Towards Composable Proofs of Cache Coherence Protocols

Martina Camaioni
EPFL
Lausanne, Switzerland
martina.camaioni@epfl.ch

Yann Herklotz
EPFL
Lausanne, Switzerland
yann.herklotz@epfl.ch

Tz-Ching Yu
EPFL
Lausanne, Switzerland
tz-ching.yu@epfl.ch

Thomas Bourgeat
EPFL
Lausanne, Switzerland
thomas.bourgeat@epfl.ch

Abstract

Verifying cache coherence protocols is a notoriously difficult problem. At the intersection between distributed protocol and computer architecture, it has long served as a premier target for formal methods. Current verification approaches hinge on the challenging discovery of large global inductive invariants. This paper introduces an alternative proof strategy that tames the complexity through two main contributions: protocol decomposition and a framework of local invariants.

We prove the correctness of the standard *MSI* protocol compositionally by independently verifying the simpler *MI* and *SI* subprotocols and proving *MSI* behaves as their combination. This decomposition revealed a useful insight: the invariants necessary in these proofs can be established using a small set of simple local invariants: invariants between a single cache and the parent and memory, eliminating complex global reasoning across children caches. We demonstrate how our approach reduces the number of required invariants from several dozens to hundreds in prior work to a small, structured and manageable set, simplifying the proof of correctness. The entire development is formalized in Lean 4.

CCS Concepts: • **Hardware → Theorem proving and SAT solving**.

Keywords: Lean, cache coherency, MSI, formal verification, theorem proving

ACM Reference Format:
Martina Camaioni, Yann Herklotz, Tz-Ching Yu, and Thomas Bourgeat. 2026. Towards Composable Proofs of Cache Coherence Protocols. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3779031.3779106

CPP '26, Rennes, France

ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779106

1 Introduction

The correctness of cache coherence protocols is fundamental to the performance and reliability of modern multi-core processors. A subtle bug in these protocols can lead to the worst kind of bugs: silent data corruption. This makes these protocols a prime target for formal verification.

The problem has been extensively studied over the past thirty years, and existing verification techniques have successfully verified representative protocols like *MSI* or *MESI*. However a significant challenge is left: full proofs still require heroic efforts. A primary bottleneck is the *manual* effort and expertise required to discover the precise global inductive invariants needed to prove correctness. These invariants are composed of several dozens [Choi et al. 2022; Vijayaraghavan 2016] to hundreds [Tan et al. 2025] (depending on the expressivity of the logic in which they are written) of intertwined logical formulas. The invariants are typically slowly iteratively discovered, a task hard for engineers to conceptualize and write, and out-of-reach for automatic tools to synthesize in the general unbounded case.

These large invariants become especially difficult to maintain and update when facing the seemingly minor design modifications that architects might want to explore: variations over the very precise *MSI* transitions used, small adjustments in the underlying cache topology or the downgrade policies, variations in the assumptions about the reordering induced by the network-on-chip interconnect, the associativity of the caches etc. The problem becomes even more major when architects decide to complexify the protocol, going from *MSI* to *MESI*, *MOSI*, *MOESI*, requiring virtually a complete proof overhaul.

This paper introduces a novel proof strategy that harnesses the proof complexity through two main insights: decomposition of the *MSI* protocol into simpler protocols and systematic simplification of the invariants.

More precisely, for the first part, instead of reasoning directly about the full *MSI* protocol, correctness is first established for two foundational subprotocols: the *MI* and *SI* protocols. Once these subprotocols are verified in isolation, we prove that the *MSI* protocol behaves as the composition of these two subprotocols. This modular approach significantly reduces proof complexity. Verifying *MI* and *SI* independently is simpler because they involve smaller state spaces, fewer types of messages exchanged between caches and the central memory, and a more limited set of possible configurations.

Addressing these smaller problems not only simplified the overall proofs but also led us to study invariants for these minimal protocols. In the process, we found simpler but still complete inductive invariants. We realized we could make the inductive invariants for the simple *MI* and *SI* protocol to share the same structural pattern, despite the seemingly distinct nature of the protocols, and that the *MSI* invariant could be described exactly as a simple combination of these, reducing the invariants to a size where they can fit on half a page. This exploration led us to the definition of a more general taxonomy of cache protocol invariants, providing a systematic framework for reasoning and supporting invariant crafting and reuse across different protocols.

Especially interesting, these new inductive invariants are not global, as was the case for previous work, but are expressed fully locally. Our local invariants are defined with respect to a single cache and the central memory, without requiring explicit reasoning about interactions between signals across different child caches. This perspective substantially simplifies invariant discovery, because it is only necessary to consider the interactions between that cache and the central memory. These local invariants can then be applied to other cache–memory subsystems, as the central memory is shared and all caches exhibit identical behavior.

Finally, we observed that introducing a controlled degree of nondeterminism and generalization into the protocol verified significantly eased some proofs while making the protocol more general, leading to proofs that are more reusable across small design variations.

In summary, the main contributions of this paper are the following:

- Definition of an invariant taxonomy applicable to the three studied protocols: *MSI*, *MI*, and *SI* and their variations (see section 3), where the inductive invariants are all local invariants, rather than global system-level properties.
- Construction of the composition of *MI* and *SI*, including a proof strategy that first establishes that *MSI* behaves as *MI* + *SI* and then verifies the correctness of *MI* and *SI*, thereby ensuring the correctness of *MI*+*SI* and consequently *MSI* (see section 4).
- Introduction of nondeterminism to simplify proofs and enhance protocol generality (see section 4).
- Extension of the proof methodology from a single cache line to multiple cache lines (see section 5).

In the rest of this paper, we first we give some background to cover the basics of cache-coherency protocols, the standard notion of refinement we use to define the correctness of cache-coherency protocol, and we give an intuition for the traditional challenges involved in finding global inductive invariants to prove correctness of cache-coherency protocols. We then first tackle the verification and formalization of a minimalist *MI* system, that lead us to introduce our taxonomy of local invariants for cache-coherency protocols and carving some simple but reusable insights. We then show that going from the simple *MI*, the seemingly more complicated *MSI* invariants is counterintuitively very straightforward.

We then elaborate on our decomposition of the *MSI* protocol, our use of nondeterminism, the extension to multiple cache lines, and an evaluation with randomized testing of a manual translation of the code to a RTL (hardware) implementation, to sanity-check that our design indeed works. We finally contrast our approach with the existing literature.

All the proofs presented are formalized in the Lean4 proof assistant, and the development is open-source.

2 Background and Motivation

This section introduces the *MSI* protocol and provides an intuitive characterization of its correctness. It then highlights the inherent difficulties in reasoning about such protocols and explains why establishing the correctness of *MSI*, in particular, constitutes a challenging verification problem.

2.1 MSI protocol description

In an *MSI* protocol, there is a central directory, called the parent, which stores multiple cache lines. Each cache line in the parent contains a memory address, the value associated with that address, and its current state. The system also includes multiple caches, called children, which can request the parent to store one or more of these cache lines locally. By doing so, a child can perform read and write operations on the data locally, without contacting the parent every time. Intuitively, we want our *MSI* protocol to provide the same properties as a centralized system, where there are no caches and each CPU can access the memory directly (figure 1).

To achieve this, we must ensure that in our shared-memory system, every cache can always read and write the most recently updated value. To maintain this coherence property, each cache line can be in one of three possible states: Modified (`M`), Shared (`S`), or Invalid (`I`). When a cache line is in the `M` state, the cache has exclusive ownership and can both read from and write to it. In the `S` state, the cache has read-only access, while in the `I` state, the cache cannot read or write to the line.

Figure 2 shows an example of the execution of the protocol, where one cache is in the `M` state, which means it can read and

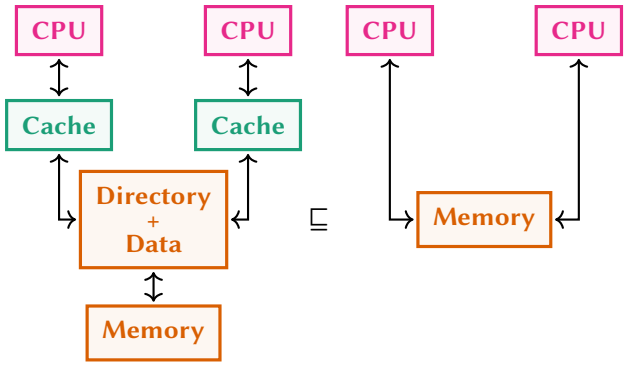


Figure 1. Left: a cache–memory scheme in which the CPU first attempts to retrieve data from the associated cache; if the data is not available, the cache forwards the request to the directory, hich coordinates requests from all caches, keeps track of the data stored in each cache, and accesses main memory as needed. Right: a centralized memory scheme in which the CPU forwards requests directly to memory.

modify the data directly without requesting any information from the parent. The second cache, however, is in the I state and therefore cannot perform either load or store operations. When this second cache receives a load request, it cannot satisfy it immediately because of its current state. To serve the request, it must first obtain at least the S state. For this reason, it sends a message requesting the S state $\mathtt{S}^{c}_{rq}$ to the parent (the superscript *c* indicates that the source of the message is a cache, while the subscript *rq* indicates that the message is a request). The parent maintains a local directory state that tracks the states of all caches. It cannot respond immediately to the request, because doing so would result in a configuration with one cache in M and another in S, which is illegal. This would break coherence: the cache in M could continue modifying the data while the cache in S would only see an outdated value.

