of requests to the children. We can use this fact to prove that any response to the parent after making the child requests is in the live messages of a non-external atomic history that took the corresponding request, since no other ongoing atomic histories can pass through the parent by the downlock. Let h_j^c be an atomic history that generates a response, where $j = 0, 1, \cdots$ matches each corresponding child request.

Since $\overline{h_e}$ does not contain any non-external atomic histories, the only possible case is that h_e and $\overline{h_j^c}$ are merged together to form a single external atomic history, i.e., $h_e^c \triangleq (h_e + \sum_j h_j^c)$ itself is the external atomic history whose live messages contain all the responses to the parent. By definition it is trivial to prove $(S \vdash h_e^c \succ_{\mathsf{ext}} h_i)$, and the rest of the proof is same as the other three cases. *(End of the nonconfluence proof)* ■

The mergeability proof is trickier, requiring a number of additional invariants about locks and messages. The intuition is already provided in section 6.1; when merging two atomic-history fragments, say h_1 and h_2, we will specifically look at the initial messages of h_2 (generated by h_1) as well as the locks associated with the initial

messages. The messages and the locks will tell us which atomic histories can pass the object to make state transitions through it.

The proof will use the following simple invariant about messages residing in the system:

Invariant 6.2. *There are no two messages with the same message type in the same channel. A downward request and response may be in the same channel, but always the response comes to the channel first.*

Theorem 6.2.10 (Mergeability in Hemiola)**.**

$$\forall S, t.\ \mathsf{OnTree}\ S\ t \wedge \mathsf{GoodRules}\ S\ t \rightarrow \mathsf{Mergeable}\ S.$$

Proof of mergeability Following the definition of mergeability, let h_1 and h_2 be the externally continuous histories ($S \vdash h_1 \succ_{\mathsf{ext}} h_2$) and $\overline{h}$ be the intermediate histories between the two. As already observed and used in the nonconfluence proof (Theorem 6.2.9), the initial messages of h_2 (i.e., $\mathsf{inits}\,(h_2)$) belong to the four categories: an upward request, a downward response, a downward request, or multiple upward responses (to the parent).

In addition to this categorization, in order to prove mergeability, we need to categorize the live messages of h_1 (i.e., $\mathsf{lives}\,(h_1)$) as well, which is phrased as an invariant:

Invariant 6.3. *The live messages of an external atomic history belong to one of the following three categories: 1) an upward request, 2) a downward response, or 3) multiple downward requests and upward responses.*

Mergeability will be proven for each combination of the live messages (of h_1) and the initial messages (of h_2), constrained by $\mathsf{inits}\,(h_2) \subseteq \mathsf{lives}\,(h_1)$ due to continuity. The following case analyses are by the live messages of h_1.

1) *An upward request*: since $\mathsf{inits}\,(h_2) \subseteq \mathsf{lives}\,(h_1)$, $\mathsf{inits}\,(h_2)$ can at most contain an upward request, and due to the categorization of $\mathsf{inits}\,(h_2)$ it should be a single

upward request as well. We need to prove another invariant about upward requests:

Invariant 6.4. *If an atomic history generates an upward request as its live message, it only consists of internal labels generated by the "rquu" rule template, i.e., taking an upward request from a child and forwarding it to the parent.*

This invariant is proven by inducting on atomic-history steps, not on ordinary state-transition steps.

For each case, we will apply Theorem 6.2.8 to merge h_1 and h_2, which requires defining left/right pushability. In this upward-request case, we set the pushability as follows:

$$\mathsf{LeftPushable}\ h \triangleq \mathsf{True}. \quad \mathsf{RightPushable}\ h \triangleq \mathsf{False}.$$

These definitions simply say that any atomic-history fragment between h_1 and h_2 is always left-pushable. Since nothing is declared to be right-pushable, we only need to prove the first condition of pushability, i.e., $\forall h \in \overline{h}.\ h_1 \leftrightarrow h$.

We apply Theorem 6.2.7 to prove it. The only information we have about h_1 is that any internal label in h_1 only sets an uplock (without any other state transition) and requests to the parent. In that sense, the object separation ($\mathsf{ObjSepHst}\ h_1\ h$) is straightforward, since h cannot affect any uplocks set by h_1. The message separation ($\mathsf{MsgSepHst}\ h_1\ h$) is obvious as well, since h cannot use any upward-request channels used by h_1 due to the uplocks made by h_1.

2) *A downward response*: by the same reasoning, $\mathsf{inits}\,(h_2)$ should be a single downward response. We need two dual invariants to prove this case.

Invariant 6.5. *If an atomic history h consumes a downward response to the object O as the initial message, the object state transitions performed by h are constrained to occur only within the subtree with the root O (denoted as $tr\,(O)$).*

Invariant 6.6. *If an atomic history h consumes a downward response to the object O as the live message, then there exist h_l and h_r such that $h = h_l + h_r$, where h_l only consists of internal labels generated by the "rquu" rule template and h_r is constrained to act only within the complement of the subtree with the root O (denoted as $tr^{-1}\,(O)$).*

Now considering the sequence of atomic histories $\oplus\overline{h}$, we decide whether a history $h \in \overline{h}$ is left- or right-pushable, by looking at the object state transitions performed by it. An important fact here is that since the downward response to the object O resides in the system after h_1, no intermediate history can perform state transitions both in $\mathrm{tr}\,(O)$ and $\mathrm{tr}^{-1}\,(O)$. More specifically, if an intermediate history is within $\mathrm{tr}^{-1}\,(O)$, it can have a downward request to O (by Invariant 6.2 it cannot have a downward response) as a live message, regarded as the closest effect to O. We can still use the message separation in Theorem 6.2.7, since the downward response and a possible downward request by an intermediate history are different messages, determined by their message types.

Now we can define the following pushability definitions for the downward-response case:

$$\begin{aligned} \mathsf{LeftPushable}\ h &\triangleq \exists O_h \in \mathsf{objs}\,(h).\ O_h \in \mathrm{tr}\,(O). \\ \mathsf{RightPushable}\ h &\triangleq \exists O_h \in \mathsf{objs}\,(h).\ O_h \in \mathrm{tr}^{-1}\,(O). \end{aligned}$$

The first pushability condition ($\forall h \in \overline{h}.\ \mathsf{LeftPushable}\ h \rightarrow h_1 \leftrightarrows h$) is proven in two steps: to prove commutativity between h_r and h first ($h_r \leftrightarrows h$) and then prove it between h_l and h ($h_l \leftrightarrows h$). The former is provable by using the fact from Invariant 6.6 that h_r is constrained within $\mathrm{tr}^{-1}\,(O)$. Now Theorem 6.2.7 will be enough to prove the commutativity. The latter is already proven in the case (1) for upward requests; again from Invariant 6.6 h_l is like h_1 in the case (1).

The second condition ($\forall h \in \overline{h}.\ \mathsf{RightPushable}\ h \rightarrow h \leftrightarrows h_2$) is proven very similarly but in the opposite way. From Invariant 6.5 we get that h_2 is constrained within $\mathrm{tr}\,(O)$. Since h is constrained within $\mathrm{tr}^{-1}\,(O)$, we can easily prove the object separation in Theorem 6.2.7. For message separation, the only corner case is when h generates a downward request to O. This case is still covered by the message-separation condition, since it requires the separation of messages ($\overline{\mathrm{lives}}_1\ \#\ \overline{\mathrm{inits}}_2$), not the separation of channel indices.

By these pushability definitions, the third condition is also proven naturally by applying Theorem 6.2.7. Let us call O the *separation point* in the sense that O

separates possible state transitions by an intermediate history. This concept will be used in the other cases as well.

3) *Multiple downward requests and upward responses*: this case is the most complex one in terms of setting the correct pushability conditions. We first define a new notion to define the common ancestor of the downward requests and upward responses:

Invariant 6.7. *If an atomic history h has multiple downward requests and upward responses as its live messages, there exists a unique common ancestor of the messages (recall* OnTree *S t, which says that S is defined on the tree topology), called the root of downward requests and upward responses, which has a downlock set by an upward request. (Note that there are two ways to make a downlock: one by an upward request and the other by a downward request.)*

Let O_r be the root of the live messages. Suppose that the downward requests are to the set of objects $\{O_i\}_{i\in I}$ with an index set I. Similarly, suppose that the upward responses are from the set of objects $\{O_j\}_{j\in J}$ with an index set J. Another invariant that can be proven while proving Invariant 6.7 is that I and J are disjoint.

Similarly to the case (2), h_1 can be decomposed into the upward/downward-request labels (no state transitions except locking) and all the other labels, and any intermediate atomic history is left-pushable before such request labels. Therefore, the following invariant about the coverage of h_1 will not mention those object state transitions as part of the constraint:

Invariant 6.8. *Borrowing the notions of the root and the index sets mentioned above, h_1 is constrained within $tr^{-1}(O_r) \cup (\bigcup_{j\in J} tr(O_j))$.*

Considering $\mathsf{inits}(h_2) \subseteq \mathsf{lives}(h_1)$ and the categorization of the initial messages, we get h_2 initiates with either a downward request or multiple upward responses. Based on this, we provide two invariants about the coverage of h_2:

Invariant 6.9. *If h_2 consumes a downward request to O_i with $i \in I$, it is constrained within $tr(O_r) - (\bigcup_{j\in J} tr(O_j))$. If h_2 consumes upward responses, then by the nonconfluence property (Theorem 6.2.9) the responses should be from $\{O_j\}_{j\in J}$ and $I = \varnothing$; it is also constrained within $tr(O_r) - (\bigcup_{j\in J} tr(O_j))$.*

Now we finally define the following pushability definitions for this downward-requests-upward-responses case:

$$\mathsf{LeftPushable}\ h \triangleq \exists O_h \in \mathsf{objs}(h).\ O_h \in \mathrm{tr}(O) \ \wedge\ O_h \notin (\textstyle\bigcup_{j\in J}\ \mathrm{tr}(O_j)).$$

$$\mathsf{RightPushable}\ h \triangleq \exists O_h \in \mathsf{objs}(h).\ O_h \in \mathrm{tr}^{-1}(O) \ \vee\ O_h \in (\textstyle\bigcup_{j\in J}\ \mathrm{tr}(O_j)).$$

The proof of the pushability conditions is then very similar to the one in case (2), by checking whether each separation meets the conditions in Theorem 6.2.7.

