EasyChair Preprint

№ 8623

Hemiola: a DSL and Verification Tools to Guide Design and Proof of Hierarchical Cache-Coherence Protocols

Joonwon Choi, Adam Chlipala and Arvind

August 9, 2022

Hemiola: A DSL and Verification Tools to Guide Design and Proof of Hierarchical Cache-Coherence Protocols

Joonwon Choi, Adam Chlipala, and Arvind

MIT CSAIL, USA
joonwonc@alum.mit.edu {adamc,arvind}@csail.mit.edu

Abstract. Cache-coherence protocols have been one of the greatest challenges in formal verification of hardware, due to their central complication of executing multiple memory-access transactions concurrently within a distributed message-passing system. In this paper, we introduce Hemiola, a framework embedded in Coq that guides the user to design protocols that never experience inconsistent interleavings while handling transactions concurrently. The framework provides a DSL, where any protocol designed in the DSL always satisfies the serializability property, allowing a user to verify the protocol assuming that transactions are executed one-at-a-time. Hemiola also provides a novel invariant proof method, for protocols designed in Hemiola, that only requires considering execution histories without interleaved memory accesses. We used Hemiola to design and prove hierarchical MSI and MESI protocols as case studies. We also demonstrated that the case-study protocols are hardware-synthesizable, by using a compilation/synthesis toolchain targeting FPGAs.

Keywords: formal verification · cache coherence · proof assistants.

1 Introduction

Programming languages and compilers help engineers describe each system at the most expedient level of abstraction. The process of experimenting with new languages is most familiar from the software world, but hardware designers also benefit from it. Of course, Verilog and VHDL themselves are significant steps up from direct circuit descriptions. Some families of hardware languages go further, in roughly the sense that, say, Java goes further than C, providing abstractions that simplify reasoning about modular design. The rule-based hardware languages like Bluespec [23] allow hardware designers to *imagine* that system modules take turns executing local atomic state-change rules, with *no concurrency*. In reality, parallel execution is essential for performance, and compilers for these languages rely on static analysis to extract parallelism soundly.

Roughly speaking, a rule in Bluespec and its relatives must run within a single clock cycle. What happens when we want to simplify reasoning about *longer-running processes*? A prime example is a cache-coherence protocol. A

memory hierarchy is a distributed system, with many caches communicating through explicit message passing, requiring at least as many clock cycles as the longest dependency chain of message exchanges. The logic is notoriously difficult to get right. One reason is that many memory requests from processor cores may be handled simultaneously. One cache may be working on one request, while a neighboring cache is working on a different request. Might there be abstractions that remove this complication from the hardware designer's thought process, much as Bluespec allows the same designer to pretend that different hardware components do not execute state-change logic in parallel?

We answer affirmatively in presenting *Hemiola*, the first hardware-description language that presents cache-coherence transactions *as if they run atomically*, while realizing the usual parallel performance gains. We define a transaction as all the activity within the memory system in response to a single request from a processor core or other user of the memory. One request may trigger a flurry of activity in the protocol, but the designer may at least pretend that no other request is active in the same period.

The foundation of Hemiola is identifying *commonalities across practical cache-coherence protocols* and embodying them in a domain-specific language (DSL). We fix a notion of node hierarchy and message-passing channels, enumerating *rule templates* capturing relevant communication patterns. Protocols are then described in terms of single-cycle, per-cache rules, each instantiated from a template. Crucially, a locking discipline is built into the language and handled automatically by the templates.

In addition to the DSL, Hemiola provides formal tools significantly easing verification of all cache-coherence protocols designed in it. The DSL is embedded in the Coq proof assistant and has a fully machine-checked proof of soundness, formalized as *serializability*: any state invariant preserved with one-transaction-at-a-time execution is also preserved in true parallel execution. The serializability property is once-and-for-all at the language level, freeing protocol designers from needing to reason about interleavings among transactions. In a sense, our work takes techniques that have been used for *per-protocol* verification and lifts them to apply at the level of a DSL, so that no verification effort need be expended on them per-protocol.

To sum up, the contribution of this paper consists of two parts[1]:

- We discover a set of topology and lock conditions that ensures *serializability*, extracted from usual cache-coherence protocol designs. We then identify a DSL, where every protocol defined in this language ensures serializability by-construction, backed up with mechanized Coq proof (section 3). Lastly, we formalize how serializability helps prove global invariants, by using the novel notion of *predicate messages* in distributed protocols (section 4).
- We provide the complete correctness proofs of hierarchical cache-coherence protocols (section 5) using Hemiola. Our case studies are the first complete

[1] Our framework and case studies are available as open source: https://github.com/mit-plv/hemiola. Choi's dissertation [9] goes into additional detail.

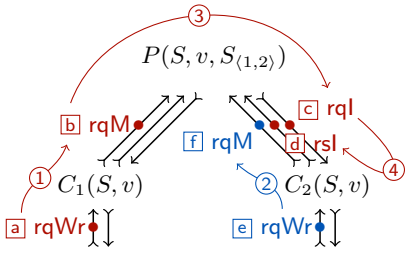


Fig. 1: A simple MSI directory protocol and its rule-execution cases

mechanized proofs that share a large segment of reusable proofs across various cache-coherence protocols. We also demonstrate that the case-study protocols are hardware-synthesizable, by using a compilation/synthesis toolchain in Hemiola (section 6).

2 A Motivating Example

Before introducing our proposed method to design and verify cache-coherence protocols, we provide a simple motivating example to explain the typical challenges and how we suggest to handle them. For simplicity, in this section, we will consider a protocol handling only *a single memory location*. We will see it is still nontrivial to design a correct protocol.

The overall goal of cache coherence is 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. Figure 1 shows caches and network channels for a directory-based MSI protocol. There are three caches (P, C_1, and C_2), and each of them has its own status (**M**odified, **S**hared, or **I**nvalid) and data (v). In this MSI protocol, a cache can read/write the data with the M status, only read with S, and cannot read/write with I. 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.

Caches communicate through ordered channels, shown as ($\rightarrowtail$) in the figure. Child caches (C_1 and C_2) have channels to receive and respond to requests from processor cores. There are three types of channels between a parent and a child: one channel is for parent-to-child messages, and the other two channels are for child-to-parent requests and responses. It is natural to wonder why two separate child-to-parent channels are required; we will see the reason very soon.

Figure 1 also depicts some example state-transition cases depending on the cache statuses. In this setting, all the caches run concurrently by repeatedly executing *rules* that make atomic, 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.