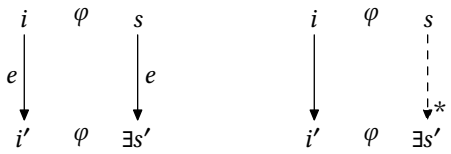
To ensure coherence, the parent must first instruct the cache in M to invalidate its line, transitioning from M to I. This is done by sending a request for invalidation $\mathtt{I}^{p}_{rq}$ (the superscript *p* indicates that the source of the message is a parent). Only once the invalidation is confirmed can the second cache safely move to S. The parent enforces this by waiting for a response confirming the invalidation $\mathtt{I}^{c}_{rs}$ (the subscript *rs* indicates that the message is a response), in which the first cache certifies that it has downgraded its state and provides the latest value it has written. Once this response is received, the parent updates its records, knowing that the first cache is now in I, and then replies to the second cache with a message granting the S state $\mathtt{I}^{p}_{rs}$, allowing it to move into S with the up-to-date value.

2.2 About Behavior and Refinement

In this section, the approach for proving coherence of a protocol is described by introducing standard concepts such as behavior and refinement. Behavior is defined as a trace of external events, such as *load requests*, *load responses*, and *store requests*.

Coherence of a protocol is established by proving a refinement. An implementation I refines a specification S, denoted $I \sqsubseteq S$, when the set of traces of I is included in that of S. The purpose of refinement is to show that the complex implementation, the *MSI* protocol, can be correctly represented by a simpler specification model, such as a central memory system (figure 1). Since central memory is coherent by definition, the implementation inherits this property.

The refinement is established by defining a simulation relation between the implementation and the specification, denoted φ. Two simulation diagrams, shown below, are used. The first describes a lockstep simulation between rules emitting external events, which are included in the trace. It states that if the implementation can step from state i to state i' emitting event e, and i relates to s by some relation φ, then the specification should be able to step from s to some state s' which relates to state i' by φ while emitting the same event e. The second diagram represents a simulation in which each internal step of the implementation is matched by zero or more internal steps of the specification, such that a new specification state s' exists with $\varphi(i', s')$ still holding.



2.3 Proving MSI correctness is a hard problem

To prove this refinement, researchers typically establish several intermediate properties (or invariants) that collectively are used to imply the refinement. An example of a key safety property of the *MSI* protocol is that no two caches should simultaneously hold the same memory address in conflicting states—specifically, it must be prevented that two caches are in state M, or one in M while another is in S, for the same address. This indeed will break the coherency of the memory, because a cache can read an old value while the other cache (in M) already modified it. Proving such a property is not straightforward. Both the parent and the child caches operate based on local state, and state transitions are not instantaneous - they depend on the exchange and processing of coherence messages. Moreover, the parent makes decisions based on its belief about the state of the children, which may be outdated or incomplete due to the asynchronous nature of communication. To prove such properties, we therefore have to formulate *inductive invariants* that encompass the global behavior of the system and implies the property we want

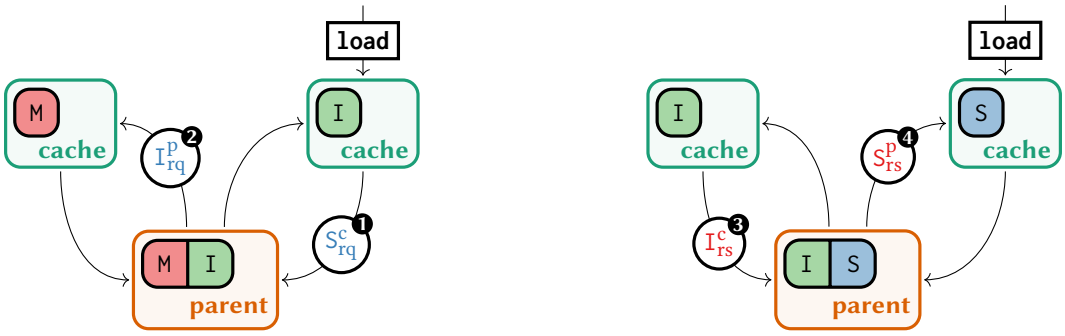


Figure 2. State transitions and message exchange (S^c_{rq}, I^p_{rq}, I^c_{rs}, S^p_{rs}) required to grant the S state to a cache in I when another cache holds M.

to prove. In this section we will go through the example in figure 2 to show the steps needed to prove that bad states are unreachable.

For simplicity, let us consider a system with only two caches and a single parent, handling just one cache line. Since there is only one line, its address does not need to be specified.

In the example in figure 2, one possible invariant to prove that the M and S configuration is illegal could be the following: we cannot simultaneously have, for one cache, a response message S^p_{rs} from the parent that authorizes the child to move into state S , and, for another cache, a parent response message M^p_{rs} that grants permission to transition into state M, both present in the parent-to-cache channels at the same time. Let us define the channel from the parent to the first cache as *pc1*, and the complementary channel from that cache to the parent as *cp1*. For the second cache, we use *pc2* and *cp2*. Then, one invariant can be expressed as $S^p_{rs} \in pc1 \rightarrow M^p_{rs} \notin pc2$, together with all the possible permutations between signals and queues. This ensures that S^p_{rs} and M^p_{rs} are mutually exclusive and cannot be present at the same time in any of the channels.

To establish that this property is an inductive invariant, it must be shown that it holds in the initial state of the system and is preserved under every atomic action the system can perform. By satisfying these two conditions, the property is guaranteed to hold in all reachable configurations. For actions that do not produce S^p_{rs} or M^p_{rs}, this preservation is trivial. But let us assume that there is an M^p_{rs} already in the system, and consider the action in which the parent produces an S^p_{rs}. How do we prove that the invariant is preserved? We must show that this leads to a contradiction, since the parent cannot send an S^p_{rs} if it believes that a cache is in M.

At this point, we need to link the parent's belief with the presence of M^p_{rs} in the system. This requires introducing a new invariant: if there is an M^p_{rs} in the channel from the parent to a cache, then the parent believes that this cache is in M. But this new invariant cannot be proved in isolation. We must add another one: if there is an M^p_{rs} in the system, then there cannot simultaneously be an I^c_{rs} in the complementary channel from the cache to the parent. Otherwise, when the parent consumes the I^c_{rs} and updates its belief about the cache's state to I, the M^p_{rs} would still remain in the system, and we could no longer prove that the parent believes the cache is in M, because in fact it just updated its belief to I.

This shows how the chain of invariants can grow: each invariant may require another one to support it, and so on, until the set of invariants is inductive. This is the typical process to prove one key property of the system. However, in practice we often need to prove several properties, each potentially requiring its own chain of invariants. As a result, the set of invariants can quickly become very large and difficult to manage. A key contribution of this work is showing that it is unnecessary to iteratively search and find these global invariants. Even for proving global properties—such as the impossibility for two caches to be in states M and S simultaneously— we will see that it is sufficient to consider a collection of local invariants involving only a single cache and its parent. This approach simplifies invariant discovery by reducing the analysis to local smaller subsystems and eliminating the need to account for interactions between signals in different cache–parent channels. Counterintuitively, even in the example, there is no need to define invariants enforcing the exclusivity of S^p_{rs} and M^p_{rs} across different cache–parent queues, because the parent's local directory already contains all the necessary information.

3 Verifying *MI* and Formulating a Generalised Invariant

This section will first describe a minimal *MI* and describe how it refines the central memory specification (*Seq*) by formulating an invariant. This invariant is shown to generalise to *MI*, *SI* and *MSI*.