(End of the mergeability proof) ■

Finally we combine all the theorems together to obtain the serializability guarantee in Hemiola:

Theorem 6.2.11 (Hemiola's serializability guarantee)**.**

$$\forall S, t.\ \mathsf{OnTree}\ S\ t \wedge \mathsf{GoodRules}\ S\ t \rightarrow \mathsf{Serializable}\ S,$$

Proof. Applying Theorem 6.2.4, it suffices to prove (Nonconfluent S) and (Mergeable S). Theorem 6.2.9 provides the former, and Theorem 6.2.10 provides the latter. ■

Chapter 7

Related Work I: Approaches to Dealing with Interleavings

In chapter 4, we have formalized serializability as a correctness criterion for safe interleavings. We also demonstrated that the serializability property eases the burden of designing and proving conventional invariants required to prove trace refinement as a correctness criterion. In chapter 5 and chapter 6, we provided a domain-specific language that guarantees serializability automatically and proved the guarantee using commuting reductions.

It is worth emphasizing again that the use of serializability is not mandatory to verify distributed protocols; it is rather a proof strategy that can make the rest of the proof much easier. Before moving on to the next part of dissertation – the actual use of serializability in case studies – we would like to explore some other approaches to dealing with interleavings, not bounded to the verification of cache-coherence protocols.

Noninterference lemmas One of the methods to ensure safe interleavings is to claim that other execution units (concurrent threads, transactions, etc.) do not affect desired (pre)conditions to execute a unit. *Noninterference* refers to such a property, where its definition and form vary by the target computer system to verify.

We first look into a case in concurrent software; to our knowledge, the very first

attempt to describe such properties occurs in the foundational paper by Owicki and Gries:

> In [62] on page 324:
>
> "The key word is of course 'interfere'. One possibility to obtain noninterference is not to allow shared variables, but this is too restrictive. A more useful rule is to require that certain assertions used in the proof $\{P_i\}S_i\{Q_i\}$ of each process are left invariantly true under parallel execution of the other processes."

This paper extends the conventional Hoare logic [33] to reason about parallel (concurrent) execution of sequential programs. Here the noninterference property is described implicitly, as a requirement that the assertions used in the Hoare triple are not interfered with by the concurrent execution of the other processes. The paper later defines noninterference formally, by providing sufficient conditions that any execution of a statement never interferes with a given Hoare-triple proof. In this case, each property is presented as a specific lemma proving the noninterference conditions.

The use of noninterference in Hoare logic has been made more explicit later by several variants like concurrent separation logic (CSL) [57] and rely-guarantee reasoning [38]. CSL provides the "Critical Region Rule" to reason about interaction among processes via a shared resource [10]:

$$\frac{\{(P * RI_r) \wedge B\}\, C\, \{Q * RI_r\}}{\{P\}\, \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\, \{Q\}}$$

Here r is the shared resource, acting like a lock to enter critical regions in a program. This rule requires defining a resource invariant RI_r, stating desired properties when a process holds r. Mutual exclusion by a shared lock is one typical way to ensure noninterference. The uplock and downlock used in Hemiola function similarly. For example, acquiring a downlock means entering a "subtree" section (although not entirely critical) whose root is the downlocked object; a transaction holding the downlock iterates through the subtree, makes state transitions, and releases it when getting out of the subtree.

The most basic rule in CSL is the "Parallel Composition Rule," which composes completely parallel processes [10]:

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \;\cdots\; \{P_n\}\, C_n\, \{Q_n\}}{\{P_1 * \cdots * P_n\}\, C_1 \parallel \cdots \parallel C_n\, \{Q_1 * \cdots * Q_n\}}$$

It is interesting to compare this rule with how we use predicate messages in Hemiola for a single transaction on top of serializability. There is still concurrency in executing a single transaction; e.g., after a parent cache makes invalidation requests to children, each child (subtree) handles its request concurrently. Since the child subtrees are disjoint, this kind of execution is completely parallel, and indeed resulting predicate messages for corresponding invalidation responses deal with disjoint parts of the system state. In the sense of dealing with trivial concurrency, we would like to note that using predicate messages for input messages to prove the case for an output message is quite similar to the Parallel Composition Rule in CSL.

On the other hand, in rely-guarantee reasoning, two binary relations, called the "rely" and "guarantee" conditions, are added to the Hoare triple to reason about concurrent programs. The judgement has the form of $R, G \vdash \{P\}\, C\, \{Q\}$, meaning that every state change by another thread is in R and every state change by C is in G. ($\{P\}\, C\, \{Q\}$ has the same meaning from the conventional Hoare triple.) Noninterference is then exhibited when constructing $R, G \vdash \{P\}\, C\, \{Q\}$ from the original Hoare triple $\{P\}\, C\, \{Q\}$, requiring that P and Q are stable (not interfered with) under R, i.e., $\forall s, t.\ P(s) \wedge R(s, t) \rightarrow P(t)$.

Now we explore the use of noninterference in hardware verification, more related to the techniques used in the Hemiola framework. To our knowledge, the notion was mentioned for the first time in a paper by McMillan as a part of compositional model checking:

> In [47] on page 234:
>
> "We can also verify the design for an arbitrary number of execution units. To do this, as one might expect, we split the result lemma into cases based on the execution unit used to produce the result, eliminating all

> the other units. This, however, requires introducing a 'noninterference lemma'. This states that no other execution unit spuriously produces the result in question."

In model checking, noninterference properties are usually represented as lemmas (invariants) described in linear temporal logic [34], and the lemmas are specific to a hardware design to verify. The use of noninterference lemmas has been later systematized to the CMP method [47, 51, 16], for model checkers to use the lemmas to reduce state space to be explored.

Commuting reduction Another way to ensure safe interleavings is to prove commutativity between two adjacent state transitions representing partial executions by two different execution units, called *(commuting) reductions* [44].

CIVL [32] employed reductions and mover types [30] to verify concurrent garbage collection. While the CIVL verifier also supports noninterference and rely-guarantee, the authors reported that the use of commutativity is optional but beneficial in their experience. CSPEC [12] is another framework, embedded in the Coq proof assistant, that also uses mover types to verify a concurrent mail server. The framework also provides a library of reusable proof patterns, easing commutativity proofs.

Reductions have been employed in verifying distributed systems as well. IronFleet [31] employed refinement layers and reductions to verify a Paxos-based [40] replicated state machine and a shared key-value store. Reductions were particularly used to deal with interleavings by event handlers in the host implementation; once the interleaved execution is fully reduced, a refinement to the distributed-protocol layer can be proven directly.

Hemiola also employs commuting reductions and the techniques mentioned above to prove serializability for any protocol defined by the Hemiola DSL. While not exhibiting mover types in protocol descriptions, a similar concept is used in the serializability proof under the name of left/right pushability, explained in section 6.2.5.

Part III

Design, Proof, Implementation, and Synthesis of Hierarchical Cache-Coherence Protocols

Chapter 8

Case Studies: Hierarchical MSI and MESI Protocols

In this chapter we specify, implement, and formally prove the correctness of the following three hierarchical cache-coherence protocols: an inclusive MSI protocol, a noninclusive MSI protocol, and a noninclusive MESI protocol. Each protocol is parameterized by a tree t that decides the topology of the memory subsystem S, thus the system naturally satisfies $\mathsf{OnTree}\ S\ t$. Cache objects in each protocol are defined using the rule templates (defined in section 5.3) thus the system satisfies $\mathsf{GoodRules}\ S\ t$ by construction. These predicates imply that each protocol satisfies the serializability property, proven in Theorem 6.2.11. We will see how Hemiola helps implement and prove the protocols by taking full advantage of serializability.

8.1 Design Principles

We first present the common design principles shared by all the case studies.

8.1.1 Topology as a parameter

Each design is parameterized by a tree t that decides the topology of the memory subsystem. In other words, whenever we instantiate the tree parameter, we get a

cache-coherence design and its correctness proof for free. For example, the following tree definition will generate a cache-coherence protocol for four L1 caches, two L2 caches, the last-level cache (LLC), and the main memory:

```
Definition t: tree := Node [Node [Node [Leaf; Leaf]; Node [Leaf; Leaf]]].
```

There are three different kinds of caches in this topology-parameterized protocol. First of all, there are L1 caches (denoted as L_1) that correspond to leaf nodes in the tree. Symmetrically, each uses the same set of rules. The second kind is the last-level cache (LLC), which is the only one attached to the main memory, the root of the tree. It is possible to design multiple LLCs attached to the main memory in Hemiola, but our case studies follow standard practice in sticking to a single LLC. All the other caches between the L1 caches and the LLC are called intermediate caches (denoted as L_i), and they share a common set of rules as well.

8.1.2 Design and proof per-line

Each case-study protocol is defined just for a single cache line first and naturally extended to all cache lines using a protocol compiler that will be introduced in section 9.1. This approach is reasonable in terms of correctness, since a transaction does not affect coherence for lines other than its own. Consider the "duplicated" protocol first, where each cache line, its status, a directory entry, communication channels, and a lock holder are all duplicated per-line. It is infeasible to extend the protocol literally in this way, since we cannot require physically distinct channels and lock holders for all cache lines. The protocol compiler restricts the resources (e.g., channels, lock holders, etc.) to make the implementation hardware-synthesizable. Note that in this sense the duplicated protocol can be regarded as the most general multi-line design, whose behaviors can cover all the behaviors of compiled implementations (see section 9.2 for details).

8.1.3 Nondeterministic invalidation/eviction

Each protocol initiates invalidations and evictions nondeterministically. In other words, there are rules in each cache that can be executed even without being triggered by input messages, to make invalidation requests to the child caches or to make an eviction request to the parent. This design choice is certainly not realistic, but it always has more behaviors than any design with a specific invalidation/eviction policy, thus in terms of correctness a refinement to the specific design is trivial.

For instance, a number of practical cache-coherence protocols manage a data structure (a cache) to keep track of the least-recently-used (LRU) cache line per set (lines with the same index) [59]. Use of such a data structure and the decision algorithm are irrelevant to the correctness of a protocol. In other words, a protocol with the LRU replacement policy, regardless of its implementation, always has fewer behaviors than the one with nondeterministic eviction.

Another instance is a back-invalidation policy used in an inclusive cache. Back invalidation is necessary to maintain the cache inclusion among the parent and child caches and may happen right before evicting a parent cache line. There is another practical policy, called self-invalidation [68], where some voluntary back invalidations happen to increase performance. Similar to the case of cache replacement, a protocol with nondeterministic invalidation includes all the behaviors by the one with a specific invalidation policy.