A rule execution ① is a case where a child C_1 takes a request [a] rqWr from a processor 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 practical 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) and to make *any further processor requests stall*.

Due to the concurrent execution of caches, we might have another rule executed at the same time. ② is executed concurrently with ①, where C_2 also takes a processor request [e] rqWr and sends [f] rqM to the parent as well. Since ① and ② happened at the same time, the parent P needs to decide which request to deal with. Suppose that it decided to handle [b] rqM first.

③ presents the next execution by P, taking the input message [b] rqM and making an invalidation request ([c] rqI) to the other child C_2 to change its status to I. This 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 to disallow any other requests from the children (e.g., [f] rqM), since otherwise it will handle two rqM messages simultaneously, which might lead to an incoherent state – two M statuses in the caches.

Lastly, ④ shows the case that C_2 handles the invalidation request ([c] rqI). A number of corner cases should be handled carefully in this step:

- Since C_2 requested [f] rqM, it has a transient state SM when [c] rqI arrives. It should still be able to handle this invalidation request even in the transient state (while any processor requests stall). In this case, C_2 accepts [c] rqI 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 is in a transient state after making an invalidation request. It cannot take [d] rsI as well, since the response is not at the head 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 [34,33]. While there are other possible correct topology and network settings, the cases shown in Figure 1 at least demonstrate that it is nontrivial to construct one of them. Note that the three-channel system is *logical* in the sense that the actual hardware implementation may use various hardware components that can simulate the requirements.

In terms of making a protocol design correct, transient states, topology, and network settings contribute to make *interleavings* correct. Considering the sequence of rule executions [①; ③; ④] (in red) as an execution flow – we will later call it a *transaction* – to handle a processor request [a] rqWr, we see that the other execution flow (in blue) could not happen after ②, which is for another processor request [e] rqWr. As explained above case-by-case, proper transient states and network channels made [f] rqM stall. This mechanism to ensure safe interleavings is called *noninterference* [18,11], which ensures that no other transactions spuriously affect state transitions by an ongoing transaction.

Hemiola in a nutshell. If transient states, proper topology, and network settings are essential for designing a correct protocol, can we craft a DSL where *only conformant protocols are expressible*?

That is exactly what we did with Hemiola. The Hemiola DSL helps designers design cache-coherence protocols in a safe way. Instead of requiring designers to use transient states coupled to a protocol, we discover *general* stall conditions that by themselves ensure noninterference and form those conditions as conceptual locks. The stall conditions are *extracted and abstracted* from the usual transient states, so they can apply to practical protocols.

For instance, a designer may write a rule for ① without any DSL support like the left rule in the following code:

```
system memoryMSI {
  cache C1 {
    state status: MSI, value: valueT, in_transition: TrsMSI
    ...
    // Without any DSL support            |    // Using the Hemiola DSL
    rule getMRqUpUp {                     |    rule getMRqUpUp from template rquu {
      msgIn = procToC1.deq();             |      receive rqWr();
      assert (msgIn.id == rqWr);          |      assert (status == S);
      assert (!in_transition);            |      send rqM();
      assert (status == S);               |    }
      in_transition <= SM;                |
      c1ToPRq.enq({id: rqM, val: 0}); }   |
} }
```

Note that a designer has to find proper input/output channels (`procToC1` and `c1ToPRq`) and check/set a proper transient state (`in_transition`) in order to define the rule.

On the other hand, the left rule can be written more easily by using the Hemiola DSL as the right rule. Instead of using explicit channels and transient states, the right rule just uses the `rquu` *rule template* (where `rquu` stands for request-up-up). The rule templates employ *proven-safe* network structures and automatically check/set/release associated locks, so users can design protocols without worrying about incorrect use of network channels, locks, etc.

3 The Hemiola Domain-Specific Language

As explained in section 2, in designing a cache-coherence protocol, it is nontrivial to make concurrent execution of transactions correct. In this section, we introduce the Hemiola DSL to ease that burden. While conventional approaches deal with transient states directly to derive noninterference per-state, the Hemiola DSL limits protocols to satisfy *abstract conditions* that can guarantee noninterference by-construction. The conditions have already been mentioned in section 2 – network topology and locking mechanisms extracted from transient states of practical cache-coherence protocols.

Notations. An overline (e.g., $\overline{l}$) denotes a list. $[]$ and $(\overline{l} + e)$ denote nil and single-element append, respectively. $\oplus\overline{l}$ flattens the list of lists $\overline{l}$ with repeated concatenation. $(\overline{l_1} + \overline{l_2})$, $(\overline{l_1} - \overline{l_2})$, and $(\overline{l_1} \# \overline{l_2})$ denote append, subtraction, and disjointness of lists, respectively. We use the same operation $(+)$ for the

single-element and general append. Regarding a list of key-value pairs as a finite map, we override notations for lists. For example, $(M + \bar{l})$ 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. $(\overline{s.\mathsf{fd}})$ is used as a shorter notation for $(\mathsf{List.map}\ (\lambda s.\ s.\mathsf{fd})\ \bar{s})$. We use $\langle\cdot\rangle$ to denote a struct and use a name (e.g., $s.\mathsf{fd}$) to access a field value.

3.1 Syntax

The Hemiola DSL is similar to well-known rule-based hardware-description languages (HDLs) such as Bluespec [23], Kami [10], and Kôika [4]. A notable difference is that rule descriptions are restricted by predesigned *rule templates* to avoid spurious interleavings among transactions.

A *system* $S ::= \langle \overline{C}, \overline{i_{\mathrm{in}}}, \overline{i_{\mathrm{rq}}}, \overline{i_{\mathrm{rs}}} \rangle$ is the biggest unit of the language; it consists of caches $(\overline{C})$ and channel indices for internal messages $(\overline{i_{\mathrm{in}}})$ and external (processor) inputs$(\overline{i_{\mathrm{rq}}})$/outputs$(\overline{i_{\mathrm{rs}}})$. A *cache* $C ::= \langle i, s_{\mathsf{init}}, \bar{r} \rangle$ consists of its index (unique within a system), an initial state (s_{init}), and rules $(\bar{r})$. A *rule* (r) makes state transitions within the cache, and it is always defined by one of the rule templates provided by the language.

Each rule template must be instantiated with a rule index (should be unique within a cache), a precondition, and a transition function, where the types of the precondition and transition vary by template. A precondition of a rule template usually takes input messages and a (partial) current cache state and decides whether the rule can be executed or not. A transition function takes the same arguments in general but returns the next cache state and output messages. Neither state transition nor input-messages consumption happens if the precondition does not hold. We will introduce the detailed rule-template forms in the next section (section 3.2).