3.1 *MI* description and proofs

We will start by describing a restricted cache coherence protocol called *MI*. In *MI*, the cache can only be in one of two states: I or M. Regardless of whether a load or store request

$$
\begin{array}{rcll}
\mathcal{I}, \mathcal{A}, \mathcal{V} & \triangleq & \mathbb{N} & \text{(identifier, address, value)} \\
\mathcal{N} & \triangleq & \{\, n \in \mathbb{N} \mid n < \#\,\text{caches} \,\} & \\
\mathcal{C}^{MI} & \triangleq & (\mathtt{I} \mid \mathtt{M}) \times \mathcal{A} \times \mathcal{V} & \text{(\textit{MI} cache line)} \\
E & \triangleq & \mathtt{Ld}_{\mathrm{rq}}\ \mathcal{I}\ \mathcal{A} \mid \mathtt{Ld}_{\mathrm{rs}}\ \mathcal{I}\ \mathcal{V} & \\
 & \mid & \mathtt{St}_{\mathrm{rq}}\ \mathcal{I}\ \mathcal{A}\ \mathcal{V} & \text{(external events)} \\
E_{\mathrm{c}}^{MI} & \triangleq & \mathtt{I}^{\mathrm{c}}_{\mathrm{rs}}\ \mathcal{I}\ \mathcal{A}\ \mathcal{V} \mid \mathtt{M}^{\mathrm{c}}_{\mathrm{rq}}\ \mathcal{I}\ \mathcal{A} & \text{(\textit{MI} child events)} \\
E_{\mathrm{p}}^{MI} & \triangleq & \mathtt{I}^{\mathrm{p}}_{\mathrm{rq}}\ \mathcal{I}\ \mathcal{A} \mid \mathtt{M}^{\mathrm{p}}_{\mathrm{rs}}\ \mathcal{I}\ \mathcal{A}\ \mathcal{V} & \text{(\textit{MI} parent events)} \\
\mathcal{S}_{\mathrm{c}}^{MI} & \triangleq & \mathcal{C}^{MI} & \\
 & \times & \mathcal{P}(E_{\mathrm{c}}^{MI}) \times \mathcal{P}(E_{\mathrm{p}}^{MI}) & \text{(internal bags)} \\
 & \times & \vec{E} \times \vec{E} & \text{(external rq/rs queues)} \\
\mathcal{B}^{MI} & \triangleq & \mathcal{N} \to (\mathtt{I} \mid \mathtt{M}) & \text{(directory/belief)} \\
\mathcal{S}_{\mathrm{p}}^{MI} & \triangleq & \mathcal{C}^{MI} & \\
 & \times & \mathcal{N} \to \mathcal{P}(E_{\mathrm{c}}^{MI}) & \\
 & \times & \mathcal{N} \to \mathcal{P}(E_{\mathrm{p}}^{MI}) & \text{(internal bags)} \\
 & \times & \mathcal{B}^{MI} & \text{(directory/belief)} \\
 & \times & \mathcal{A} \to \mathcal{V} & \text{(main memory)} \\
\mathcal{S}^{MI} & \triangleq & (\mathcal{N} \to \mathcal{S}_{\mathrm{c}}^{MI}) \times \mathcal{S}_{\mathrm{p}}^{MI} & \\
\mathcal{S}^{Seq} & \triangleq & \mathcal{A} \to \mathcal{V} & \text{(main memory)} \\
 & \times & \vec{E} \times \vec{E} & \text{(external queues)}
\end{array}
$$

Figure 3. State definition of the state of *MI*.

is issued, the cache must be in the M (Modified) state in order to either store or load a value. Unlike *MSI*, the S state is not present. To preserve coherence, no two caches may be in the M state simultaneously. The concepts described in this section as well as the structure of *MI* will also be valid and similar to those of the other protocols described in later sections, such as *MSI* itself. We first describe how we model an *MI* system. We then describe the invariant that is sufficient to prove that *MI* is coherent and provide a taxonomy for this invariant that will be useful for understanding other systems.

Figure 3 shows the entire state of the *MI* system, which we denote as $\mathcal{S}^{MI}$. It comprises a set of $\mathcal{N}$ cache states ($\mathcal{S}_{\mathrm{c}}^{MI}$) for each child cache in the system, as well as the state of the parent cache ($\mathcal{S}_{\mathrm{p}}^{MI}$). We generalise over the number of child caches because this already avoids some duplication when defining the rules of the system. Next, the parent and child cache states are very similar. At the core of the cache there is a single cache line $\mathcal{C}^{MI}$ with the state of the line, the address, and the value. In addition to that, all caches also have sets of internal events that they can use to communicate, one for messages from the parent ($\mathcal{P}(E_{\mathrm{p}}^{MI})$), and one for messages from the child ($\mathcal{P}(E_{\mathrm{c}}^{MI})$). To model the fact that messages can overtake each other, we store messages in a set, without order. The *MI* system has four types of messages, two responses and two requests, all related to the I and M states. The response messages are:

- $\mathtt{M}^{\mathrm{p}}_{\mathrm{rs}}$, which instructs the child to upgrade from I to M for a given address and value.
- $\mathtt{I}^{\mathrm{c}}_{\mathrm{rs}}$, which informs the parent that the child has downgraded from M to I. Here, μ represents exactly this downgrade: in the *MI* system there is only one such downgrade, while in the *MSI* system there will be two, one for the downgrade from M to I (namely $\mathtt{I}^{\mathrm{c}}_{\mathrm{rs}}$) and one for the downgrade from S to I (namely $\mathtt{I}^{\mathrm{c}}_{\mathrm{rs}}$).

The request messages are:

- $\mathtt{M}^{\mathrm{c}}_{\mathrm{rq}}$, with which the cache asks the parent to allow a transition to the M state.
- $\mathtt{I}^{\mathrm{p}}_{\mathrm{rq}}$, which indicates that the parent requests the child to downgrade from M to I. Again, here μ represents this particular request. In the *MSI* system there will also be a request to downgrade from S to I, represented by the $\mathtt{I}^{\mathrm{p}}_{\mathrm{rq}}$ message.

Figure 3 also shows the state of our sequential memory $\mathcal{S}^{Seq}$, which only consists of the external queues and a memory. The transition system of *MI* acting on this state is separated into transitions acting on the parent state and transitions acting on the child state. Furthermore, there are external transitions acting on each child which will add load request $\mathtt{Ld}_{\mathrm{rq}}$ and load response $\mathtt{Ld}_{\mathrm{rs}}$ into the external request and response queues. The parent has access to the main memory, and there are transitions that will flush the state of the parent to main memory when a different address is requested by one of the children.

Without going into detail about the specific transitions associated with *MI* we will sketch a proof of refinement by simulation between *MI* and *Seq*. To prove refinement, we have to find a simulation relation φ between states of $\mathcal{S}^{MI}$ and states of $\mathcal{S}^{Seq}$. In addition to the φ, we generally also need an invariant which can state, for example, that certain states are unreachable. In the case of the *MI* system we need to show that no two caches can be in the M state for the same address. Because the cache we are modelling is *inclusive*, this condition can be relaxed to stating that no two caches can be in the M state. Here we will just consider a few projection functions to extract parts of the overall state.

$$
\begin{array}{c}
\cdot.\mathrm{addr} : \mathcal{S}_{*}^{MI} \to \mathcal{A},\ \cdot.\mathrm{val} : \mathcal{S}_{*}^{MI} \to \mathcal{V},\ \cdot.\mathrm{st} : \mathcal{S}_{*}^{MI} \to (\mathtt{I} \mid \mathtt{M}) \\
\cdot.\mathrm{mem} : \mathcal{S}_{\mathrm{p}}^{MI} \to \mathcal{A} \to \mathcal{V}, \cdot.\mathrm{dir}_{\_} : \mathcal{S}_{\mathrm{p}}^{MI} \to \mathcal{N} \to (\mathtt{I} \mid \mathtt{M}) \\
\cdot.\mathrm{cm} : \mathcal{S}_{\mathrm{c}}^{MI} \to \mathcal{P}(E_{\mathrm{c}}^{MI}),\ \cdot.\mathrm{cm}_{\_} : \mathcal{S}_{\mathrm{p}}^{MI} \to \mathcal{N} \to \mathcal{P}(E_{\mathrm{c}}^{MI}) \\
\cdot.\mathrm{pm} : \mathcal{S}_{\mathrm{c}}^{MI} \to \mathcal{P}(E_{\mathrm{p}}^{MI}),\ \cdot.\mathrm{pm}_{\_} : \mathcal{S}_{\mathrm{p}}^{MI} \to \mathcal{N} \to \mathcal{P}(E_{\mathrm{p}}^{MI})
\end{array}
$$

We note that the state transition system of *MI* ensures that $p.\mathrm{pm}_i$ always equals $c_i.\mathrm{pm}$, and $p.\mathrm{cm}_i$ always equals $c_i.\mathrm{cm}$, since they represent the same channels connecting parent and child in both directions. In the invariant description, we adopt the parent notation $p.\mathrm{pm}_i$ and $p.\mathrm{cm}_i$. We can formulate the complete invariant $\psi(c, p)$, assuming that $(c, p) \in \mathcal{S}^{MI}$, on the *MI* system which implies the desired property using the following eight conjunctive clauses.