8.1.4 Directory-based coherence

Each protocol uses a directory structure to ensure coherence, introduced in section 2.3. In our designs, each node with children has its own directory structure to track their statuses. The directory holds sound information about the status of each child *subtree*. For example, for a certain cache line, if an L1 cache L_1 has M status for the line, then all the ancestors (including the main memory) of L_1 have the directory status M pointing to the child subtree that contains L_1.

8.1.5 Noninclusive-cache inclusive-directory structure

Our noninclusive protocols employ the noninclusive-cache inclusive-directory (NCID) [82] structure to optimize the cache space. As explained in section 2.3, in noninclusive protocols the parent cache does not have to contain all the line values that children have, and back invalidations are not required to evict a line.

Measuring performance among various cache-inclusion policies is beyond the scope of this thesis. That said, we choose noninclusive caches as part of our case studies to demonstrate that Hemiola is general enough to design and prove various cache-coherence protocols, where specifically the noninclusive caches are the ones that most previous work had difficulty dealing with properly.

8.2 The MSI Protocol

The MSI protocol is known as a base cache-coherence protocol that can be optimized to more sophisticated protocols like MESI, MOSI, etc. Even though it is a base protocol, there are a lot of nontrivial cases that require deep understanding of the protocol itself and the nature of distributed protocols, especially in hierarchical protocols. In this section we design two hierarchical, directory-based MSI protocols, one with an inclusive cache-inclusion policy and the other with a noninclusive policy. As explained in section 8.1, the description and the correctness proof are for a single cache line, parameterized by a tree deciding the system topology.

8.2.1 Protocol description

Cache states

A cache state has the form $O(\mathsf{st}, \mathsf{v}, \mathsf{dir}, \mathsf{owned})$, consisting of a status, a value, a directory, and a Boolean called an *ownership bit*.

A status is either $\mathsf{M}(=3)$, $\mathsf{S}(=2)$, $\mathsf{I}(=1)$, or the "Not Present" status ($\mathsf{NP}=0$), encoded by natural numbers. NP means that the line does not exist in the cache object. In our case-study protocols we distinguish I and NP to avoid unnecessary

line creation when invalidating the line [55]. That is, when the cache object gets an invalidation request, if the line status is NP, then it should not change its status to I but maintain the NP status.

A directory contains a status of its children called a *directory status* and a list of child-cache indices that have the directory status. We will denote the directory like $\mathsf{S}_{\langle 1,2\rangle}$, saying that the directory status is S and child subtrees with indices 1 and 2 may contain caches with S status.

The ownership bit is to determine whether the cache is responsible for writing the value back to the parent when invalidated. When a cache has the M status, the ownership bit is always true. However, when the cache has S, the ownership bit can be either true or false. Note that L_1 does not have a directory since it has no children. It also does not have an ownership bit, since it does not have a case where it has S status but is responsible for writing the value back.

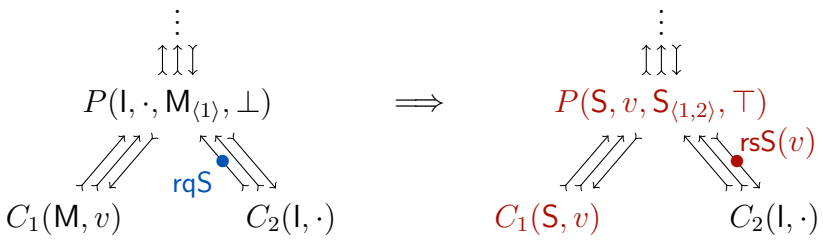


Figure 8-1: An ownership bit set in a shared-state cache

Figure 8-1 presents an example of a shared-state cache having its ownership bit set in a hierarchical memory subsystem. It could happen when an L1 cache C_2 wants to get S while another L1 cache C_1 has M. In this case, C_2 sends rqS and waits for the response with the value rsS(v). After a few steps, the value is pulled from C_1, and the parent P gets the data as well, entering a shared status with ownership bit set. Here the bit says that the shared value might be dirty, so it should be written back.

The ownership bit intuitively constrains which caches can have valid status. In the figure, all the other caches outside P, after pulling a value from C_1, are in the invalid status, since previously C_1 had the dirty value with M. We will see how this

intuition is used for the correctness proof soon in section 8.2.2.

Protocol description with rule templates

We present a number of rule descriptions in the MSI protocols that employ the rule templates provided in Hemiola. Each rule template is defined in Coq, taking in several parameters and generating a rule. We exploited Coq's notation mechanism to have a compact definition for each rule template.

```
1 Definition l1GetSRqUpUp: Rule :=
2   rule.rquu
3   :accepts getRq
4   :from cidx
5   :me oidx
6   :requires (fun ost msgIn => ost#[status] ≤ msiI)
7   :transition (fun ost msgIn => <| rqS; 0 |>).
```

Figure 8-2: L_1 requesting S to its parent

Figure 8-2 presents an actual rule definition, starting with an invocation of a particular rule template `rule.rquu`, which takes a request from one of its children and sends a further request to its parent. An L1 cache does not have any children; in this case `cidx` will be instantiated to an abstract node referring to the external interface (i.e., processor core) for it. Each line starting with a colon (`:`) provides more information about the rule. This rule `accepts` a message with the ID `rqS` `from` the child with the index `cidx`. This rule template also requires to write down which object (cache) it belongs to (`me`), the precondition (`requires`), and the state-transition function (`transition`). The output `rqS` message that we generate carries no data value, so we pair it with a dummy zero value.

Each rule template has different (Coq) types for a precondition and a state-transition function. In this example rule, the precondition takes the current object state (`ost`) and an input message (`msgIn`) from the child. It suffices to say that the current status is either I or NP (`ost#[status]` $\leq$ `msiI`), thus the cache needs to request to the parent. Note that the statuses are defined as natural numbers, so we can use arithmetic operations inside the precondition.

The transition function also takes the current object state and the input message, and it returns only the message to the parent. In this example the cache forwards `rqS` to the parent without any meaningful value. As explained in section 5.3, when forwarding a request to the parent, no state transition should happen except locking. The rule template ensures this requirement by restricting the type of the transition function. Note that the rule does not mention locking at all. Each rule template automatically sets a proper lock; in this case an uplock is set right after sending a request to the parent.

```
1  Definition liDownSRsUpDownM: Rule :=
2    rule.rsudo
3    :accepts downRsS
4    :holding rqS
5    :requires (fun _ _ _ => True)
6    :transition (fun ost rsFrom rs rq rsbTo =>
7      (ost +#[status <- msiS]
8           +#[val <- rs.(msg_value)]
9           +#[dir <- setDirS [rsbTo; rsFrom]]
10          +#[owned <- true],
11       <| rsS; rs.(msg_value) |>)).
```

Figure 8-3: L_i responding to the child that requested `rqS`

Figure 8-3 presents another example rule that is fired at the last of the steps shown in Figure 8-1, which sends the response to the child who requested `rqS`. Template `rule.rsudo` says that the rule takes a single response and responds back to the original requestor. The rule `accepts` the response message with the ID `downRsS`. This rule can be executed when it is `holding` a downlock, where the holder contains the original request message with the ID `rqS`. While the rule does not require any additional precondition (`fun _ _ _ => True`), it changes the object state, unlike rules that make requests.

The transition function takes the current object state (`ost`), the object index that sent a response (`rsFrom`), the response message (`rs`), the original request in a lock holder (`rq`), and the index of the original requestor (`rsbTo`). The transition returns a pair of the next state and an output message; this rule sets its status to S, stores the

up-to-date value brought from the `downRsS` message, sets the directory status to S by adding the two children – one that was downgraded to S before and the other that originally requested S – as sharers, and sets the ownership bit as true since the up-to-date value might be dirty in this case. It also sends a response (`rsS`) to the requestor with the up-to-date value. Lastly, the rule automatically releases the downlock.

```
Definition liDropImm: Rule :=
  rule.imm
  :requires (fun ost _ _ => ost#[status] <= msiS ∧ ost#[owned] = false)
  :transition (fun ost => ost +#[status <- msiNP]).
```

Figure 8-4: L_i immediately dropping a line

Figure 8-4 is a rule only used in the noninclusive protocol. Template `rule.imm` is for making an immediate state transition that neither takes any input messages nor generates any output messages. Its precondition just says that the line is possibly shared but need not be written back (the ownership bit is false). In this case, we can *silently* drop the line by setting the status to NP, to denote explicitly that the line is removed. Note that there is no precondition about the directory status at all; a line can be dropped even when the directory status is S or M, which is not allowed in inclusive caches. Our noninclusive protocol employs NCID, and in this case the dropped line may migrate to the so-called extended directory [82]. We will see in section 9.1 how this migration is implicitly processed in the actual hardware implementation.

```
Definition liBInvRqS: Rule :=
  rule.rqsd
  :requires (fun ost => ost#[dir].(dir_st) = msiS)
  :transition (fun ost => (ost#[dir].(dir_sharers), <| downRqIS; 0 |>)).

Definition liBInvRqM: Rule :=
  rule.rqsd
  :requires (fun ost => ost#[dir].(dir_st) = msiM)
  :transition (fun ost => ([ost#[dir].(dir_excl)], <| downRqIM; 0 |>)).
```

Figure 8-5: L_i initiating a back invalidation

Lastly, Figure 8-5 shows two rules to initiate back invalidation, only used in the

inclusive protocol. Template `rule.rqsd` is for making downward requests without any input messages. `liBInvRqS` `requires` the directory status to be S, and in this case it makes downward requests to the sharers. On the other hand, `liBInvRqM` `requires` the directory status to be M, and in this case it makes a single downward request to the exclusive child cache.

It is worth recalling that thanks to the rule templates, rule descriptions in the protocol never mention anything about how to set/release locks to deal with interleavings safely; they are rather designed as if only a single transaction is being processed when executing a rule.

8.2.2 Correctness proof

Now we present the correctness proof of the MSI protocol, claiming that the implementation refines to a single-line memory as a spec. We already discussed in section 4.3 that a number of invariants are necessarily required to prove simulation. The two variants – inclusive and noninclusive protocols – have different set of rules but require the same invariants. In this section, we provide necessary invariants to prove the correctness of the MSI protocol, and we demonstrate how Hemiola helps prove such invariants using predicate messages.

Logical status of a cache

Before talking about invariants, we would like to clarify how to deal with cache statuses in transition. For example, what would be the representative cache status if a cache currently has a status S but is just about to handle the eviction response (say rsPut) that will remove the line? It implies that the parent directory already accepted the eviction request and changed the directory status (for the child) properly. In this case, it would make more sense to regard the child cache status as I.