A *message* $m ::= \langle \mathsf{ty}, \mathsf{id}, \mathsf{val} \rangle$ is composed of a Boolean message type (request or response), a message ID (effectively from an enumeration of message kinds), and a value. We use *value* to refer to the set of legal contents of memory addresses. A pair $im ::= (i, m)$ is used sometimes to represent a message m in a channel with an index i.

3.2 Rule Templates

The Hemiola DSL follows syntax and semantics of traditional rule-based HDLs, but the major difference is that Hemiola further restricts the way of describing rules, which itself guarantees noninterference among transactions.

Topology and network requirements. First of all, Hemiola requires that the caches in a given system 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.

Note that the topology and network settings are required *logically*; the actual hardware implementation may use various hardware components (e.g., finite-capacity FIFOs or buses) that can simulate the requirements.

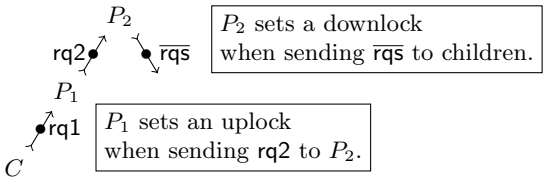


Fig. 2: Locking mechanism in Hemiola

Locking mechanism. We saw in section 2 why transient states are required to ensure noninterference in cache-coherence protocols. Revisiting the issue described in Figure 1, a child should be able to handle an invalidation request from the parent even if it is in a transient state (SM), and after handling the request it changes its transient state to IM.

Hemiola supports a locking mechanism reflecting this discovery; the locking is more general in that the framework looks at whether the message is from the parent or a child. This mechanism is still enough to describe practical cache-coherence protocols and sufficient to ensure noninterference.

In particular, Hemiola employs two kinds of locks: *uplocks and downlocks*. We say a cache is uplocked (or downlocked) when it holds an uplock (or downlock), respectively. Figure 2 depicts the locking mechanism in Hemiola. An uplock is set when a cache (P_1 in the figure) makes an upward request to its parent (P_2); it is released when the cache gets a corresponding response from the parent. The cache cannot make any further upward requests while uplocked. On the contrary, a downlock is set when a cache (P_2 in the figure) makes a downward request(s) to some of its children; similarly it is released when the cache gets corresponding response(s) from the child requestee(s). The cache cannot make any further downward requests while downlocked.

Now every cache defined in Hemiola *does not need to set transient states* to consider all possible combinations among stable statuses. For instance, instead of setting a transient state SM, it is now desirable to maintain its status S and *set an uplock* to record it just made an upward request. We emphasize that the Hemiola locks *do not enforce more restrictions* on protocols than what is enforced by transient states; e.g., as an uplock makes certain messages like `rqS` and `rqM` stall, a transient state SM makes them stall as well.

Each cache defined by Hemiola has a semantic lock state holding a lock type (uplock or downlock) and related messages/indices. The user, however, does not need to deal with this lock state while using the DSL; locks are managed implicitly by Hemiola.

Note that the DSL supports design of single-cache-line protocols, and thus the uplock and downlock are assigned per-line. The single-line protocol is then naturally extended to all cache lines using a protocol compiler that will be introduced in section 6. This approach is sound in terms of correctness, since a transaction does not affect coherence for the other lines.

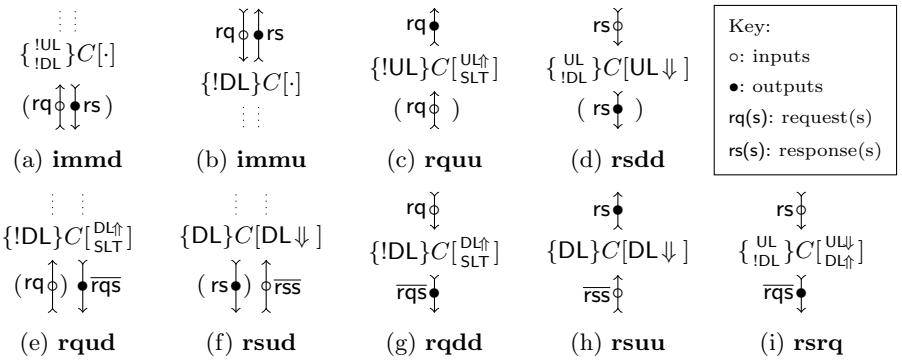


Fig. 3: Rule templates in Hemiola

The nine rule templates. Hemiola provides a set of rule templates for describing protocols in a way that guarantees noninterference by-construction. Figure 3 presents the nine rule templates. Each diagram has the form $\{P\}C[Q]$ and arrows (representing the directions of messages; e.g., a downward arrow indicates messages from a parent) with circles (○ for inputs and ● for outputs) and labels representing requests (rq(s)) and responses (rs(s)). It means that the rule template is for a cache C, requires input messages (○) with the message types determined by the label, has a precondition P, performs a state transition Q, and generates output messages (●). The precondition and state transition are implicit in the sense that they are automatically checked and performed, respectively, whenever the rule is executed. Note that some rule templates may make local state transitions without any input/output messages (input/output messages marked with parentheses in Figure 3).

UL, DL, !UL, and !DL in a precondition indicate that the cache should be uplocked, downlocked, uplock-free, and downlock-free, respectively. UL⇑, DL⇑, UL⇓, and DL⇓ 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.

The rule templates are carefully designed to avoid any spurious interleavings among transactions. We see a number of cases that are worth analyzing:

- **immu** and **rqdd** show that a cache can handle a downward request even when uplocked. These rules do not have a precondition that the cache should be uplock-free. This relaxation is necessary to avoid a deadlock.
- **rsdd** says that in order to handle a response from the parent, the cache should be downlock-free. This precondition is required to ensure noninterference.
- **rsrq** forces the order of a traversal, saying that the traversal for the outer caches must be done before traversing the inner caches. This rule is used when a transaction needs to traverse all the caches in the system, e.g., invalidating all the other caches to obtain the M status. The forced order is important to avoid a deadlock.

$$\text{SSilent:}\ \frac{}{s \xrightarrow[S]{l_\epsilon} s} \qquad \text{SIns:}\ \frac{\overline{im} \neq [] \quad \overline{im.i} \subseteq S.\overline{i_{\text{rq}}}}{\langle \overline{c}, M\rangle \xrightarrow[S]{l_{\text{in}}(\overline{im})} \langle \overline{c}, M + \overline{im}\rangle}$$