1. *Signal to state*
   - $\mathtt{M}^{\mathtt{p}}_{\mathtt{rs}} \in p.\mathrm{pm}_i \;\rightarrow\; p.\mathrm{dir}_i = \mathtt{M} \wedge c_i.\mathrm{st} = \mathtt{I}$.
   - $\mathtt{I}^{\mathtt{c}}_{\mathtt{rs}} \in p.\mathrm{cm}_i \;\rightarrow\; p.\mathrm{dir}_i = \mathtt{M} \wedge c_i.\mathrm{st} = \mathtt{I}$.
2. *Signal uniqueness*
   - There can be no two $\mathtt{M}^{\mathtt{p}}_{\mathtt{rs}}$.
   - There can be no two $\mathtt{I}^{\mathtt{c}}_{\mathtt{rs}}$.
3. *Overapproximation*
   - If a cache is in M the parent must believe it is in M:
   $$c_i.\mathrm{st} = \mathtt{M} \;\rightarrow\; p.\mathrm{dir}_i = \mathtt{M}$$
   - If the parent believes the cache is in I, the cache should be in I:
   $$p.\mathrm{dir}_i = \mathtt{I} \;\rightarrow\; c_i.\mathrm{st} = \mathtt{I}$$
4. *Relation between signals*
   - $\mathtt{I}^{\mathtt{c}}_{\mathtt{rs}} \in p.\mathrm{cm}_i \;\rightarrow\; \mathtt{M}^{\mathtt{p}}_{\mathtt{rs}} \notin p.\mathrm{pm}_i$.
5. *Conflicting Configurations*
   - If the parent believes a cache is in M, no other cache can be in M:
   $$\forall j,\; p.\mathrm{dir}_i = \mathtt{M} \wedge j \neq i \;\rightarrow\; p.\mathrm{dir}_j \neq \mathtt{M}.$$

All the conjunction states defined above are local, as they concern only the parent subsystem p and a generic cache c, and we prove that these are all the conjunctives necessary for ψ to be inductive. No invariants relate the states of different caches or signals in different parent-cache channels. In other words, contrary to what might be expected, no global invariants such that

$$\mathtt{M}^{\mathtt{p}}_{\mathtt{rs}} \in p.\mathrm{pm}_i \;\rightarrow\; \mathtt{M}^{\mathtt{p}}_{\mathtt{rs}} \notin p.\mathrm{pm}_j$$

where i and j are two distinct caches, are required.

Conversely, if we would like to prove global properties of the system, for example, that two caches cannot simultaneously be in the M state, this can be achieved by relying solely on the local invariants defined above. Thus:

Lemma 3.1. $\psi(c, p) \;\rightarrow\; \forall j.\, c_i.\mathrm{st} = \mathtt{M} \wedge j \neq i \;\rightarrow\; c_j.\mathrm{st} \neq \mathtt{M}$.

Proof sketch. The conflicting configuration (conjunction 5.) together with overapproximation (conjunction 3.) imply that there is only one cache in the M state. Intuitively, if a cache is in M, then the parent must believe that this cache is in M. Moreover, the parent cannot simultaneously believe that two distinct caches are in the M state. □

Theorem 3.2 (*MI* refines *Seq*)**.**

$$MI \sqsubseteq Seq$$

Proof sketch. We formulate a φ which relates the main memory in *MI* to the main memory in *Seq*, except for the value associated with an address that is in a cache in an M state, is in an $\mathtt{I}^{\mathtt{c}}_{\mathtt{rs}}$ message or is in an $\mathtt{M}^{\mathtt{p}}_{\mathtt{rs}}$ message. Together, φ and our invariant ψ form our simulation relation which implies the refinement. □

3.2 Invariant taxonomy description

Based on the specific example already illustrated for *MI*, we now present the general taxonomy of invariants.

To introduce the taxonomy, certain assumptions about the protocol to be modeled are required. A base state is defined (for example, the I state in *MSI*, *MI*, and *SI*). The set of all possible states is denoted by Σ (e.g., in *MSI*: I, S, M). A set of conflicting states, Σ_{conf}, is then defined as tuples of states that cannot occur simultaneously. For instance, in *MSI* it is not possible to have one cache in M and another cache in S, hence $(\mathtt{S}, \mathtt{M}) \in \Sigma^{MSI}_{conf}$. Similarly, in *MI* the configuration $(\mathtt{M}, \mathtt{M}) \in \Sigma^{MI}_{conf}$ is forbidden, since having two or more caches in M is not allowed.

An upgrade relation ($\prec$) is defined over all states that are not part of a conflicting configuration. For example, in *MI* it holds that $\mathtt{I} \prec \mathtt{S}$. In *MSI*, both $\mathtt{I} \prec \mathtt{S}$ and $\mathtt{I} \prec \mathtt{M}$ hold, but $\mathtt{S} \prec \mathtt{M}$ cannot be stated because $(\mathtt{S}, \mathtt{M}) \in \Sigma^{MSI}_{conf}$. In all *MI*, *SI*, and *MSI* protocols, the order relation is defined between at most two states, such as $A \prec B$. There is no transitive ordering of the form $A \prec B \prec C$. The state B is referred to as the upgrade state, while A is referred to as the downgrade state. The upgrade state is M in *MI*, while in *MSI* the upgrade states are both S and M.

Two classes of signals are introduced:

- $\mathtt{U}^{\mathtt{p}}_{\mathtt{rs}}$, the upgrade response from the parent to the cache (e.g., $\mathtt{M}^{\mathtt{p}}_{\mathtt{rs}}$ in *MI*),
- $\mathtt{D}^{\mathtt{c}}_{\mathtt{rs}}$, the downgrade response from the cache to the parent (e.g., $\mathtt{I}^{\mathtt{c}}_{\mathtt{rs}}$ in *MI*).

Counterintuitively, in the invariant taxonomy, focus is placed solely on $\mathtt{U}^{\mathtt{p}}_{\mathtt{rs}}$ and $\mathtt{D}^{\mathtt{c}}_{\mathtt{rs}}$, and we show that requests can be ignored. We define a directory-based protocol, in which parent nodes maintain local views of the state of each cache. Naturally, these beliefs ($p.\mathrm{dir}_i$) may be outdated due to the asynchronous nature of cache coherence protocols. We now present an intuitive taxonomy of these invariants. Since all conjunctions hold uniformly for every cache, we omit the explicit quantification over caches. The conjunctions can be naturally grouped into 4 distinct classes:

1. *Signal to State*
   - If there is an in-flight upgrade response message from the parent to the cache, the parent believes that the cache is in the upgrade state, while the cache itself remains in the downgrade state.
   $$\mathtt{U}^{\mathtt{p}}_{\mathtt{rs}} \in p.\mathrm{pm}_i \;\rightarrow\; p.\mathrm{dir}_i = U \wedge c_i.\mathrm{st} = D$$
   - If there is an in-flight downgrade response message from the cache to the parent, the cache is in the downgrade state, while the parent believes that the cache is in the upgrade state.
   $$\mathtt{D}^{\mathtt{c}}_{\mathtt{rs}} \in p.\mathrm{cm}_i \;\rightarrow\; p.\mathrm{dir}_i = U \wedge c_i.\mathrm{st} = D$$
2. *Signal Uniqueness*

- In the channel from parent to cache, all upgrade responses for a specific state are unique:

$$\mathtt{U}^{\mathtt{p}}_{\mathtt{rs}} \in p.\mathrm{pm}_i \ \wedge\ \mathtt{U}'^{\mathtt{p}}_{\mathtt{rs}} \in p.\mathrm{pm}_i \ \rightarrow\ U = U'$$

- In the channel from cache to parent, all downgrade responses for a specific state are unique:

$$\mathtt{D}^{\mathtt{c}}_{\mathtt{rs}} \in p.\mathrm{cm}_i \ \wedge\ \mathtt{D}'^{\mathtt{c}}_{\mathtt{rs}} \in p.\mathrm{cm}_i \ \rightarrow\ D = D'$$

3. *Overapproximation*
   - The parent's belief always over-approximates the state of the cache:

$$c_i.\mathrm{st} \preceq p.\mathrm{dir}_i$$

4. *Relation Between Signals*
   - If there is a downgrade response from the cache to the parent, there cannot be an upgrade response from the parent to the child:

$$\mathtt{D}^{\mathtt{c}}_{\mathtt{rs}} \in p.\mathrm{cm}_i \ \rightarrow\ \mathtt{U}^{\mathtt{p}}_{\mathtt{rs}} \notin p.\mathrm{pm}_i$$