In order to deal with this situation, we introduce a notion called *logical status* to obtain an abstract status of each cache. In the above example, even if the cache has not handled rsPut yet, the logical status is I. Logical statuses are defined formally as

follows:

- A cache has logical status M iff it has M and there is no rsPut to it.
- A cache has logical status S iff either 1) it has S and there is no rsPut to it or 2) there is a response rsS to it.
- A cache has logical status I iff either 1) it has I and there is no rsM or rsS to it or 2) there is a response rsPut to it.

Note that the logical status of a cache is *not* M when there is a response rsM to it, since the response could still imply an ongoing invalidation process. We will see the actual use case of the rsM message very soon in Figure 8-6.

Invariants The MSI protocol largely requires three invariants, where each invariant corresponds to a desired property of one status – M, S, and I.

Invariant 8.1 (Exclusiveness invariant)**.** *Whenever a cache object O has logical status M, then all the other caches are in logical I status. We denote this predicate by* Invalid $(\lambda c.\ c \neq O)$*, where* Invalid (S) *claims that any cache object in the object set S has logical I status.*

Invariant 8.2 (Sharing invariant)**.** *All caches in logical S status have the same coherent value.*

Invariant 8.3 (Invalidness invariant on ownership bits)**.** *If a cache C has an ownership bit true, then all the caches outside C (i.e., $tr^{-1}(C)$) have logical I status, i.e.,* Invalid $(tr^{-1}(C))$*.*

Invariant 8.4 (Invalidness invariant on directory status M)**.** *If a parent P has a directory status M for a child C_i, then all the caches outside C_i (including P) have logical I status, i.e.,* Invalid $(tr^{-1}(C_i))$*.*

The sharing invariant (Invariant 8.2) is the easiest one to prove, since the rules involved with sharing the coherent value employ just simple value forwarding. For

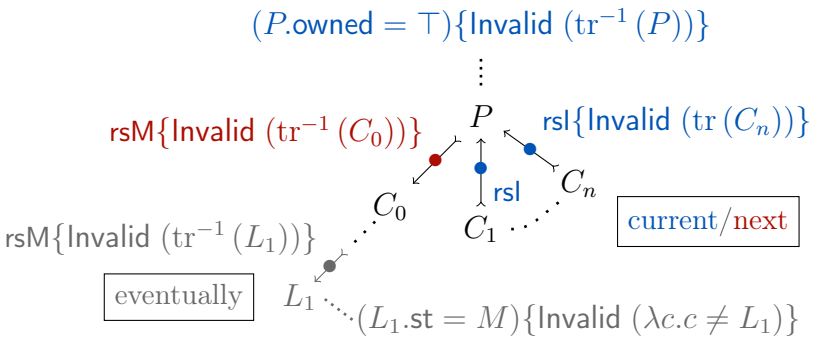


Figure 8-6: Use of predicate messages to prove the exclusiveness invariant

example, the rule in Figure 8-3 just takes a message `downRsS` that contains the coherent value and stores/sends the value. Proving the preservation of the invariant for this rule is very straightforward.

Invariant proofs using predicate messages

While proving the sharing invariant is easy, it is nontrivial to prove the exclusiveness (Invariant 8.1) and invalidness (Invariant 8.3 and Invariant 8.4) invariants, since these invariants are involved with global cache states. We have already learned in section 4.3 that in order to prove this kind of invariant, it is desirable to state some supporting invariants based on the existence of certain messages, called predicate messages. We also discovered that on top of serializability it is much easier to state and prove predicate messages.

We would like to present a practical usage of predicate messages in proving the exclusiveness invariant, shown in Figure 8-6. Suppose that an L1 cache (shown as L_1 in gray in the figure) requested to the parent to get M status. When it finally handles the response rsM, it should know all the other caches (except itself) have the logical I status to prove the exclusiveness invariant (denoted as $\{\mathsf{Invalid}\ (\lambda c.c \neq L_1)\}$). This proof case can be supported using a predicate message for rsM, stating $\mathsf{Invalid}\ (\mathrm{tr}^{-1}\,(C))$ (outside of the subtree rooted at C) when the message goes to C. Since L_1 is the leaf node in the tree, it is trivial to prove $(\mathsf{Invalid}\ (\mathrm{tr}^{-1}\,(L_1)) \rightarrow \mathsf{Invalid}\ (\lambda c.c \neq L_1))$, so we see an example of a predicate message helping prove a conventional invariant.

Figure 8-6 also shows another case, where conventional invariants and predicate messages coordinate to prove a predicate message for the next state. When a child C_i sends its invalidation response rsI, it should know that all the caches inside the subtree of C_i have been invalidated (denoted as Invalid $(\mathrm{tr}\,(C_i))$). When the parent P subsequently handles the invalidation responses, it responds with rsM to the original requestor (C_0 in the figure), requiring to prove Invalid $(\mathrm{tr}^{-1}\,(C_0))$. While P also changes its status to I in this state transition, how do we infer that the caches outside P are in the logical I status, which is required to prove the predicate over rsM? In this case, we should prove a simple object-level invariant of P that it has the ownership bit true. Then we can use Invariant 8.3 to obtain the desired predicate (denoted as $(P.\mathsf{owned} = \top)\{\mathsf{Invalid}\ (\mathrm{tr}^{-1}\,(P))\}$). Combining all the predicates and the state transition by P, we can prove the next predicate message for rsM.

Refinement proof

Once equipped with sufficient invariants, it is straightforward to prove the refinement between the implementation and the spec. The only work is to define a correct simulation that relates all the coherent values in the implementation and the single value in the spec. The coherent values are collected by looking at the logical status of each object; if the logical status is S or M then values either in the object or in some messages (e.g., `rsS`) are coherent.

Denoting by $Coh\,(s, o, v)$ that an object o contains a coherent value v in a system state s, the simulation can be stated as follows:

Theorem 8.2.1 (Correctness of the MSI protocol)**.** *The following simulation relation holds between the implementation system state s^I and the spec system state s^S.*

$$s^I \sim s^S \triangleq \exists v.\ \forall o.\ Coh\,(s^I, o, v) \wedge s^S = Spec\,(v).$$

where $Spec\,(v)$ represents a single-value state for the spec.

8.3 The MESI Protocol

The MESI protocol [65] applies further optimizations to the MSI protocol, by adding a status called Exclusive-Unmodified. As the name of the status says, if a cache line has E status, then the line is exclusive to the cache but also clean. In this section we present a hierarchical MESI protocol and demonstrate that the design and the correctness proof are easily extended from the ones for the MSI protocol with the support of Hemiola.

8.3.1 Protocol description

Cache states

The cache state in the MESI protocol is the same as in the MSI protocol, taking the form $O(\mathsf{st}, \mathsf{v}, \mathsf{dir}, \mathsf{owned})$. The only difference is that the status may be E.

New rules added beyond the MSI protocol

There are several rules added in order to deal with the E status. Like the previous rule-template examples, proper preconditions and transitions are automatically set in terms of locking.

```
Definition l1GetMImmE: Rule :=
  rule.immd
  :accepts rqM
  :from cidx
  :requires (fun ost _ _ => ost#[status] = mesiE)
  :transition
     (fun ost msg => (ost +#[status <- mesiM]
                          +#[val <- msg.(msg_value)],
                      <| rsM; 0 |>)).
```

Figure 8-7: L_1 silently upgraded to M

Figure 8-7 shows a basic case, where an L1 cache is silently upgraded from E to M to write data. Template `rule.immd` says that it takes a request from the external world (similar to Figure 8-2) and immediately sends an external response. As defined

in the state `transition` function, the cache silently changes its status to M, stores the new value from the input message, and responds with `rsM`.

```
Definition liGetSImmME: Rule :=
  rule.immd
  :accepts rqS
  :from cidx
  :requires (fun ost _ _ => mesiE ≤ ost#[status] ∧
                            ost#[dir].(dir_st) = mesiI)
  :transition (fun ost _ => (ost +#[status <- mesiI]
                                 +#[dir <- setDirE cidx],
                             <| rsE; ost#[val] |>)).
```

Figure 8-8: L_i responding with `rsE`

Another case, shown in Figure 8-8, happens when an intermediate cache gets a request from a child to read the data, while it has status E or M. In this case, instead of responding with `rsS`, the cache sends `rsE` to provide E. Once the original requestor obtains E status, it can both read and write the data.

8.3.2 Correctness proof

Logical status and new invariants for E

We extend the notion of logical status from the MSI protocol, declaring that a cache in MESI is E if either 1) it has E and there is no eviction response to it or 2) there is a response `rsE` to it. We should extend the invariants as well to cover caches with E status.

- The exclusiveness invariant (Invariant 8.1) also applies to E; whenever a cache has logical status E, all the other caches are logically in I.
- The sharing (Invariant 8.2) and invalidness (Invariant 8.3 and Invariant 8.4) invariants remain the same.
- A new invariant for E claims that if a cache takes an eviction request *without writeback* from a child, and the directory status pointing to the child is E, then it has a coherent value.

Invariant and refinement proofs

Unlike the exclusiveness and the invalidness invariants, the invariant for E does not involve a large chunk of caches; it is rather an invariant that just relates a child and the parent cache states. Therefore, we do not employ predicate messages in this invariant proof, instead using a normal induction on state-transition steps.

The simulation relation for the MESI protocol is just the same as the one for the MSI protocol, while the coherence predicate $Coh\,(s^I, o, v)$ is extended slightly to cover caches with E status and messages with ID `rsE`.

Chapter 9

Compilation and Synthesis to Hardware

As mentioned in section 8.1.2, so far we have dealt with cache-coherence protocols for a single line, where the specification has a single line as well. In order to build a hardware-synthesizable multiline implementation, we developed a simple compiler that takes a single-line Hemiola protocol as a source program and generates a multi-line implementation described in Kami [15]. Kami is a hardware formal-verification framework, where its own HDL and proof tools are defined in Coq, allowing users to design, specify, verify, and synthesize their hardware components.

The protocol transition system and the rule templates given in the Hemiola DSL match well rule-based HDLs like Kami; a rule in Hemiola naturally maps to an equivalent rule in Kami, which describes atomic state transitions in hardware modules. Instead of directly compiling Hemiola protocols to a register-transfer language (RTL), we chose to build a compiler from Hemiola to Kami as a first step toward using the protocols and their correctness proofs within larger Kami proofs including processors – though this dissertation does not include those composition proofs. Since Kami already has a hardware-synthesis toolchain, we can just compile a Hemiola program to Kami and use the toolchain to run it on FPGAs.