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

$$\text{SInt:}\ \frac{\begin{array}{cc} S = \langle \overline{C}, \overline{i_{\text{in}}}, \overline{i_{\text{rq}}}, \overline{i_{\text{rs}}}\rangle & C \in S.\overline{C} \quad r \in C.\overline{r} \\ \overline{im^{\text{ins}}.i} \subseteq S.\overline{i_{\text{in}}} \cup S.\overline{i_{\text{rq}}} & \overline{c}[C.i] = c_1 \quad \overline{im^{\text{ins}}} \subseteq M.\mathsf{hds} \\ r.p\,(c_1, \overline{im^{\text{ins}}}) & r.t\,(c_1, \overline{im^{\text{ins}}}) = (c_2, \overline{im^{\text{outs}}}) \\ \overline{im^{\text{outs}}.i} \subseteq S.\overline{i_{\text{in}}} \cup S.\overline{i_{\text{rs}}} & \overline{im^{\text{ins}}.i} \,\#\, \overline{im^{\text{outs}}.i} \end{array}}{\langle \overline{c}, M\rangle \xrightarrow[S]{l_{\text{int}}(C.i, r.i, \overline{im^{\text{ins}}}, \overline{im^{\text{outs}}})} \left\langle \begin{array}{l} \overline{c} + (C.i, c_2), \\ M - \overline{im^{\text{ins}}} + \overline{im^{\text{outs}}} \end{array} \right\rangle}$$

Fig. 4: Transition steps of the Hemiola DSL

4 Verification in Hemiola

We have introduced the Hemiola DSL in section 3 and provided an intuition that rule templates ensure general noninterference, i.e., interleavings among any transactions are safe. That said, we have not yet showed how the rule templates guarantee such noninterference in a formal way. We also have not explained how noninterference eases the verification of cache-coherence protocols.

In this section, we provide the semantics of the Hemiola DSL and the formal meaning of general noninterference called *serializability*. We then introduce our novel approach to proving invariants called *predicate messages*, which eliminates the burden of considering interference while proving invariants.

4.1 Semantics of the Hemiola DSL

A system in Hemiola follows so-called "one-rule-at-a-time semantics" [5,10,34,4], i.e., any state transition by concurrent rule executions can be interpreted as a serial execution of rules. Thus, it is fair to consider that a state transition happens by executing a single rule.

Transition steps. Figure 4 describes the complete semantics for transition steps of the Hemiola DSL. 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 $\langle \overline{c}, M\rangle$ of cache states ($\overline{c}$) and message states (M). Cache states are represented in a finite map from cache indices to cache states, and message states are represented in a finite map from channel indices to ordered queues of messages.

Rule [SSilent] represents the case where no state transition happens in the current step; an empty label (l_ϵ) is generated in this case. From now on, we assume that all the input/output messages used in the step definitions do not

share the same channel, i.e., ($\mathsf{List.NoDup}\ \overline{im.i}$). [SIns] describes the case for external input messages coming to the system; an external-inputs label ($l_{\text{in}}(\overline{im})$) is generated in this case. [SOuts] describes the opposite case, for output messages being released to the external world, generating an external-outputs label ($l_{\text{out}}(\overline{im})$).

Lastly, [SInt] deals with a state transition by a rule (r) in a cache (C). It nondeterministically chooses a cache and a rule in the cache, checks that the precondition holds, and applies the transition to update the state of the system; an internal label ($l_{\text{int}}(C.i, r.i, \overline{im^{\text{ins}}}, \overline{im^{\text{outs}}})$) is generated in this case, which records a cache index, a rule index, input messages, and output messages. Note that the semantics is based on ordered channels, so messages are *enqueued* and *dequeued* in each state-transition case.

The step semantics is naturally lifted to one for *multiple steps*, presented as a judgment $s_0 \overset{\bar{l}}{\underset{S}{\Rightarrow}} s_1$, where $\bar{l}$ is a sequence of labels generated by executions of the steps in order. We will sometimes call such 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. We use a simpler notation $S \Rightarrow s$ for reachable states. We also call such a history $\bar{l}$ *legal*, denoted as $S \overset{\bar{l}}{\Rightarrow} \bullet$. We call $\mathcal{I} : \mathbb{S} \to \mathbb{P}$[2] an *invariant* over a system S if $\mathcal{I}$ holds for all reachable states, i.e., $\forall s.\ (S \Rightarrow s) \to \mathcal{I}(s)$.

Behaviors and correctness. A system S has a behavior $\lfloor \bar{l} \rfloor$ (denoted as $S \Downarrow \lfloor \bar{l} \rfloor$) iff $S_{\mathsf{init}} \overset{\bar{l}}{\underset{S}{\Rightarrow}} s$ holds, where $\lfloor \cdot \rfloor$ 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*. Lastly, we say that 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:

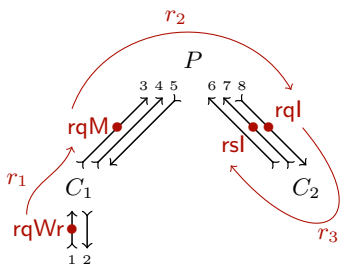
$$I \sqsubseteq S \triangleq \forall \bar{t}.\ I \Downarrow \bar{t} \to S \Downarrow \bar{t}.$$

In order to prove trace refinement, we usually establish a *simulation relation* [6] between the implementation and the spec states and prove that the relation is preserved over steps, and it is crucial to state and prove proper invariants of the implementation for the simulation proof. Since the invariant proof is indeed the most significant part of the whole correctness proof, in this paper we would like to focus on how Hemiola helps a user state and prove invariants.

4.2 Serializability in Hemiola

Serializability [28,3] is a celebrated notion of concurrency correctness. While each transaction in a system affects multiple values, serializability guarantees that interleaved execution of such transactions is correct in that the effect (state change) is the same as if the transactions were executed serially, i.e., *atomically in some order with no interleaving*.

[2] $\mathbb{P}$ is Prop in Coq, which can reasonbly be interpreted as Boolean in this paper.



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

* The atomic predicate:

$$[(1, \mathsf{rqWr})] \overset{h}{\rightsquigarrow\!\triangleright} [(7, \mathsf{rsI})]$$

Fig. 5: An example of an atomic history

In order to define serializability formally, we first provide basic definitions of atomic histories and transactions. A history h is *atomic* iff it satisfies the predicate $(\overline{im^{\text{init}}} \overset{h}{\rightsquigarrow\!\triangleright} \overline{im^{\text{end}}})$ with *initial messages* $\overline{im^{\text{init}}}$ and *live messages* $\overline{im^{\text{end}}}$, constructed inductively by the following two cases:

- Any singleton history with an internal label is an atomic history with its input and output messages as initial and live messages, respectively.
- If h is an atomic history, $(h{+}l)$ is also an atomic history if l *consumes* its input messages from the live messages of h. The new live messages are constructed by subtracting the input messages and adding the output messages of l to the previous live messages.