5. *Conflicting Configurations*
   - For all conflicting configurations, the parent cannot believe that such a configuration exists among the caches:

$$\forall\, i\ j.\ \forall (x, y) \in \Sigma_{conf}.\quad p.\mathrm{dir}_i = x \ \rightarrow\ p.\mathrm{dir}_j \neq y$$

Invariant locality. A key characteristic of our invariant taxonomy is its simplicity, not only in terms of having few conjunctive clauses, but also because we do not need to reason about signals or states across different caches. Consequently, each conjunctive clause concerns only the relationship between a single cache and its parent; there are no clauses that, for example, assert that if a signal is present in a channel between a parent and one cache, another signal cannot be present in a channel between the same parent and a different cache, as in the example described in Section 2. In other words, all our conjunctive clauses are local to the cache–parent subsystem. We then extend the result by asserting that the same property holds for all other cache–parent pairs in the system. This approach explicitly exploits the modularity of the system, allowing us to reason about one cache–parent subsystem independently of others. An important benefit of this modularity is that it facilitates scaling the system to a parametric number of caches, as extending the invariants to additional cache–parent pairs is straightforward.

SI ***invariant.*** The taxonomy is applied to describe the invariants for the *SI* state. The invariant is composed of seven conjunctive clauses, grouped into four categories. In the *SI* protocol, there are no conflicting configurations; therefore, the last category of the taxonomy is not applicable. The upgrade relation is defined as $\mathtt{I} \prec \mathtt{S}$, resulting in a single upgrade request and response signal associated with the S state.

3.3 From *SI* and *MI* Invariants to *MSI* Invariants

A notable observation is that we get inductive invariants of the *MSI* model with precisely the conjunction of the *SI* and *MI* invariants, and consists of only 14 conjunctive clauses, significantly fewer and more structured than reported in the state of the art [Tan et al. 2025; Vijayaraghavan 2016]. Counterintuitively, it is not necessary to consider potential interference from $\mathtt{S}^{\mathtt{p}}_{\mathtt{rs}}$ and $\mathtt{M}^{\mathtt{p}}_{\mathtt{rs}}$, overapproximation between S and M. The combination of the *SI* and *MI* invariants alone is sufficient. All conjunctions of *MSI* correspond precisely to the combinations of the conjunctions of *SI* and *MI*, except in the case of over-approximation, where duplicate clauses concerning the I state can be eliminated, and in cases involving conflicting configurations. We chose not to merge $\mathtt{I}^{\mathtt{c}}_{\mathtt{rs}}$ and $\mathtt{I}^{\mathtt{c}}_{\mathtt{rs}}$ because one does not have to carry data as it can simply be discarded. We also did not merge $\mathtt{I}^{\mathtt{c}}_{\mathtt{rq}}$ and $\mathtt{I}^{\mathtt{c}}_{\mathtt{rq}}$ mainly because real-world protocols might also distinguish between such requests [Tan et al. 2025]. Consequently, the conjunctions related to signal-to-state and signal uniqueness do not introduce duplicates, unlike in the case of overapproximation.

This example illustrates how compositional reasoning allows one to avoid re-proving all statements for a larger system such as *MSI*. In particular, the conflicting configuration clause *M* and *M* does not need to be re-established for *MSI*, as it has already been proven in the *MI* component. The invariant for *MSI* is therefore:

1. *Signal to state*
   - Signal to state of *SI*
   - Signal to state of *MI*
2. *Signal uniqueness*
   - Signal uniqueness of *SI*
   - Signal uniqueness of *MI*
3. *Overapproximation*
   - $c_i.\mathrm{st} = \mathtt{M} \ \rightarrow\ p.\mathrm{dir}_i = \mathtt{M}$
   - $c_i.\mathrm{st} = \mathtt{S} \ \rightarrow\ p.\mathrm{dir}_i = \mathtt{S}$
   - $p.\mathrm{dir}_i = \mathtt{I} \ \rightarrow\ c_i.\mathrm{st} = \mathtt{I}$
4. *Relation between signals*
   - Relation between signals of *SI*
   - Relation between signals of *MI*
5. *Conflicting Configurations*
   - $\forall j,\ p.\mathrm{dir}_i = \mathtt{M} \wedge j \neq i \ \rightarrow\ p.\mathrm{dir}_j \neq \mathtt{S}$.

4 Verifying *MSI* by Composition

Instead of attempting a direct proof of correctness, our approach focuses on decomposing the problem. We first prove that the *MSI* protocol can be expressed as the composition of the *MI* and *SI* protocols, and then show that this composition behaves as a single sequential memory by using the fact that each of the protocols *MI* and *SI* behaves like a sequential memory (theorem 3.2). The main challenge, therefore, lies not in proving *MI* or *SI* individually, but in establishing that *MSI* indeed behaves as their composition. The key insight is that a correct *MSI* protocol naturally enforces consistency by

forbidding conflicting cache states, such as one cache being in the Modified state while another is in the Shared state. This exclusivity implies that, at any point in time, the system effectively operates either in *MI* mode or in *SI* mode, but never both simultaneously. If the behavior were to mix these modes improperly, the protocol would violate coherence. Thus, our proof approach leverages this symmetry to formalize *MSI* as the disjoint union of *MI* and *SI* behaviors, making the overall correctness argument more modular, simple and comprehensible.

It is important to emphasize that the intermediate system *MI+SI* is not intended as a hardware design, but as a verification artifact introduced solely for the purposes of proof decomposition. The following subsections first formalize the *MSI* system, then describe in detail the composition *MI+SI*. The proof structure is subsequently presented, with particular attention to the intermediate step from *MSI* to *MI+SI*, together with an explanation of how introducing a degree of nondeterminism facilitates the argument.

4.1 Formalization of *MSI*

The *MSI* protocol, for which correctness is established, adopts the same design choices as the *MI* protocol described in section 3. In particular, the number of caches is parametric, and the communication channels between caches and the parent are modeled as sets rather than queues, thereby mimicking out-of-order communication. Another feature is the assumption of an inclusive cache hierarchy, which implies that the parent stores all cache lines held in the children. For example, if two caches each contain two lines, then the parent holds a total of four lines.

The state definition of *MSI* is very similar to the one shown in figure 3 for the *MI* system. The differences, illustrated in figure 4, are that a cache line $\mathcal{C}^{MSI}$ may now also be in the S state, and correspondingly the parent's belief $\mathcal{B}^{MSI}$ may include S as well. New signals are introduced: a cache may request to move to the S state, and the parent may respond with an approval for this transition, namely $\mathtt{S}^{c}_{rq}$ and $\mathtt{S}^{p}_{rs}$, respectively. As in the standard *MSI* protocol, two further signals handle downgrades from S to I. Specifically, when a cache in S is downgraded, it sends $\mathtt{I}^{c}_{rs}$, which does not require a value since S does not involve modifications. Symmetrically, if the parent's belief is that the cache is in S, it sends $\mathtt{I}^{p}_{rq}$.

Two example transition rules are presented—one for the cache and one for the parent—to illustrate the semantics using the notation introduced in section 3.

In figure 5, the rule corresponding to a cache in the I state handling a $\mathtt{M}^{p}_{rs}$ is shown. Above the horizontal bar, the precondition of the rule is specified: the channel between the parent and the cache contains a $\mathtt{M}^{p}_{rs}$, and the cache is in the I state. Below the bar, the postcondition describes how the cache state is updated. In particular, the cache transitions

$$
\begin{array}{lll}
\mathcal{C}^{MSI} & \triangleq & (\mathtt{I} \mid \mathtt{S} \mid \mathtt{M}) \times \mathcal{A} \times \mathcal{V} \qquad (\textit{MSI}\text{ cache line}) \\
E_c^{MSI} & \triangleq & \mathtt{I}^{c}_{rs}\ \mathcal{I}\ \mathcal{A} \mid \mathtt{I}^{c}_{rs}\ \mathcal{I}\ \mathcal{A}\ \mathcal{V} \\
 & \mid & \mathtt{S}^{c}_{rq}\ \mathcal{I}\ \mathcal{A} \mid \mathtt{M}^{c}_{rq}\ \mathcal{I}\ \mathcal{A} \qquad (\textit{MSI}\text{ child events}) \\
E_p^{MSI} & \triangleq & \mathtt{I}^{p}_{rq}\ \mathcal{I}\ \mathcal{A} \mid \mathtt{I}^{p}_{rq}\ \mathcal{I}\ \mathcal{A} \\
 & \mid & \mathtt{S}^{p}_{rs}\ \mathcal{I}\ \mathcal{A}\ \mathcal{V} \mid \mathtt{M}^{p}_{rs}\ \mathcal{I}\ \mathcal{A}\ \mathcal{V} \qquad (\textit{MSI}\text{ parent events}) \\
\mathcal{B}^{MSI} & \triangleq & \mathcal{N} \to (\mathtt{I} \mid \mathtt{S} \mid \mathtt{M}) \qquad (\text{directory/belief})
\end{array}
$$