In this chapter, we will explore how a single-line Hemiola protocol is compiled to a multiline cache-coherence protocol implementation in Kami and demonstrate its

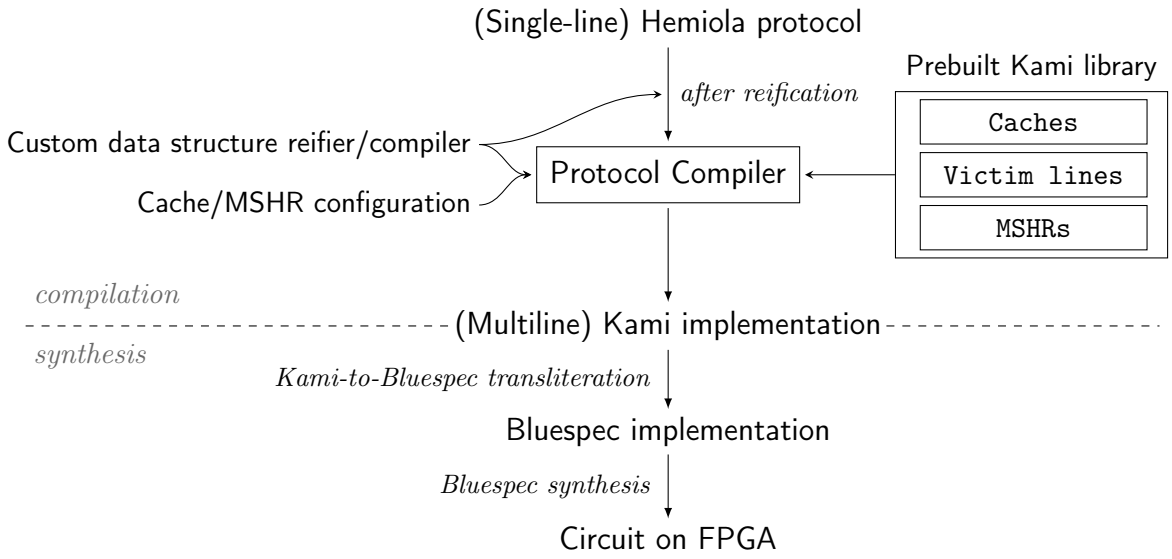


Figure 9-1: Compilation and Synthesis of Hemiola protocols

synthesis to hardware (FPGA). We will further discuss the desired specification of the multiline implementations, which is naturally derived from the source Hemiola protocol.

9.1 Compilation of a Hemiola Protocol

9.1.1 Compiler ingredients

Figure 9-1 depicts a compilation/synthesis flow from a given Hemiola protocol to an FPGA-ready circuit. We provide an overview for each ingredient in the compiler, following the diagram.

Preprocessing: reification

A source program of the protocol compiler is a single-line protocol described in Hemiola with the rule templates. Before feeding a Hemiola source program to the compiler, a preprocessing step is required, which is to reify the program into an AST we can hand off to the compiler. Hemiola supports automated, correct-by-construction reification driven by a series of tactics in Coq. For instance, the rule-reification tactic (`reify_rule`) reifies a Hemiola rule to the corresponding rule AST, where `HRule` is a

Coq record (struct) containing the AST and its correctness, i.e., denotation of the AST matches the denotation of the original rule:

```
Definition hl1GetMImmE: HRule l1GetMImmE := ltac:(reify_rule).
```

Kami: the target language

Kami [15] is a hardware-verification framework embedded in Coq with its own HDL at register-transfer level. A hardware design in Kami consists of modules with encapsulated private state (registers), public methods, and *rules* that make atomic state changes. Rules are fired by a global scheduler by Bluespec, later synthesized to the corresponding scheduling circuitry. The Bluespec scheduler tries to find a maximal number of rules to execute in a cycle, with the convenient semantic guarantee that the execution can be interpreted as if the rules are executed atomically and sequentially, called *one-rule-at-a-time execution*.

As mentioned in section 3.1, the underlying transition system of Hemiola is also rule-based, and the semantics is based on one-rule-at-a-time execution. In this sense, Kami is a good target HDL in that a rule in Hemiola can be compiled to a corresponding rule in Kami.

Another advantage of using Kami as a target language is its verification tools defined in Coq. Similarly to Hemiola, Kami also employs trace refinement as a correctness criterion. Kami as a formal-verification framework provides an effective verification technique called *modular refinement*, which basically says that trace refinement proofs of submodules can be combined together to obtain the refinement for the whole module:

Theorem 9.1.1 (Modular refinement in Kami)**.**

$$\forall I_1, I_2, S_1, S_2.\ I_1 \sqsubseteq S_1 \wedge I_2 \sqsubseteq S_2 \rightarrow I_1 + I_2 \sqsubseteq S_1 + S_2.$$

A corollary of Theorem 9.1.1 is called *modular substitution*, where we can substitute a submodule in another submodule to have a simpler design as an intermediate step to the final refinement:

Theorem 9.1.2 (Modular substitution in Kami)**.**

$$\forall I_1, I_2, I_1'.\ I_1 \sqsubseteq I_1' \rightarrow I_1 + I_2 \sqsubseteq I_1' + I_2.$$

This corollary will be referred to when substituting our optimized cache-controller submodules in the entire memory subsystem with simpler ones, explained in section 9.2.

Reification and compilation of custom data structures

Since a Hemiola protocol may use its own custom data structure (e.g., directory structure for the MESI protocol), the compiler requires a user to provide a reifier and a compiler for the data structure. This task is straightforward for the user, since both reification and compilation work at the level of expressions, not rules. For instance, a field access `dir.(dir_sharers)` for a Coq record `dir` is reified to an expression AST node `(HDirGetSh hdir)`, where `hdir` is the reified directory structure, and compiled to `cdir@."dir_sharers"` in Kami, which uses a field-access expression.

Prebuilt cache-related components

The compiler uses prebuilt hardware components described in Kami. Some of them are for implementing NCID [82] introduced in section 8.1.5, whose operations include asynchronous read and write of the line information (ownership bits, statuses, directory statuses, etc.) and value. The cache normally uses a primitive BRAM (block RAM) module in Kami, later synthesized to a BRAM on an FPGA. The cache module also manages *victim lines* that should be evicted eventually.

Another prebuilt component holds a finite number of MSHRs, whose abstract interface includes registering, updating, and releasing MSHRs with respect to their types (uplock or downlock) and locking addresses. Recall that ideally (as a spec) MSHRs are assigned per-line, but the actual design can contain only a finite number of them. The compiler takes several counts as configuration parameters to determine the sizes of caches (e.g., the number of lines and ways) and MSHRs (e.g., the number

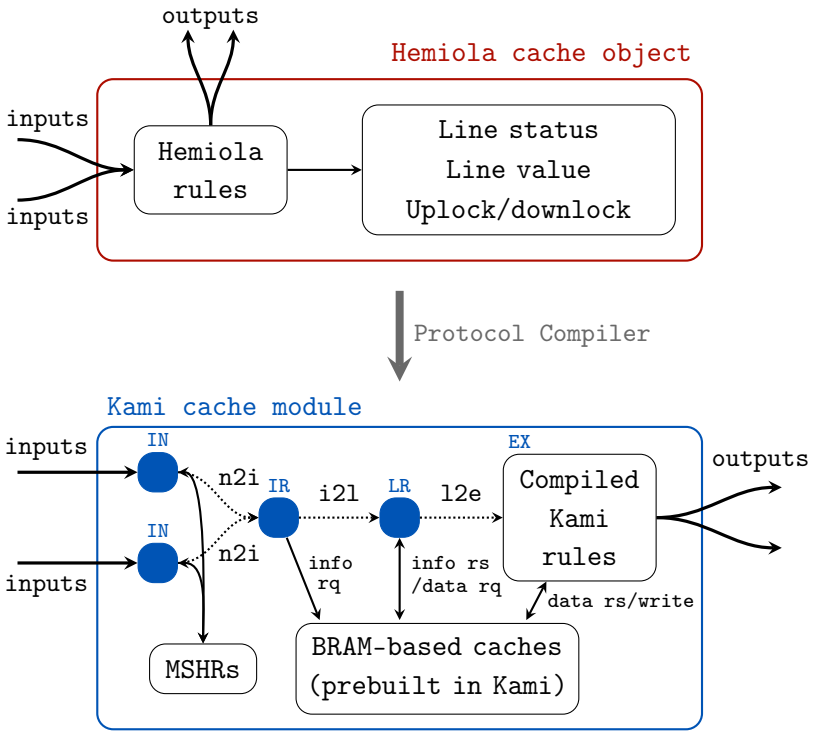


Figure 9-2: Compilation of a pipelined cache module

of uplocks and downlocks).

9.1.2 Compilation of a pipelined cache module

One of the biggest differences between a source Hemiola protocol and the target Kami code is that the target accesses multiple lines *asynchronously*. In the source protocol, a single line can be read (or written) *immediately* by directly accessing a value, whereas in the target the value is accessed asynchronously by making a read (or write) request with a certain line address to a cache and by handling the response.

In order to optimize asynchronous line accesses, the protocol compiler employs a prebuilt pipeline. Figure 9-2 presents a source Hemiola cache object and the corresponding target Kami module, generated by the protocol compiler. Each Hemiola rule in the source cache object takes input messages from various channels, reads/writes values (e.g., a line status, a line value, locks, etc.), and generates output messages (to various channels) atomically.

A Hemiola rule execution corresponds to the execution of multiple stages in the

pipeline, each of which takes part in the source-rule execution. The first stage (IN in the figure) takes input messages from various channels; in order to avoid a deadlock, the pipeline has different entries for inputs from the parent and the children. The IN stage also checks MSHRs to see if the input message can be accepted to the pipeline or not. Depending on the management of MSHRs, it either accepts the input, stalls it, or adds it to the "retry" entries in the MSHR-managing module. Whether to stall or retry later determines whether the protocol is *blocking or nonblocking*; we will discuss the difference in detail right in the next section.

The second stage (IR) asynchronously requests the line information (status, directory status, etc.) with the line address from the input message. It makes the request in terms of the index (from the line address), i.e., it tries to read all the tags and information values with the same index.

The third stage (LR) gets the response for the line information and tries to find if any tag matches the line address. If so, it is a *cache hit*; the LR stage requests to get the line value with the index and tag. Otherwise, it is a *cache miss*; the LR stage decides the victim line by means of the replacement-managing module and requests to get the line value for the victim.

The last stage (EX) gets the response for the line value. In case of a cache hit, using the line information and value read from the earlier stages, a properly selected Kami rule (compiled from a source Hemiola rule) possibly requests a write to the line and generates output messages. In case of a cache miss, the line status is decoded to NP (Not Present), and a proper state transition is made by a Kami rule. The line information and value in this case are the ones for the new victim; the EX stage registers the victim using those values.