Figure 5 presents an atomic history already shown in Figure 1. h is generated by executions of three rules, $r_1 \in C_1.\overline{r}$, $r_2 \in P.\overline{r}$, and $r_3 \in C_2.\overline{r}$. Rule r_1 takes an input message $(1, \mathsf{rqWr})$ (from the channel with index 1) as an initial message of the history. Rule r_2 takes $(3, \mathsf{rqM})$, the output message from r_1. Finally, r_3 takes $(8, \mathsf{rqI})$, the output message from r_2. Summing up all the rule executions, by the definition of an atomic history we get the predicate lower-right in Figure 5.

This example shows that an atomic history intuitively captures a *transaction flow* triggered by the initial messages. Note that an atomic history does not need to be completed, e.g., h in the example is incomplete in the sense that the live message (rsI) is not a response sent to an external channel.

We call an atomic history $(\overline{im^{\text{init}}} \overset{h}{\rightsquigarrow\!\triangleright} \overline{im^{\text{end}}})$ a *transaction* if its initial messages are external requests $(\overline{im^{\text{init}}.i} \subseteq S.\overline{i_{\text{rq}}})$; we denote it as $S \not\swarrow h$.

With a clear notion of transactions, we can now easily define sequential histories and serializability. A history h is *sequential* iff the history is a concatenation of transactions:

$$\mathsf{Sequential}\ S\ h \triangleq \exists\, \overline{t}.\ (\forall t \in \overline{t}.\ S \not\swarrow t) \wedge h = \oplus \overline{t}.$$

A legal history h is *serializable* in the system S iff there exists a sequential history that reaches the same state:

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

A system S is *serializable* iff every legal history is serializable:

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

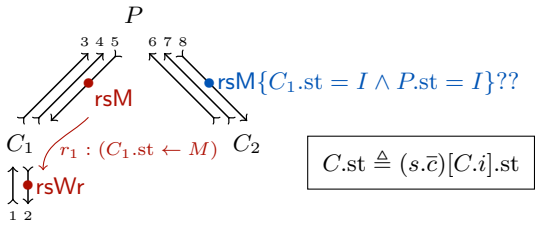


Fig. 6: Interference breaks a predicate message

4.3 Predicate Messages

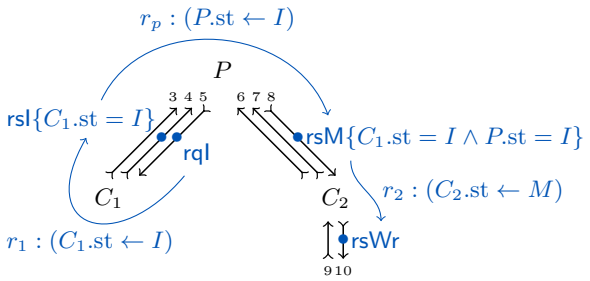
Now we discuss how to exploit our notion of serializability: how does it help prove global invariants of a system? In proving the correctness of a cache-coherence protocol, it is very common to state an invariant like "an important property holds *whenever the system includes a certain message in a certain channel*." We call such an invariant a *predicate message*, giving the intuition of messages that logically carry predicates that must be true so long as those messages remain in play. More formally, $S \vdash im\{P\} \triangleq \forall s.\ (S \Rightarrow s) \rightarrow im \in s.M \rightarrow P(s)$, where $s.M$ refers to the message state of the system. We will write just $im\{P\}$ when the system S is clear from context, also often using a shorter version $\mathsf{id}\{P\}$ (considering only messages with a given ID) when it is not ambiguous.

Figure 6 presents an example of a predicate message. When a child C_2 is about to handle a response message rsM, which is a permission to change the cache status to M, we expect the parent and the other child C_1 to have I status (like $\{C_1.\mathrm{st} = I \wedge P.\mathrm{st} = I\}$ in the figure). However, between the sending of that message and receipt by C_2, the predicate may be broken *by another transaction*; for instance, the predicate no longer holds if a state transition happens by $r_1 \in C_1$, which takes another $(5, \mathsf{rsM})$ and updates the status of C_1 to M.

Investigating this corner case carefully, we find that actually no two different rsM messages can be in the system at the same time. It 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:

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

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 interference into account*.



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

Fig. 7: Predicate messages defined as an atomic invariant

Our novel approach to employing predicate messages in atomic histories begins with formalizing the notion of atomic invariants. We say that $\mathcal{I}_A : \overline{\mathbb{IM}} \times \mathbb{S} \rightarrow \mathbb{P}$ is 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 $\overline{im_i} \overset{h}{\rightsquigarrow} \overline{im_o}$.

Figure 7 shows an example of predicate messages defined in an atomic history, formalized as an atomic invariant. An atomic invariant $\mathcal{I}_A$ is a conjunction of clauses $(im \in \overline{im_o} \rightarrow P(s))$, each claiming that the predicate P holds when im is in *the live messages* $\overline{im_o}$. We can prove that the atomic invariant $\mathcal{I}_A$ holds by induction on state-transition steps through the atomic history in the figure:

- 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 $(4, \mathsf{rsl})\{C_1.\text{st} = I\}$. Since r_p changes the status of P to I, we can prove the predicate for $(8, \mathsf{rsM})$.
- 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 $(10, \mathsf{rsWr})$.

Note 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 definition, for every reachable state there is a sequential history that reaches the same state. Since the sequential history is a concatenation of transactions, an invariant can be proven *by showing that any transaction preserves it.*

Since a transaction is an (external) atomic history, we can make use of corresponding 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}):

$$\mathcal{I}_A\,(\overline{m_i}, s_i) \wedge \mathcal{I}(s_i) \rightarrow (\mathcal{I}_A\,(\overline{m_{i+1}}, s_{i+1}) \wedge \mathcal{I}(s_{i+1})).$$

For instance, in proving a cache-coherence protocol, we usually want to have an invariant claiming that at most one node of the system has M status at a time. The predicate messages defined in Figure 7 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 (section 5).