Figure 4. State definition of the state of *MSI*.

$$
\textsc{handling\_rsm}\quad
\frac{\mathtt{M}^{p}_{rs}(a,t,v) \in c.\mathrm{pm} \qquad c.\mathrm{st} = \mathtt{I}}{
c \longrightarrow \left\{ c \;/\!/ \begin{array}{l} c.\mathrm{st} := \mathtt{M} \\ p.\mathrm{addr} := a \\ p.\mathrm{val} := v \\ c.\mathrm{pm} := c.\mathrm{pm} \setminus \mathtt{M}^{p}_{rs}(a,t,v) \end{array} \right\}
}
$$

Figure 5. *MSI* internal cache state transition relation.

$$
\textsc{rqm\_data\_available}\quad
\frac{\mathtt{M}^{c}_{rq}(a,t) \in p.\mathrm{cm}_i \qquad p.\mathrm{st} = \mathtt{I} \qquad \forall i,\ p.\mathrm{dir}_i = \mathtt{I}}{
p \longrightarrow \left\{ p \;/\!/ \begin{array}{l} p.\mathrm{st} := \mathtt{M} \\ p.\mathrm{addr} := a \\ p.\mathrm{val} := p.\mathrm{mem}(a) \\ p.\mathrm{cm}_i := p.\mathrm{cm}_i \setminus \mathtt{M}^{c}_{rq}(a,t) \\ p.\mathrm{pm}_i := p.\mathrm{pm}_i \cup \mathtt{M}^{p}_{rs}(a,t) \end{array} \right\}
}
$$

Figure 6. *MSI* internal parent state transition relation.

to the M state, stores the new value v at the new address a, and removes the $\mathtt{M}^{p}_{rs}$ message from the channel.

Similarly, figure 6 illustrates a parent rule. In this case, if the parent receives a $\mathtt{I}^{c}_{rq}$ from the same cache i, the parent is in state I, and it believes that all other caches are also in I, then the parent can update its state to M, fetch the value associated with the write address from memory, generate a $\mathtt{M}^{c}_{rq}$ message, and produce the $\mathtt{M}^{p}_{rs}$ signal.

4.2 Combination of *MI* and *SI*

The central idea of the composition proof is to demonstrate that $MSI \sqsubseteq MI{+}SI \sqsubseteq Seq$. However, in order to capture the full behavior of the *MSI* protocol in *MI+SI*, we must allow the system to switch between the two worlds. These transitions must be carefully designed to preserve the necessary information so that, when returning to a previously active world, the protocol can resume execution correctly.

Technically, we define the *MI+SI* state as an union type: the system is either in an *MI* state or in an *SI* state. The key insight is that a switch between the two worlds is only allowed when all caches and the parent are in the invalid state I. In this configuration, we do not need to retain any cache data, as the parent being in I guarantees that the most

recent value has already been written back to main memory. When a world switch occurs, we reinitialize the caches and the parent with default values, preserving the I states. These default values are used for both the address and the data fields in the cache and parent registers, as these fields are irrelevant in the I state. Since the meaning of I is identical in both *MI* and *SI* models, this transformation is safe.

Importantly, we do not require the communication channels between caches and parent to be empty at the switching point, as enforcing such a condition would be overly restrictive. Instead, we snapshot the set of in-flight messages and store them. For instance, during a switch from *SI* to *MI*, we save all messages from the *SI* channels and restore the most recent set of *MI* messages previously saved before switching to *SI* (section 4.4 presents an example of world switching). This mechanism ensures that each world maintains its own independent set of messages, which are only modified while the system is in that specific world. In summary, the combination of disjoint state execution and safe world switching enables us to model the *MSI* protocol by composing *MI* and *SI* behaviors in a coherent and well-structured way.

4.3 Proof structure

To prove that *MSI* works equivalently to the sequential memory model, we will prove a refinement between the two systems, by exhibiting a simulation relation. Specifically, we will prove that the *MSI* protocol refines ($\sqsubseteq$) a sequential memory. Formally, the refinement is established via the following two theorems:

Theorem 4.1. $MSI \sqsubseteq MI{+}SI$

Proof sketch. Proof sketched in section 4.4. □

Theorem 4.2. $SI \sqsubseteq Seq \wedge MI \sqsubseteq Seq \;\rightarrow\; MI{+}SI \sqsubseteq Seq$

Proof sketch. Recall that the state of $MI + SI$ is a union type. Performing a case analysis on this union generates two subgoals:

$$MI \sqsubseteq Seq \quad \text{and} \quad SI \sqsubseteq Seq.$$

Both subgoals can be discharged directly using the hypothesis. □

As a consequence, by transitivity of $\sqsubseteq$, the desired refinement result follows:

$$MSI \sqsubseteq Seq \tag{1}$$

4.4 Proofs under Nondeterminism

The most challenging part of the proof concerns theorem 4.1. To facilitate the refinement argument, the definitions of *MI* and *SI* are slightly relaxed so as to admit nondeterministic behavior. This modification does not introduce any difficulties in proving $MI \sqsubseteq Seq$ and $SI \sqsubseteq Seq$. Concretely, for a given system state and a set of messages in flight, the system is allowed to choose among multiple possible transitions, rather than being constrained to a uniquely determined step. In cache coherence protocols, state changes are usually enabled only when specific conditions on the current state or on pending messages are satisfied. By relaxing some of these enabling conditions, nondeterminism is introduced into the protocol definition, without affecting the correctness argument. An example of this is shown below, where an $\mathtt{M}^{c}_{rq}$ signal may be sent even without the condition of having a store request in the external queue. In this case, the transition can occur spontaneously when a cache is in the I state.

$$\textsc{produce\_rqm}\quad \frac{c.\mathrm{st} = \mathtt{I}}{c \longrightarrow \left\{\, c \;//\; c.\mathrm{cm} := c.\mathrm{cm} \cup \mathtt{M}^{c}_{rq}(a,t,v) \quad \right\}}$$

Intuitively, nondeterminism makes the system more general, as it admits a broader set of behaviors. This generality is useful in refinement proofs: to establish refinement, it suffices to show that the behavior of *MSI* is a subset of the (more general) behavior of the nondeterministic *MI*+*SI* system.

To illustrate this idea, we analyze a representative case from the refinement proof. The proof proceeds by defining a simulation relation between the implementation and the specification, which captures the correspondence between their states. Then, it must be shown that for every step in the implementation (in our case, *MSI*), there exists a step or sequence of steps in the specification (*MI*+*SI*) such that the relation is preserved.

Let us consider an action in the *MSI* system where every cache and its parent are in the I state, and the parent receives a $\mathtt{M}^{c}_{rq}$ from a cache. In this scenario, the parent can fetch the value from memory and send it back to the requesting cache, thereby generating a $\mathtt{M}^{p}_{rs}$ message. The parent also updates its belief about the cache state from I to M and raises the $\mathtt{M}^{c}_{rq}$ message. Figure 7 illustrates the state of the system at the end of this transition step. In the *MI*+*SI* system, the configuration may instead correspond either to the *MI* or to the *SI* world. To establish refinement, it is necessary to construct a sequence of actions starting from both worlds such that the resulting state coincides with that of *MSI*.