Blocking vs. nonblocking protocols

The protocol compiler takes a single-line Hemiola protocol and generates a corresponding multiline implementation. In terms of safety (correctness), this compilation is sound enough, since the coherence of each line is orthogonal to coherence of others.

However, in terms of progress, since the implementation can only use finite re-

sources (e.g., communication channels, MSHRs, etc.), a transaction for a certain line is often *blocked* by another transaction for a different line. For instance, suppose that a request is blocked (stalled) at the `IN` stage in Figure 9-2, because there is already an ongoing transaction with the same line address, represented as an acquired lock in MSHRs. If the pipeline just let this request remain in the pipeline, some other requests, possibly with different line addresses, will be blocked as well since the pipeline is in-order. This case just defines a *blocking cache-coherence protocol*; a stalled request affects the other transactions to be blocked as well.

We can optimize this blocking protocol by moving the blocking request to a designated slot (e.g., a fresh MSHR) and allowing subsequent messages to pass through the pipeline. This is one way to implement a *nonblocking protocol* [39, 72, 4]; a transaction has a chance not to be blocked by other messages. Note that the degree of nonblocking is determined by the number of resources, e.g., the more MSHRs the pipeline has, the less blocking would happen.

The MSHR-managing module, prebuilt in Kami, supports this kind of nonblocking while maintaining the safety of interleavings. That is, a new input message is temporarily held in an MSHR slot when there is already a slot occupied with the same line address. The module manages MSHRs and tracks execution dependencies among them, so the pipeline retries processing the input message that was held temporarily in the MSHR right after the preoccupied MSHR slot is released, i.e., the ongoing transaction is finished.

9.2 Correctness of the Protocol Compiler

As explained in section 9.1, the protocol compiler takes a cache configuration as an argument, thus we can have several different implementations by providing different configurations. Then what would be the specification for all possible implementations from a given source protocol? In this section we naturally extend a single-line Hemiola protocol to a multiline one and justify its role as the specification for multiline implementations.

The core idea is mentioned already in the previous section: *the coherence of each line is orthogonal to coherence of others*. In this case, a single-line Hemiola protocol is naturally extended to a multiline one by using the notion of compositionality. Compositionality claims that if two systems are index-disjoint (i.e., objects and channel indices are disjoint) thus not communicating with each other, then refinement of the composed system is obtained for free just by composing the specs:

Theorem 9.2.1 (Compositionality)**.**

$$\forall I_1, I_2, S_1, S_2.\ I_1 \sqsubseteq S_1 \wedge I_2 \sqsubseteq S_2 \rightarrow I_1 \oplus I_2 \sqsubseteq S_1 \oplus S_2,$$

where $I_1 \oplus I_2$ implicitly assumes that the indices used in I_1 and I_2 are disjoint.

Hemiola additionally supports an *index-extension* mechanism, which takes a system S and a *prefix index* i, generating a new system $S^{(i)}$ where every object or channel index in the system is extended by attaching i. Note that an index in Hemiola is a list of numbers, so it is easy to extend an index just by concatenating another one. Hemiola also provides a lemma that $S^{(i)}$ and $S^{(j)}$ are index-disjoint when $i \neq j$.

Composing these elements, we obtain a replication theorem that is used directly to convert a single-line cache-coherence protocol to an ideal multiline protocol:

Theorem 9.2.2 (Replication)**.** $\forall I, S.\ I \sqsubseteq S \rightarrow \forall n.\ \bigoplus_{i=0}^{n} I^{(i)} \sqsubseteq \bigoplus_{i=0}^{n} S^{(i)}$.

The multiline protocol derived from the replication theorem is indeed ideal; it has line values, lock holders, and communication channels per cache line. Thus the protocol can serve as a spec for all the multiline implementations generated by the protocol compiler, since they have limited resources, which implies that the behavior of the multiline protocol covers their behaviors.

Figure 9-3 elaborates more on the role of the multiline specification. For a given Hemiola single-line protocol (I_0^h), we can lift the refinement using the replication theorem to obtain a refinement for the multiline protocol (I_M^h). Since both Hemiola and Kami are rule-based description languages, we expect it is straightforward to have an ideal multiline protocol in Kami (I_M^k) in the sense of simple transliteration, while

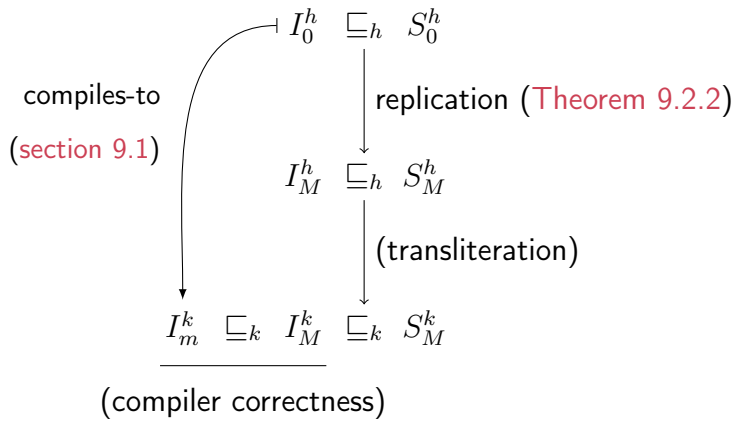


Figure 9-3: A single-line protocol, a multiline spec, and multiline implementations

preserving the refinement. After all, the correctness of the protocol compiler must be refinement from a target Kami implementation (I_m^k) to the multiline protocol. In proving this refinement, Theorem 9.1.2 will be required, since every pipelined cache module in I_m^k should be substituted with a naive module in I_M^k, transliterated from I_M^h.

As presented in Figure 9-3, the compiler correctness proof is a part of the bigger correctness proof of a cache-coherent memory subsystem implementation at the lower level. While this dissertation only focuses on proving cache-coherence protocols described in the high level, we believe that proving the low-level implementation in this manner is another valuable future work.

9.3 Synthesis of a Hemiola Protocol

Once we have obtained a multiline cache-coherence protocol implementation from the compiler, we can use Kami's synthesis toolchain to transliterate it to a Bluespec [56] program and synthesize it to load on an FPGA.

Before synthesis, we first evaluated two Hemiola protocols, Hem_2 and Hem_3, instantiated from our hierarchical noninclusive MESI protocol described in section 8.3, using the Bluespec simulator. Hem_3 is a 3-level protocol, consisting of four 32KB 4-way set-associative L1 caches, two 128KB 8-way L2 caches, and a 512KB 16-way

Performance (#trs/cycle)	AllSh	PairSh	ExSh1	ExSh2
Hem_3	0.259	0.868	0.506	0.764
Hem_2	0.270	0.800	0.637	0.913
RiscyOO	0.336	0.791	0.637	0.988
#trs in 5×10^5 cycles				
Hem_3	129,459	434,063	252,797	381,801
Hem_2	134,927	399,919	318,727	456,674
RiscyOO	167,978	395,612	318,521	493,851

Figure 9-4: Evaluation of Hemiola protocols

last-level cache. Hem_2 is 2-level, consisting of four L1 caches and the last-level cache. Each line holds 32 bytes in all the protocols. We compared the performance with an existing Bluespec implementation, RiscyOO [80, 81], featuring a 2-level inclusive MESI protocol with self-invalidation [68]. We set the cache sizes of RiscyOO the same as for Hem_2. We also set the number of MSHR slots in each cache between RiscyOO and Hem_2 the same.

In order to measure the performance of the Hemiola protocols, we designed four artificial workloads, shown in Figure 9-4, that make random requests but mimic some amount of temporal/spatial locality of memory accesses. We measure the performance of each workload with a register-file-based 512KB reference memory, thus any memory address used in the workloads is within $[0, 2^{19} - 1]$.

The AllSh workload generates random memory requests for each core, where each request address is randomly chosen throughout the entire memory range, regarding all the memory values are shared by all the cores. We assume the memory read/write ratio as $3 : 1$ (i.e., memory reads are three-times more than writes) for each workload.

The PairSh, ExSh1, and ExSh2 workloads use designated exclusive and shared memory regions, determined by several parameters. An exclusive memory region is set for each L1 cache, whose size equals the size of the cache. Any two exclusive memory regions are disjoint to each other. A shared region has the size of $2^{\mathsf{LgShRange}}$

	Clock length (ns)	Critical path (ns)	#LUTs	#FFs
Hem_2	40	36.861	126,714	41,203
Hem_3	40	37.608	240,034	61,011

Figure 9-5: Synthesis of Hemiola protocols

lines, where LgShRange specifies the log size of the shared region. All the three workloads use LgShRange = 5, i.e., 32 lines are shared by the L1 caches.

In the PairSh workload, the first two L1 caches only accesses the first shared region, and the other two caches only accesses the second one. The two shared regions are disjoint to each other. Any exclusive region and shared region are disjoint as well, but they can have the same L1-cache index, thus some evictions may happen from each L1 cache. ExSh1 and ExSh2 use a single shared region.

Each of the three workloads takes two additional parameters to simulate temporal/spatial locality. LgExShRatio specifies the access ratio between the exclusive and shared regions. PairSh and ExSh1 use LgExShRatio = 1, meaning that the workload makes memory accesses of the exclusive and shared regions by equal chances. ExSh2 uses LgExShRatio = 2, i.e., the probability to access the shared region is $1/2^2$. NumTlCycles specifies how many consecutive memory accesses happen in a specific region. All the workloads use NumTlCycles = 25, meaning that 25 memory accesses happen for each visit of a region.

Figure 9-4 also shows the performance result. We measured the performance by counting the number of transactions performed in 5×10^5 simulation cycles. Though one should not draw too many conclusions from the precise measurements, the result shows that the Hemiola protocols are competitive with a practical implementation coded by hand.

Next we synthesized the Hemiola protocols, shown in Figure 9-5. We used Xilinx's Virtex-7 VC707 FPGA [1] and its toolkit for synthesis. Each protocol uses a minimal clock length that can safely cover its critical path. Both Hem_3 and Hem_2 stayed within the FPGA's budget of lookup tables (LUTs) and flip-flops (FFs). We performed tandem verification covering over 10^9 memory requests for each protocol

on the FPGA, by connecting it to a tester module that generates a random workload and a reference memory to check its correctness.