4.4 Serializability Guarantee by the Hemiola DSL

The biggest contribution of the Hemiola framework includes the serializability proof. The highest-level theorem simply claims that use of good topology ($\mathsf{OnTree}\ S\ t$) and the rule templates ($\mathsf{GoodRules}\ S\ t$) guarantees serializability:

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

In the proof we used a well-established technique called commuting reductions [15], showing that any interleaving transactions can be serialized by performing a finite number of reductions. Interested readers are referred to Choi's dissertation [9], which describes more details of the proof.

5 Case Studies: Hierarchical MSI and MESI Protocols

In this section we explain how we designed, specified, and formally proved the correctness of the following three hierarchical cache-coherence protocols: inclusive/noninclusive MSI protocols and a noninclusive MESI protocol. Each protocol is *parameterized by a tree* 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*.

The protocols are directory-based and support arbitrary evictions. The inclusive MSI protocol requires back-invalidation to maintain the cache-line inclusion [30]. The noninclusive protocols employ the noninclusive-cache inclusive-directory (NCID) [38] structure to optimize cache space.

We will introduce common points among our case-study protocols. Particularly, we focus on *how predicate messages are used* (introduced in section 4.3) to ease the invariant proofs required to prove protocol correctness. More details about the correctness proofs are provided in Choi's dissertation [9].

5.1 Cache States

A cache state consists of a status, a value, a directory, and a Boolean called an ownership bit. A status is either `M`, `E`, `S`, or `I`. The MESI protocol applies further optimizations to the MSI protocol: if a cache line has `E` status, then the line is exclusive to the cache but also clean.

A directory contains a status of its children called a directory status and a list of child-cache indices that have the directory status. An L1 cache does not have a directory since it has no children.

The *ownership bit* decides whether the cache is responsible for writing the value back to the parent when evicted. The ownership bit intuitively constrains which caches can have valid status; we will see how this intuition is formalized as an invariant in section 5.3.

5.2 Protocol Description with Rule Templates

We present a number of rule descriptions, used in our case studies, 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 define each rule template compactly.

```
rule l1GetMRqUpUp from template rquu {
  receive rqWr();
  assert (status != M);
  send rqM();
}
```

The above code presents an actual rule definition in an L1 cache, starting with an invocation of a particular rule template `rquu`, which takes an upward request (to the cache) and sends a further request to the parent. This rule `receive`s a message with the ID `rqWr` from the processor core[3] to get a write permission. This rule template also requires to write down the precondition (`assert`) and the output message (`send`). In this example the cache simply forwards `rqM` to the parent. As explained in section 3.2, the `rquu` rule template does not allow any state transition except locking – the template automatically sets an uplock.

```
rule liDownIRsUpDownM from template rsud {
  receive downRsI();
  hold {rsbTo, rqM()};
  status <= I;
  dir <= M [rsbTo];
  owned <= false;
  send rsM();
}
```

The above rule presents another case that sends the response to the child who requested `rqM`. Template `rsud` says that the rule takes responses from children and responds back to the original child requestor. The rule `receive`s the response message with the ID `downRsI`. In order to execute this rule, the cache should `hold` a downlock containing the index of the original requestor (`rsbTo`) and the request message with the ID `rqM`, acting like an assertion for the lock state.

As a state transition, this rule sets its status to `I`, sets the directory status to `M` by adding the requestor, and sets the ownership bit as false since the requestor will make the value dirty after it obtains `M`. It also `send`s a response (`rsM`) to the requestor. Lastly, the downlock is released *automatically and implicitly* by the `rsud` rule template.

```
rule liGetSImmME from template immd {
  receive rqS() from cidx;
  assert (status == E || status == M);
  assert (dir.status == I);
  status <= I;
  dir <= E [cidx];
  send rsE(value);
}
```

The above rule is for the MESI protocol, fired when an intermediate cache gets a request from a child to read the data, while the parent has status `E` or `M`.

[3] It is a rule defined in an L1 cache, thus an upward request is from the core.

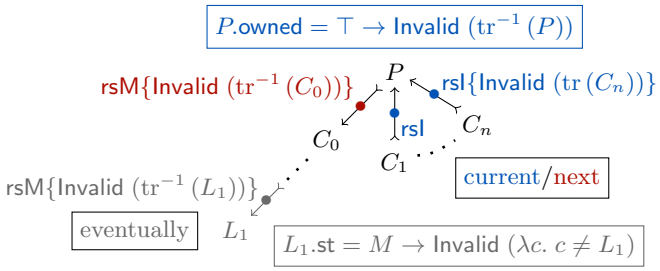


Fig. 8: Use of predicate messages in the case-study protocols

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.

5.3 Invariant Proof Using Predicate Messages

Now we present how predicate messages (introduced in section 4.3) are used to prove a nontrivial invariant required for all of our three case-study protocols.

Figure 8 shows a coordination between predicate messages and conventional invariants. Suppose that an L1 cache (shown as L_1 in gray in the figure) requested to the parent to get the `M` status. When it finally handles the response rsM, it should know all the other caches (except itself) have been invalidated to prove the desired invariant about `M` (denoted as $L_1.\mathsf{st} = M \rightarrow \mathsf{Invalid}\ (\lambda c.\ c \neq L_1)$). This proof case can be supported using the predicate message for rsM, stating $\mathsf{Invalid}\ (\mathrm{tr}^{-1}(C))$ (the caches outside of the subtree rooted to C are invalid) when the message goes to C. Since L_1 is a 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 also shows another coordination to prove a predicate message. When a child C_i sends the invalidation response rsI, it should know that all the caches inside the subtree of C_i have been invalidated (denoted as $\mathsf{Invalid}\ (\mathrm{tr}(C_i))$). When the parent P subsequently handles the responses, it responds with rsM to the original requestor (C_0 in the figure), requiring to prove $\mathsf{Invalid}\ (\mathrm{tr}^{-1}(C_0))$, the predicate message for rsM.

While P also changes its status to `I` in this state transition, how do we infer that the caches outside P have already been invalidated, which is required to prove the predicate over rsM? In this case, we should know that 1) P has the ownership bit true (from a simple cache-level invariant of P) and 2) the caches outside of a cache with ownership bit set should have `I` status (denoted as $P.\mathsf{owned} = \top \rightarrow \mathsf{Invalid}\ (\mathrm{tr}^{-1}(P))$) as an invariant. Combining all the predicates and the state transition by P, we can prove the next predicate message for rsM to the original requestor C_0.

6 Compilation and Synthesis to Hardware

So far we have dealt with cache-coherence protocols for a single line. In order to build a hardware-synthesizable multiline implementation, we developed

a compiler that takes a single-line Hemiola protocol and generates a multiline implementation described in Kami [10].