The most difficult case occurs when the system is initially in the *SI* world. In this case, a world switch is required, since in the end a $\mathtt{M}^{p}_{rs}$ must be sent, and such a signal does not exist in *SI*. This switch is safe because all caches are in I, making the transition consistent. After the switch, the steps can be performed in the *MI* system to reproduce the behavior of *MSI*, namely handling the $\mathtt{M}^{c}_{rq}$ and producing a $\mathtt{M}^{p}_{rs}$. Concretely, this requires generating a $\mathtt{M}^{c}_{rq}$ in *MI*, fetching the value from memory, and sending back a $\mathtt{M}^{p}_{rs}$ with the value to the cache. At this point, the state of the system and the messages in flight are the same in *MSI* and in *MI*, proving that there exists a sequence of actions in *MI*+*SI* such that the two systems remain related. This process is visualized in

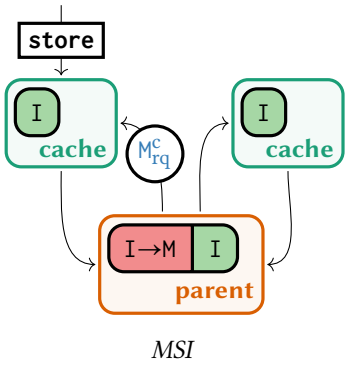


Figure 7. *MSI* state transition for handling M^c_{rq}, resulting in M^c_{rs}

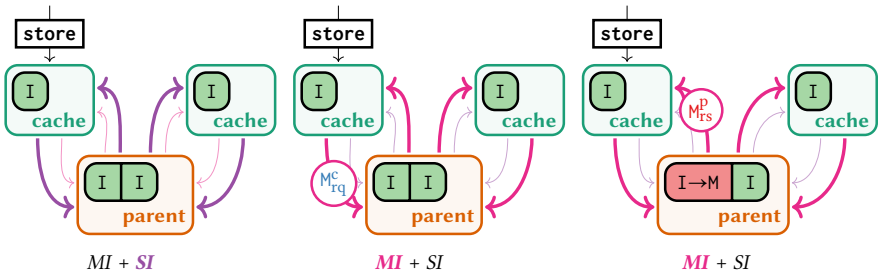


Figure 8. *MI+SI* state transition for handling M^c_{rq}, resulting in M^p_{rs}.

figure 8, which illustrates the three transition steps of the *MI + MSI* system described above.

The first question is: why must a M^c_{rq} be generated in the MI world, rather than already being present as in MSI? Consider the case where, in the *MSI* system, a cache sends a M^c_{rq} while another cache is in the *S* state. In this configuration, *MSI* is related to the *SI* world, since in *MI* the S state is not possible. The generation of the M^c_{rq} is valid in *MSI*, but it cannot happen in *SI*, where such a signal does not exist. This implies that, after the parent invalidates the cache in S and the system can finally be related to the *MI* world, the M^c_{rq} is no longer present in the channel. Recall that, when switching worlds, only the signals generated in that world are put back into the channel; since this M^c_{rq} was generated while *MSI* was aligned with *SI*, it cannot appear in *MI*.

The second question is: why must the M^c_{rq} be generated nondeterministically? The external queues of read and write requests are assumed to be the same in both *MSI* and *MI+SI*. One might wonder why the M^c_{rq} cannot be derived directly from the first request in the queue. This would be possible, but it would require introducing an additional invariant relating the external queue to the internal signals, such as: if there is a M^c_{rq}, then the first element of the external queue must be a write request. To avoid introducing such invariants, which would lead to a long and complex chain of proofs as in previous work, the step of generating a M^c_{rq} is instead allowed nondeterministically, without relying on the queue.

This example shows how nondeterminism can play a crucial role in refinement proofs. By permitting additional behaviors in the *MI+SI* system, it becomes possible to avoid introducing further invariants about *MSI*, thereby reducing both the number and the complexity of the proofs while still establishing refinement.

Nondeterminism for general behavior. Some characteristics of our *MSI* implementation, such as the parametric number of caches and the use of out-of-order communication channels, are sufficiently general to be reused across different settings. For example, to verify a specialized variant of the *MSI* protocol—denoted *MSI′*, with a fixed number of caches and ordered communication channels—it is sufficient to establish *MSI′* ⊑ *MSI*. By transitivity of refinement, MSI′ can then be regarded as a refinement of the sequential memory model Seq.

The compositional structure of the proofs further enhances scalability. If only a subprotocol of the system is modified, rather than the entire protocol, it is enough to reason about the refinement of the modified subprotocol in isolation. For instance, if a modified version of the *MI* protocol (*MI′*) is introduced, it suffices to prove that *MI′* ⊑ *MI*. All higher-level results, from *MSI* up to *Seq*, then remain valid without requiring additional proofs.

5 Extension to multiple cache lines

Until now we have only dealt with a single cache line and a single address, which simplifies the system. Obviously, real caches operate on more than one address, so we would like to be able to build a cache with multiple addresses, but we would like to reuse the basic protocols we have already defined.

We call the multiple cache line implementation of the *MI* protocol *MImult*. The goal is then to find a way to prove that this protocol is coherent, *using the building blocks that we already have*. To do this, we want to find a way to compose multiple *MI*, each modeling a single cache line, thereby simulating multiple cache lines.

5.1 Attempt #1: *MImult* ⊑ *MI* ⊑ *Seq*

The first idea one might try is to prove *MImult* ⊑ *MI*, which would then imply *MImult* ⊑ *Seq*. The main problem with this proof is that proving *MImult* ⊑ *MI* is not trivial. These two systems cannot proceed in lock-step, because *MImult* can keep multiple cache lines in the M state, whereas *MI* would have to downgrade every time. This would require complex formulations for when two states are related, and it seems

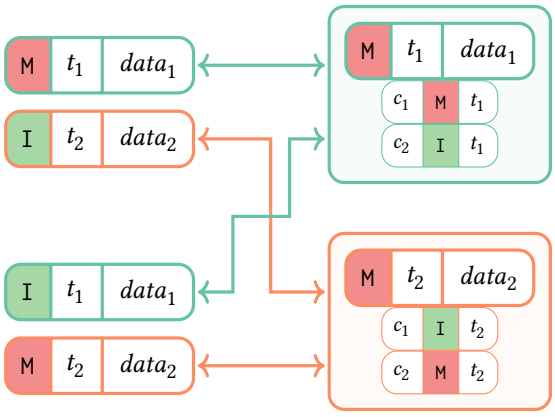


Figure 9. *MIdup* with two caches (C_1, C_2) and two cache lines. The parent stores both cache lines in state M, since one of the two caches is in M for each cache line. The directory records the state of both children, ensuring that if one child is in M for a cache line, the other can only be in I.

like it might end up being as complex as the original *MI* ⊑ *Seq* proof.

5.2 Attempt #2: *MImult* ⊑ *MIdup* ⊑ *Seq*

Instead, we can create *MIdup* (figure 9), which is a systems with duplicated *MI* state machines for each cache line, and then prove that the more optimised *MImult*, where there is a single state machine that orchestrates the multiple cache lines, refines it. Each child cache made up of many smaller single-line caches, and at the boundary between the caches and the core there is an address translation that will send requests to the correct core depending on the cache line offset. Each single-line cache then connects to a single-line parent cache with a single-line directory (only the connections for one single-line cache system are shown).

The main problem is that *MIdup* ⊑ *Seq* is still not actually easily provable without having to reason about the *MI* protocol again, because one cannot simply reuse the *MI* ⊑ *Seq* proof.

5.3 Attempt #3: *MImult* ⊑ *MIdup* ⊑ *SeqDup* ⊑ *Seq*

The final strategy to complete the proof is to add one more system into the mix, *SeqDup*. This system duplicates a *whole memory* that will be attached to each cache line. This makes it nearly trivial to prove that the following theorem holds:

Theorem 5.1. *MI* ⊑ *Seq* → *MIdup* ⊑ *SeqDup*.

This means that we can *reuse the correctness of MI opaquely* to get a proof of a system that handles multiple cache lines. Next, we need the following theorem:

Theorem 5.2. *MImult* ⊑ *MIdup*.

Proof sketch. Using a lock-step simulation proof. This is the longest proof, and even though it is conceptually simple, it is quite tedious in practice because one system is running a single state machine at a higher level, and the specification is running multiple state machines for each cache line. One has to therefore specify many properties to relate the states of these two systems which then have to be proven for every step. One additional property that is required to relate two states, which is not required in the original *MI* proof, is that the tag of a cache line within an *MI* cache in the *MIdup* system is coherent with the index of the *MI* cache within the system. This is needed because tags in our *MI* system contain the whole address, and so the cache line index of the address should be coherent with the index of the cache itself that contains it. □

Even though *SeqDup* contains multiple memories for each cache line, they are essentially banked, and only contain addresses that are supposed to be assigned to that cache line, meaning *SeqDup* ⊑ *Seq* is relatively easy to prove. The main subtlety is reasoning about the external request/response queues. In *SeqDup*, addresses are associate with different caches, which can execute in a nondeterministic order, whereas in *Seq* there is a single external request and response queue which has to be handled in order. This means that if *SeqDup* handles a request, it might not be possible to handle that request in *Seq*, because other requests have to be processed beforehand.

We therefore have to restrict *SeqDup* to only include the valid executions that can also be observed by the sequential memory with a single request/response queue for each cache. A global state is added, which determines exactly which address is now allowed to be processed in each of the caches.