Chapter 10

Related Work II: Verification of Cache-Coherence Protocols

In this chapter, we explore a number of approaches to verifying various cache-coherence protocols. While testing and bounded model checking (BMC) [18] have dominated the verification of the protocols in industry, formal verification also has been developed and used steadily to verify modern protocols. Formal verification (at least of hardware) is performed largely by two methods: model checking or theorem proving. We describe how each methodology has been developed to cover practical cache-coherence protocols.

Model-checking cache-coherence protocols Model checking has dominated the formal verification of hardware. Various model checkers like Murphi [27, 26], SMV [50, 37], and TLA+ [41] have been used to verify either high-level hardware designs or RTL implementations. Verification of cache-coherence protocols, in the beginning, was limited to finite numbers of caches and specific protocols due to conventional state-space-explosion issues. McMillan and Schwalbe [53] verified the Gigamax cache-coherence protocol, which employs bus snooping. They used SMV [50, 37] to verify safety and liveness of the protocol, but at that time only succeeded with the 2-level protocol with two clusters, each of which has six processors. Joshi et al. tried to verify various cache-coherence protocols using TLA+ and TLC [41]. In early phases

the TLC model checker was not mature enough to provide a full correctness proof, so they focused on proving a number of invariants and finding bugs through those invariant proofs. Later they tried to verify another protocol with a more-developed TLC, but the verification was still limited to a single cache line, two data values, and three processors, due to the state-space explosion to twelve million reachable states. These verifications were performed quite a long time ago, but we want to draw from these early approaches that state-space explosion has been a typical issue in model checking.

In order to overcome such state-space explosion, model checkers have been developed featuring sound techniques such as symbolic model checking [49, 50], partial-order reduction [6], symmetry reduction [7, 35], data-type reduction [47], and compositional model checking [47]. As explored in chapter 7, model-checking cache-coherence protocols especially requires the design and use of noninterference lemmas to deal with the state-space explosion by interleavings [47, 51, 16].

In order to obtain effective noninterference lemmas, a number of approaches used descriptions in terms of transactions (called "message flows") [74, 58, 69]. These papers claimed that effective verification obligations can be constructed by looking at the ordering information in message flows and demonstrated that such obligations help the CMP method [51, 16] converge. Instead of looking at each transaction, Hemiola provides serializability that guarantees noninterference among any transactions defined on top of the framework.

In order to verify cache-coherence protocols with arbitrary numbers of cores (but no hierarchy), parameterization has been used in designing and model-checking the protocols [28, 29, 16, 79, 78, 3]. Since Hemiola is built on Coq, we can take full advantage of parameterization, and indeed the framework supports verification of cache-coherence protocols with an arbitrary tree shape as a parameter.

In order to increase scalability further, recent approaches used abstraction and modularity in protocol design and successfully verified hierarchical cache-coherence protocols in a compositional way [13, 14, 45, 46]. However, these approaches face a common obstacle: because of the modularity requirement, it became hard to de-

sign and verify noninclusive protocols. There have been approaches [13, 14] to solve this problem using assume-guarantee reasoning and history variables [21, 19], while still maintaining the concept of compositional verification, but they faced state-space explosion again, and thus they just verified a two-level MSI protocol with three L2 caches. Recently, the Neo theory [45, 46] has been developed as a safe way to compose "subtrees" of caches to have a hierarchical protocol. The authors argued that it is possible to verify noninclusive protocols in the Neo framework when a directory is still inclusive (e.g., NCID [82]) but did not provide the actual design and proof. Here in Hemiola we provided the proofs of hierarchical noninclusive cache-coherence protocols, without any such restrictions.

Another notable success of cache-coherence verification employed program synthesis to generate a protocol for a given atomic specification [61, 60]. The synthesizer can generate various hierarchical protocols, each of which is nontrivial and more optimized than the protocols defined in Hemiola. The atomic specification is similar to Hemiola's rule templates in the sense that a user can describe the protocol just with stable states. They used Murphi to verify synthesized protocols, but in a two-level protocol [61] they only succeeded up to three caches without exhausting memory, and for hierarchical protocols [60] they succeeded only with the root, two cache-H, and two cache-L nodes. They also mentioned that this extension to hierarchical protocols cannot cover noninclusive protocols.

Theorem proving for cache-coherence protocols While not dominant, theorem proving also has been used steadily to verify cache-coherence protocols. In the early days, the correctness proofs were done with specific protocol models, where the relation to the corresponding RTL implementation is left as TCB [67, 54]. The FLASH cache-coherence protocol model was proven to be correct using the PVS theorem prover [67]. They used aggregation of distributed transactions [66] to verify the protocol and claimed that the technique can be applied generally for other protocols. Another approach proved a write-invalidate cache-coherence protocol using the ACL2 theorem prover [54]. The protocol is based on bus snooping, and the design and proof

are parameterized by the number of processors.

One past project tried to prove both correctness and liveness of adaptive cache-coherence protocols using the PVS theorem prover [71, 73, 70]. They verified the Cachet adaptive cache-coherence protocol that consists of several micro-protocols. Inside the protocol, each line in a cache can dynamically switch from one micro-protocol to another one for performance. In order to verify the protocol more easily, they suggested to design and prove protocols in two stages: one for imperative rules, which are only about correctness, and the other for directive rules that affect both correctness and liveness. Imperative rules correspond to the rules defined in Hemiola for protocol description, whereas directive rules correspond to the rules in the pipelined cache controller prebuilt in the Hemiola protocol compiler.

A recent success was a proof of a hierarchical MSI protocol with an arbitrary tree topology using Coq [76]. However, the proof was also for the specific protocol and thus not structured to promote streamlined reuse of results for other protocols. It also employed rather complex and ad-hoc invariants needed to characterize transient states. This work provided basic motivation of the Hemiola framework in designing and reusing invariants more systematically in a theorem prover.

Another notable project designed a modular-specification approach for cache coherence, verifying each module (i.e., cache) against the spec while automatically generating (and proving) invariants, using the Ivy verification tool [63, 48, 52]. The modular spec is specialized to the TileLink protocol [22], whose interface is similar to the rule templates in Hemiola. That being said, this project targeted only the specific TileLink protocol, thus not clearly distinguishing which invariants can be reused for other protocols (like serializability provided by Hemiola) and which are for the actual protocol.

In order to avoid tedious invariant search, a toolchain was built to generate invariants and their proofs automatically for the FLASH cache-coherence protocol [43]. However, it did not provide the final refinement proof between the implementation and the spec.

Conclusion

In this dissertation, we discussed difficulties in designing and proving hardware cache-coherence protocols, proposed a methodology to ease the burden, and demonstrated it by building a framework Hemiola within the Coq interactive theorem prover. We also designed and proved various hierarchical cache-coherence protocols on top of the framework and synthesized them using the protocol compiler provided by the framework.

As the last chapter of this dissertation, we would like to summarize each previous chapter and to discuss a number of future-work directions. Afterwards, we conclude by adding remarks, recalling our introduction to why hardware verification is complex and how we can use an interactive theorem prover to alleviate the verification burden.

Chapter summaries

We began with constructing an underlying basis to reason about hardware cache-coherence protocols. We defined protocol transition systems in chapter 3, viewing cache-coherence protocols as message-passing systems. A system consists of objects, and the objects communicate by ordered message channels. Local state transitions within an object are performed by rules, each of which has a precondition and a transition function. In a protocol transition system, thanks to the so-called one-rule-at-a-time semantics, it is safe to assume that a state-transition step of a system is always made nondeterministically by a rule in an object.

On top of protocol transition systems, we provided the formal definition of serializability in chapter 4. A memory subsystem consists of several caches and the main memory, and a cache-coherence protocol defined within the system allows handling

multiple memory requests concurrently in a distributed way. Viewing cache-coherence protocols as distributed protocols, we call such concurrent executions interleaved, and the serializability property guarantees that any interleaving can be interpreted as a sequence of transaction executions reaching the same resulting state. We also provided a novel mechanism to prove invariants, called predicate messages, and demonstrated that it is much easier to state and prove predicate-message invariants in a serializable system.

As the next part of this dissertation, we established a framework called Hemiola, embedded in the Coq proof assistant, to design and prove cache-coherence protocols more easily. Even if serializability can ease the burden of stating and proving invariants, it will still be a large burden if a user has to prove serializability per-protocol. Thus we presented the Hemiola DSL in chapter 5, where any protocol defined by the DSL automatically obtains the serializability property for free. The DSL contains rule templates, each of which takes input messages, sets/releases locks, and produces output messages properly to satisfy serializability.

We described a proof in chapter 6 that the use of the rule templates indeed implies serializability, using a well-known technique called commuting reductions. This implication is the biggest theorem in Hemiola, but the theorem is general (thus reusable) in that it can be applied to any protocol defined using the rule templates. Commuting reductions are the most basic notion of noninterference between two adjacent state transitions, and the serializability proof employs them by showing that any interleaved history of a system defined with the Hemiola DSL can be reduced to a sequential history that reaches the same state, by a finite number of reductions. In order to show that finitely many reductions suffice, we developed the notion of mergeability to figure out which two atomic-history fragments to merge while not breaking the other fragments.

The final part of this dissertation was to design and synthesize hierarchical cache-coherence protocols as case studies. We defined three hierarchical protocols on top of Hemiola: inclusive and noninclusive MSI protocols and a noninclusive MESI protocol. All of the protocols require a similar set of invariants: the exclusiveness, sharing,

and invalidness invariants. We provided in chapter 8 the correctness proofs of the protocols using these invariants and demonstrated that predicate messages help prove the invariants.

Lastly, the case-study protocols were compiled to corresponding hardware implementations and synthesized to run on FPGAs. In chapter 9, we developed the protocol compiler, which takes a single-line Hemiola protocol and generates a multiline implementation defined in the Kami hardware-verification framework. The compiler employs a number of prebuilt hardware components also defined in Kami, which include noninclusive caches, inclusive directories, MSHR controllers, and so on. Furthermore, we synthesized compiled Kami protocol implementations by borrowing the toolchains in Kami and Bluespec. Our evaluation shows that the implementations are competitive with a practical implementation coded by hand.

Future work

There are a number of valuable future-work directions to make the Hemiola framework more robust. First of all, as already mentioned in section 9.2, we plan to prove the correctness of the protocol compiler. The compiler correctness proof will be used eventually for proving the correctness of low-level cache-coherent memory subsystems compiled from Hemiola protocols.

Another important future direction is to prove liveness of the protocols designed in Hemiola. Even though we designed the rule templates carefully to avoid deadlocks, we think that a formal proof should be supported by the framework as well. Proving liveness generally requires deadlock, livelock, and starvation freedom. Even though such properties should be properly stated and proven for low-level hardware implementations, it will be interesting to explore some properties in the protocol-description level that can help prove liveness.