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. 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.

6.1 Compilation of Hemiola Protocols

The compiler uses prebuilt hardware components described in Kami. One of them is NCID [38], whose interfaces include asynchronous read and write of the line status and value. Another prebuilt component holds a *finite* number of miss-status holding registers (MSHRs), whose abstract interface includes registering, updating, and releasing MSHRs with respect to their types (uplock or downlock) and locking addresses. The compiler also takes a cache configuration as an argument to set the capacity of a cache, the number of MSHRs, etc.

One of the biggest differences between a source Hemiola protocol and the target Kami implementation is that the target accesses multiple lines *asynchronously*. In the source protocol, a single-line value is read (or written) *immediately*, whereas in the target the value is accessed first by making a read (or write) request to a cache and next by handling the response. In order to optimize such line accesses, the compiler uses a prebuilt pipeline to deal with multiple line accesses in parallel.

6.2 Synthesis of Hemiola Protocols

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 [23] implementation and synthesize it to load on an FPGA.

Before synthesis, we first evaluated two Hemiola protocols, $\mathsf{Hemiola}_2$ and $\mathsf{Hemiola}_3$, instantiated from our hierarchical noninclusive MESI protocol described in section 5, using the Bluespec simulator. $\mathsf{Hemiola}_3$ 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 last-level cache. $\mathsf{Hemiola}_2$ 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 [37], featuring a 2-level inclusive MESI protocol with self-invalidation [30]. We set the cache sizes of RiscyOO the same as for $\mathsf{Hemiola}_2$.

Figure 9 shows the performance result. We measured performance by counting the number of transactions performed in 5×10^5 simulation cycles, with various workloads that make random requests but mimic some amount of temporal/spatial locality of memory accesses. 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, also shown in Figure 9. We used Xilinx's Virtex-7 VC707 FPGA [1] for synthesis. Each protocol uses a minimal

Performance (#trs/cycle)	all-shared	pair-shared	ex:sh=1:1	ex:sh=4:1
Hemiola$_3$	0.259	0.868	0.506	0.764
Hemiola$_2$	0.270	0.800	0.637	0.913
RiscyOO	0.336	0.791	0.637	0.988

	Clock length	Critical path	#LUTs	#FFs
Hemiola$_2$	40 ns	36.861 ns	126,714	41,203
Hemiola$_3$	40 ns	37.608 ns	240,034	61,011

Fig. 9: Evaluation and synthesis of Hemiola protocols

clock length that can safely cover its critical path. Both Hemiola$_3$ and Hemiola$_2$ 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 safety and liveness.

Optimization and verification of the cache-controller design are nontrivial; the pipeline requires correct stall logic, which is as sophisticated as the logic in pipelined processors. While the verification of the pipeline is one of our future-work directions, we see it as orthogonal to the verification of cache-coherence protocols, our focus with Hemiola.

7 Related Work

Model checking. Model checking has long been widely used to verify cache-coherence protocols. Various model checkers like Murphi [12], SMV [20], and TLA+ [14,13] have been used.

In order to overcome the usual state-space-explosion problem, model checkers have developed noninterference lemmas to deal with the state-space explosion by interleavings [18,11]. In order to obtain effective lemmas, a number of approaches used descriptions in terms of transactions (called "message flows") [32,24,31]. 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 [36,35,2]. 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 modularity in protocol design and successfully verified hierarchical cache-coherence protocols [7,8,16,17]. The enforced modularity, however, made it hard to design and verify noninclusive protocols. [7,8] tried to solve this problem using assume-guarantee reasoning and history variables, while still maintaining the concept of compositional verification, but faced state-space explosion again, and thus they just verified a two-level MSI protocol with three L2 caches. [16,17] have developed the Neo theory as a safe way to compose "subtrees" of caches to have a hierarchical protocol. They argued it is possible to verify noninclusive protocols

in the Neo framework when a directory is still inclusive but did not provide the actual design and proof. We provided the proofs of hierarchical noninclusive cache-coherence protocols in Hemiola, without any such restrictions.

Another notable success of cache-coherence verification employed program synthesis to generate a protocol for a given atomic specification [26,25]. The ProtoGen/HieraGen synthesizer can generate various hierarchical protocols including 3-hop protocols and even unconventional protocols like TSO-CC but does not support noninclusive protocols as well. Furthermore, they used Murphi to verify synthesized protocols, but in ProtoGen [26] they only succeeded up to three caches without exhausting memory, and in HieraGen [25] they succeeded only with the root, two cache-H, and two cache-L nodes. Since Hemiola supports noninclusive protocols but not 3-hop ones, we see protocol-design-space coverage between Hemiola and ProtoGen/HieraGen as incomparable. That said, in terms of verification, Hemiola provides a much higher level of formal assurance by allowing verification of protocols with arbitrary tree topologies.

Theorem proving. Theorem proving also has been used to verify cache-coherence protocols. A number of works proved correctness of specific protocols [29,22]. A recent success was a proof of a hierarchical MSI protocol with an arbitrary tree topology using Coq [34], but it was not structured to promote streamlined reuse of results for other protocols. It also included rather complex and ad-hoc invariants that needed to characterize transient states.

Another notable project designed a modular-specification approach for cache coherence, verifying each cache against the spec while generating/proving invariants automatically, using the Ivy verification tool [27,19,21]. While in Hemiola a user should state and prove invariants manually, the framework provides serializability as a large essential invariant that can be reused by various protocols, and then invariants become easier to prove on top of it.

8 Conclusion

We have developed a framework called Hemiola for simplified design and formal proof of cache-coherence protocols. The template-based DSL ensures that the only protocols that can be expressed are those that admit a form of per-memory-access serializability. On top of the framework, we proved the correctness of hierarchical MSI and MESI protocols as case studies, demonstrating that Hemiola indeed eases proof burden. We also built a protocol compiler and demonstrated these protocol implementations running on FPGAs.

Acknowledgements We thank our reviewers for their feedback and detailed comments. This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Grant No. HR001118C0018. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. 7 Series FPGAs Configurable Logic Block – User Guide (September 2016), https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf
2. Banks, C.J., Elver, M., Hoffmann, R., Sarkar, S., Jackson, P., Nagarajan, V.: Verification of a lazy cache coherence protocol against a weak memory model. pp. 60–67. FMCAD '17, Austin, TX, http://dl.acm.org/citation.cfm?id=3168451.3168470
3. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc. (1987)
4. Bourgeat, T., Pit-Claudel, C., Chlipala, A., Arvind: The essence of Bluespec: A core language for rule-based hardware design. pp. 243–257. PLDI 2020, New York, NY, USA. https://doi.org/10.1145/3385412.3385965
5. Braibant, T., Chlipala, A.: Formal verification of hardware synthesis. CAV 2013, vol. 8044, pp. 213–228. Springer, http://gallium.inria.fr/~braibant/fe-si/
6. Brookes, S.D., Rounds, W.C.: Behavioural equivalence relations induced by programming logics. In: Diaz, J. (ed.) Automata, Languages and Programming. pp. 97–108. Springer Berlin Heidelberg, Berlin, Heidelberg (1983)
7. Chen, X.: Verification of Hierarchical Cache Coherence Protocols for Futuristic Processors. Ph.D. thesis, USA (2008)
8. Chen, X., Yang, Y., Gopalakrishnan, G., Chou, C.T.: Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. Form. Methods Syst. Des. **36**(1), 37–64 (Feb.). https://doi.org/10.1007/s10703-010-0092-y
9. Choi, J.: Structural design and proof of hierarchical cache-coherence protocols. Ph.D. thesis (2021), https://hdl.handle.net/1721.1/130759
10. Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., Arvind: Kami: A platform for high-level parametric hardware specification and its modular verification. Proc. ACM Program. Lang. **1**(ICFP), 24:1–24:30. https://doi.org/10.1145/3110268
11. Chou, C.T., Mannava, P.K., Park, S.: A simple method for parameterized verification of cache coherence protocols. pp. 382–398. FMCAD '04, Springer Berlin Heidelberg, Berlin, Heidelberg
12. Dill, D.L.: The Murphi verification system. pp. 390–393. CAV '96, Springer-Verlag, London, UK, UK (1996), http://dl.acm.org/citation.cfm?id=647765.735832
13. Joshi, R., Lamport, L., Matthews, J., Tasiran, S., Tuttle, M., Yu, Y.: Checking cache-coherence protocols with TLA+. Form. Methods Syst. Des. **22**(2), 125–131 (Mar 2003). https://doi.org/10.1023/A:1022969405325
14. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc. (2002)
15. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (Dec 1975). https://doi.org/10.1145/361227.361234
16. Matthews, O., Bingham, J., Sorin, D.J.: Verifiable hierarchical protocols with network invariants on parametric systems. pp. 101–108. FMCAD '16, FMCAD Inc, Austin, Texas (2016)
17. Matthews, O., Sorin, D.J.: Architecting hierarchical coherence protocols for push-button parametric verification. pp. 477–489. MICRO '17 (2017). https://doi.org/10.1145/3123939.3123971
18. McMillan, K.L.: Verification of infinite state systems by compositional model checking. pp. 219–237. CHARME '99, Springer Berlin Heidelberg, Berlin, Heidelberg
19. McMillan, K.: Modular specification and verification of a cache-coherent interface. pp. 109–116. FMCAD '16, FMCAD Inc, Austin, TX (2016), http://dl.acm.org/citation.cfm?id=3077629.3077651

20. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, USA (1993)
21. McMillan, K.L., Padon, O.: Ivy: A multi-modal verification tool for distributed algorithms. pp. 190–202. CAV 2020
22. Moore, J.S.: An ACL2 proof of write invalidate cache coherence. pp. 29–38. CAV '98, Springer-Verlag, London, UK, UK (1998), http://dl.acm.org/citation.cfm?id=647767.733778
23. Nikhil, R.: Bluespec System Verilog: efficient, correct RTL from high level specifications. pp. 69–70 (2004). https://doi.org/10.1109/MEMCOD.2004.1459818
24. O'Leary, J., Talupur, M., Tuttle, M.R.: Protocol verification using flows: An industrial experience. pp. 172–179. FMCAD '09. https://doi.org/10.1109/FMCAD.2009.5351126
25. Oswald, N., Nagarajan, V., Sorin, D.J.: HieraGen: Automated generation of concurrent, hierarchical cache coherence protocols. pp. 888–899. ISCA '20. https://doi.org/10.1109/ISCA45697.2020.00077
26. Oswald, N., Nagarajan, V., Sorin, D.J.: Protogen: Automatically generating directory cache coherence protocols from atomic specifications. pp. 247–260. ISCA '18, Piscataway, NJ, USA. https://doi.org/10.1109/ISCA.2018.00030
27. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: Safety verification by interactive generalization. SIGPLAN Not. **51**(6), 614–630 (Jun 2016). https://doi.org/10.1145/2980983.2908118
28. Papadimitriou, C.: The Theory of Database Concurrency Control. Computer Science Press, Inc., USA (1986)
29. Park, S., Dill, D.L.: Verification of FLASH cache coherence protocol by aggregation of distributed transactions. pp. 288–296. SPAA '96, ACM, New York, NY, USA. https://doi.org/10.1145/237502.237573
30. Ros, A., Kaxiras, S.: Complexity-effective multicore coherence. pp. 241–252. PACT '12, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2370816.2370853
31. Sethi, D., Talupur, M., Malik, S.: Using flow specifications of parameterized cache coherence protocols for verifying deadlock freedom. In: Automated Technology for Verification and Analysis. pp. 330–347 (2014)
32. Talupur, M., Tuttle, M.R.: Going with the flow: Parameterized verification using message flows. FMCAD '08
33. Vijayaraghavan, M.: Modular Verification of Hardware Systems. Ph.D. thesis (2016), http://hdl.handle.net/1721.1/106096
34. Vijayaraghavan, M., Chlipala, A., Arvind, Dave, N.: Modular deductive verification of multiprocessor hardware designs. pp. 109–127. CAV 2015
35. Zhang, M., Bingham, J.D., Erickson, J., Sorin, D.J.: PVCoherence: Designing flat coherence protocols for scalable verification. pp. 392–403. HPCA '14. https://doi.org/10.1109/HPCA.2014.6835949
36. Zhang, M., Lebeck, A.R., Sorin, D.J.: Fractal coherence: Scalably verifiable cache coherence. pp. 471–482. MICRO '10, USA. https://doi.org/10.1109/MICRO.2010.11
37. Zhang, S., Wright, A., Bourgeat, T., Arvind, A.: Composable building blocks to open up processor design. pp. 68–81. MICRO '18. https://doi.org/10.1109/MICRO.2018.00015
38. Zhao, L., Iyer, R., Makineni, S., Newell, D., Cheng, L.: NCID: A non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. pp. 121–130. CF '10, New York, NY, USA. https://doi.org/10.1145/1787275.1787314