The last future work is to extend the rule templates to cover more sophisticated message-communication patterns. Highly optimized cache-coherence protocols often require message-communication patterns that cannot be performed by the current rule templates. For instance, some protocols require direct communication among

siblings (i.e., not arbitrated by the parent), and no rule templates currently support it. Extending the rule templates will require the extension of the serializability proof, which includes adding more rule-template cases for each invariant proof.

Concluding remarks

Hemiola as a research project has started to answer the question "how we can formally verify practical cache-coherence protocols with reasonable effort." This question involves two research challenges. First, practical protocols are huge, designed with cache hierarchies, which implies that a verification methodology should be scalable. Second, there are a lot of design options, and the methodology should be general enough to cover many designs.

We tried to solve these problems by taking full advantage of an interactive theorem prover. Verification effort is reduced by proving a general theorem that can be applied for various purposes. The more general the theorem is, the more verification effort is reduced. This idea looks very trivial but is hard to realize in practical verification, since to design such a general theorem itself is challenging, and the proof will be difficult as well. Indeed, in building the Hemiola framework, we spent a significant amount of the time finding a general set of requirements that ensures serializability, plus proving that assertion in Coq.

That said, we believe Hemiola represents good evidence that interactive theorem proving is worthwhile to use for practical hardware verification. While model checking has dominated the formal verification of hardware especially in industry, it apparently seems that (interactive) theorem proving has not been employed practically, due to a stereotype that theorem proving requires too much human effort. We would like to argue that this stereotype is no longer true, once equipped with good proof structures, e.g., reusable proofs. Lastly, we hope that in the future there will be more approaches to verifying realistic hardware components with interactive theorem provers.

Bibliography

[1] 7 Series FPGAs Configurable Logic Block – User Guide, September 2016.

[2] 6th Gen Intel® Core™ X-Series Processor Family Datasheet, Vol. 1, December 2018.

[3] Christopher J. Banks, Marco Elver, Ruth Hoffmann, Susmit Sarkar, Paul Jackson, and Vijay Nagarajan. Verification of a lazy cache coherence protocol against a weak memory model. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, FMCAD '17, pages 60–67, Austin, TX, 2017. FMCAD Inc.

[4] Samson Belayneh and David R. Kaeli. A discussion on non-blocking/lockup-free caches. *SIGARCH Comput. Archit. News*, 24(3):18–25, June 1996.

[5] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[6] Ritwik Bhattacharya, Steven German, and Ganesh Gopalakrishnan. Symbolic partial order reduction for rule based transition systems. In *Proceedings of the 13 IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods*, CHARME '05, pages 332–335, Berlin, Heidelberg, 2005. Springer-Verlag.

[7] Ritwik Bhattacharya, Steven M. German, and Ganesh Gopalakrishnan. Exploiting symmetry and transactions for partial order reduction of rule based specifications. In *Proceedings of the 13th International Conference on Model Checking Software*, SPIN '06, pages 252–270, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The essence of Bluespec: A core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 243–257, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Thomas Braibant and Adam Chlipala. Formal verification of hardware synthesis. In *CAV 2013, 25th International Conference on Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2013.

[10] Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47âĂŞ65, August 2016.

[11] Stephen D. Brookes and William C. Rounds. Behavioural equivalence relations induced by programming logics. In Josep Diaz, editor, *Automata, Languages and Programming*, pages 97–108, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg.

[12] Tej Chajed, M. Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 307–322, Berkeley, CA, USA, 2018. USENIX Association.

[13] Xiaofang Chen. *Verification of Hierarchical Cache Coherence Protocols for Futuristic Processors*. PhD thesis, USA, 2008. AAI3322423.

[14] Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan, and Ching-Tsun Chou. Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. *Form. Methods Syst. Des.*, 36(1):37–64, February 2010.

[15] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP):24:1–24:30, August 2017.

[16] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design*, pages 382–398, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[17] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[18] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, July 2001.

[19] Edmund M. Clarke. Proving the correctness of coroutines without history variables. In *Proceedings of the 16th Annual Southeast Regional Conference*, ACM-SE 16, pages 160–167, New York, NY, USA, 1978. Association for Computing Machinery.

[20] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, Berlin, Heidelberg, 1981. Springer-Verlag.

[21] M. Clint. Program proving: Coroutines. *Acta Inf.*, 2(1):50–63, March 1973.

[22] Henry Cook. Productive design of extensible on-chip memory hierarchies. 2016.

[23] The Coq Development Team. *The Coq Reference Manual*, 12 2020.

[24] N. Dave, M. C. Ng, and Arvind. Automatic synthesis of cache-coherence protocol processors using Bluespec. In *Proceedings of the 2Nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '05, pages 25–34, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as rule composition. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE '07, pages 51–60, Washington, DC, USA, 2007. IEEE Computer Society.

[26] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers Processors*, pages 522–525, 1992.

[27] David L. Dill. The Murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 390–393, London, UK, UK, 1996. Springer-Verlag.

[28] E. Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, pages 247–262, 2003.

[29] E. Allen Emerson and Vineet Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 144–159, Berlin, Heidelberg, 2003. Springer-Verlag.

[30] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4), August 2008.

[31] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, June 2017.

[32] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 449–465, Cham, 2015. Springer International Publishing.

[33] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[34] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, USA, 2004.

[35] Chung-Wah Norris Ip. *State Reduction Methods for Automatic Formal Verification*. PhD thesis, 1996.

[36] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 353–364, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 396–410, Berlin, Heidelberg, 2001. Springer-Verlag.

[38] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983.

[39] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 81–87, Washington, DC, USA, 1981. IEEE Computer Society Press.

[40] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[41] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[42] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, pages 382–401, July 1982.

[43] Y. Li, J. Cao, and K. Duan. An automatic parameterized verification of FLASH cache coherence protocol. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 47–58, July 2018.

[44] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975.

[45] Opeoluwa Matthews, Jesse Bingham, and Daniel J. Sorin. Verifiable hierarchical protocols with network invariants on parametric systems. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, FMCAD '16, pages 101–108, Austin, Texas, 2016. FMCAD Inc.

[46] Opeoluwa Matthews and Daniel J. Sorin. Architecting hierarchical coherence protocols for push-button parametric verification. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50

'17, pages 477–489, New York, NY, USA, 2017. Association for Computing Machinery.

[47] K. L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 219–237, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[48] Kenneth McMillan. Modular specification and verification of a cache-coherent interface. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, FMCAD '16, pages 109–116, Austin, TX, 2016. FMCAD Inc.

[49] Kenneth L. McMillan. *Symbolic model checking - an approach to the state explosion problem.* PhD thesis, 1992.

[50] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, USA, 1993.

[51] Kenneth L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '01, pages 179–195, Berlin, Heidelberg, 2001. Springer-Verlag.

[52] Kenneth L. McMillan and Oded Padon. Ivy: A multi-modal verification tool for distributed algorithms. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 190–202, Cham, 2020. Springer International Publishing.

[53] KL McMillan and James Schwalbe. Formal verification of the Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 111–134, 1992.

[54] J. Strother Moore. An ACL2 proof of write invalidate cache coherence. In *Proceedings of the 10th International Conference on Computer Aided Verification*, CAV '98, pages 29–38, London, UK, UK, 1998. Springer-Verlag.

[55] V. Nagarajan, D. J. Sorin, M. D. Hill, D. A. Wood, N. Enright Jerger, and M. Martonosi. *A Primer on Memory Consistency and Cache Coherence: Second Edition.* 2020.

[56] R. Nikhil. Bluespec system verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, pages 69–70, 2004.

[57] Peter W. O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 49–67, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[58] J. O'Leary, M. Talupur, and M. R. Tuttle. Protocol verification using flows: An industrial experience. In *2009 Formal Methods in Computer-Aided Design*, pages 172–179, 2009.

[59] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 297–306, New York, NY, USA, 1993. Association for Computing Machinery.

[60] N. Oswald, V. Nagarajan, and D. J. Sorin. HieraGen: Automated generation of concurrent, hierarchical cache coherence protocols. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 888–899, 2020.

[61] Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. Protogen: Automatically generating directory cache coherence protocols from atomic specifications. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 247–260, Piscataway, NJ, USA, 2018. IEEE Press.

[62] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6(4):319–340, December 1976.

[63] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. *SIGPLAN Not.*, 51(6):614–630, June 2016.

[64] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Inc., USA, 1986.

[65] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, January 1984.

[66] Seungjoon Park. *Computer Assisted Analysis of Multiprocessor Memory Systems*. PhD thesis, 1996.

[67] Seungjoon Park and David L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, pages 288–296, New York, NY, USA, 1996. ACM.

[68] Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 241–252, New York, NY, USA, 2012. Association for Computing Machinery.

[69] Divjyot Sethi, Muralidhar Talupur, and Sharad Malik. Using flow specifications of parameterized cache coherence protocols for verifying deadlock freedom. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis*, pages 330–347, Cham, 2014. Springer International Publishing.

[70] Xiaowei Shen. *Design and verification of adaptive cache coherence protocols*. PhD thesis, 2000.

[71] Xiaowei Shen, Arvind, and Larry Rudolph. Cachet: An adaptive cache coherence protocol for distributed shared-memory systems. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, page 135âĂŞ144, New York, NY, USA, 1999. Association for Computing Machinery.

[72] Gurindar S. Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):53–62, April 1991.

[73] Joseph Stoy, Xiaowei Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, pages 43–71, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[74] Murali Talupur and Mark R. Tuttle. Going with the flow: Parameterized verification using message flows. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08. IEEE Press, 2008.

[75] Muralidaran Vijayaraghavan. *Modular Verification of Hardware Systems*. PhD thesis, 2016.

[76] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular deductive verification of multiprocessor hardware designs. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 109–127, Cham, 2015. Springer International Publishing.

[77] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 888–904, 2019.

[78] M. Zhang, J. D. Bingham, J. Erickson, and D. J. Sorin. Pvcoherence: Designing flat coherence protocols for scalable verification. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 392–403, 2014.

[79] Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 471–482, USA, 2010. IEEE Computer Society.

[80] S. Zhang, A. Wright, T. Bourgeat, and A. Arvind. Composable building blocks to open up processor design. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 68–81, 2018.

[81] Sizhuo Zhang. *Constructing and evaluating weak memory models*. PhD thesis, 2019.

[82] Li Zhao, Ravi Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. NCID: A non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 121–130, New York, NY, USA, 2010. Association for Computing Machinery.