6 Evaluation

To validate the practical functionality of our *MSI* protocol and evaluate its liveness (a property not covered by our formal proofs), we translated the Lean formalization of our *MSI* protocol into synthesizable Bluespec [Nikhil 2004] hardware modules. We then tested the Bluespec design with randomized tests.

Translating transition relations of the *MSI* protocol to Bluespec is straightforward thanks to the rule-based nature of Bluespec: specifically, the preconditions of each constructor of our inductive predicates (i.e. transition rule of the system verified) directly correspond to the guard of a corresponding *rule* in Bluespec, and the consequent describing the state of the module after the transition corresponds to action methods in the body of the rule. We systematically but *manually* instantiated and tested a Bluespec design from the *MSI* formalization in Lean, an effort that required approximately three person-days.

Our system features a 2-level hierarchy with two 64 KB L1 caches and a 64 KB L2 cache (While the identical L1 and L2 sizing is a simplification - as discussed in the limitations section - it is sufficient for the purpose of this evaluation .

We validated the design using two randomized differential testbenches, both comparing our system behavior against a two-ported atomic memory reference design (as shown in figure 1).

The first testbench issues randomized load and store requests to both the implementation and the specification, constrained to a single in-flight request per port. The test asserts that the responses from the two designs match until 100,000 load responses have been received. In a second similar testbench, we aim to test multiple in-flight requests. To avoid the challenge caused by nondeterministic answers from the reference design, the address space of the requests sent to these two ports are disjoint; this avoids significant complication related to memory ordering during testing, while still permitting aliasing conflicts between the two L1 caches. Our Bluespec design successfully passes these tests.

6.1 Lean Formalisation and Limitations

In this section we will discuss the lean formalisation and its limitations. The total size of the lean 4 development is 9228 lines of code excluding white space and comments [Camaioni et al. 2025].

Firstly, the nondeterminism added to *MI* and *SI* is currently only used to simplify the proof of *MI+SI* and *MSI*. Instead, we believe we could take more advantage of the nondeterminism and introduce new transitions in *MSI* that did not and could not exist in either *MI* or *SI* alone. For example, it should be possible to add a single transition from `M` directly do `S` in a cache, and show that it can be simulated by chaining rules in the existing *MSI* that go through the `I`. This would further demonstrate the flexibility of nondeterminism.

Next, the extension of the protocol to multiple cache lines currently is only for *MI* system and can only handle cases where the size of the child cache is the same as the size of the parent cache. In practice, this is not the case because child caches will want to use faster, smaller memory. In addition to that, when one cache line is in `M`, the other cache lines have to be in `I`, because inclusivity forces the same cache line in multiple caches to have the same address. With a parent that has a much larger cache, it could still be inclusive while allowing multiple caches to be in `M` for that cache line but for different addresses. In keeping with our methodology of reusing proofs and modularity, we envision to address systems with a smaller L1 than L2 cache again via a refinement strategy. We will begin by viewing our existing model with equal cache sizes as an intermediate abstraction. This model will then be constrained with preconditions that render a subset of L1 cache lines unusable and inactive (a simple refinement of the current system). The final step will then to prove that the concrete system, which will feature a physically smaller L1 cache, refines this intermediate model with inactive lines.

7 Related Work

Formal verification of cache coherence protocols using interactive theorem provers. Vijayaraghavan et al. [2015] was the first to formalise a directory-based *MSI* system in the Rocq prover. The system itself supported an arbitrary tree hierarchy of caches, however, one had to find more than 50 invariants in higher-order logic, detailed in the dissertation [Vijayaraghavan 2016], to prove the overapproximation property of the *MSI* protocol. Unlike other formalisations, this work also extended the system to support multiple cache lines. Following this work, Hemiola [Choi et al. 2022] was designed to provide a language in which one can implement and verify cache coherence protocols, providing useful shared infrastructure accross protocols for handling arbitrary memory hierarchies. They also tackled the problem of specifying and proving invariants, requiring a proof that all concurrent behaviors of the system are linearizable and using that two specify invariants by specifying traces. Finally, they had an unverified but automatic compilation from the single cache line description of the protocol to a multi cache line implementation. However, they still need around 50 invariants, and due to the nature of them being specified as traces, they are often non-local and difficult to verify. Finally, Tan et al. [2025] formalised CXL.cache [Cutress 2019] in Isabelle as an *MSI* model. They note that they needed 796 invariants and 53332 lemmas to prove the single writer, multiple reader (SWMR) property, a necessary invariant for sequential consistency. We believe that we could use our invariant and proof technique to also prove this protocol, significantly simplifying the proof. There have been other proofs of cache coherence, such as in PVS [Stoy et al. 2001] or ACL2 [Moore 1998].

Using model checkers. The Murϕ [Ip and Dill 1996] description language and compiler have been successfully used to verify many cache coherence protocols [Chou et al. 2004; Komuravelli et al. 2014; Oswald 2023; Park and Dill 1996; Zhang et al. 2010]. These systems are often parametric in the number of caches, however, they will mainly model a single cache line, as well as bounded, in order, message queues. Next, Emerson and Kahlon [2003] prove a snoop-based *MSI* protocol and McMillan [2001] prove the FLASH protocol correct using the SMV proof assistant based on symbolic model checking.

Compositional memory models. Related to the compositional aspect, there has been work exploring compositional memory models [Goens et al. 2023]. Our work tackles the single memory model, i.e. sequential consistency, and in contrast compose and decompose the coherence protocols.

Invariant generation and taxonomies. For distributed systems (not specific or evaluated on cache-coherence protocols) there have been prior works that introduced automatic invariant discovery [Yao et al. 2022], as well as invariant taxonomies [Zhang et al. 2024, 2025] to be used with model

checkers. These methods generally assume the presence of a complete histories of visited states to express invariants on, which works well with model checkers but can be more tedious when using interactive theorem provers. Li et al. [2016] generated invariants automatically from a small reference model of a cache coherence protocol and generalized it to apply it to the FLASH protocol in Isabelle.

8 Conclusion and Future Work

To conclude, this work demonstrates that verifying the correctness of *MSI* becomes significantly simpler by decomposing the problem into smaller subproblems: *MI* and *SI*. We establish the correctness of these two subprotocols and then derive the correctness of *MSI*. A key insight is that the invariants required to describe all three systems are local rather than global. Moreover, we introduce a taxonomy that captures a uniform structure across *MI*, *SI*, and the full *MSI* protocol.

This compositional approach not only simplifies correctness proofs but also opens new research directions. One important advantage is proof reuse: for instance, extending *MSI* to *MESI*—by adding an Exclusive (E) state—could potentially be achieved by defining and verifying an EI protocol and then composing it with the already verified *MI* and *MI* protocols. Crucially, *MI* and *SI* need not be reverified, enabling scalable development of richer coherence protocols.

Another promising direction is to push further the nondeterminism. By proving that a highly nondeterministic, generalized, cache-coherence protocol (to be defined, encompassing a superset of *MOESI* behaviors) refines to a sequential specification once, one could then show that any concrete implementation (using an arbitrary subset of *MOESI* states, with various associativity architectures, NoC reordering policies, replacement policies, inclusivity policy, downgrade scheduling policies), is a refinement of the general nondeterministic model, and thus, by transitivity, also conforms to sequential consistency.

Data-Availability Statement

The artifact associated with this paper is publicly available and can be accessed via its DOI: [Camaioni et al. 2025]. All data and materials necessary to reproduce the results presented in this work are contained within the archived package. The authors encourage reuse and verification of the artifact in accordance with open-science best practices.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. Thomas Bourgeat and Yann Herklotz were partially supported by the Swiss State Secretariat for Education, Research, and Innovation (SERI) through the SwissChips research project.

References

Martina Camaioni, Yann Herklotz, Tz-Ching Yu, and Thomas Bourgeat. 2025. Compositional MSI Proof. Zenodo. doi:10.5281/zenodo.17805558

Joonwon Choi, Adam Chlipala, and Arvind. 2022. Hemiola: A DSL and Verification Tools to Guide Design and Proof of Hierarchical Cache-Coherence Protocols. In *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Springer International Publishing, Cham, 317–339. doi:10.1007/978-3-031-13188-2_16

Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. 2004. *A Simple Method for Parameterized Verification of Cache Coherence Protocols.* Springer Berlin Heidelberg, 382–398. doi:10.1007/978-3-540-30494-4_27

Ian Cutress. 2019. CXL Specification 1.0 Released: New Industry High-Speed Interconnect From Intel. (2019). https://bit.ly/cxl-spec

E. Allen Emerson and Vineet Kahlon. 2003. *Exact and Efficient Verification of Parameterized Cache Coherence Protocols.* Springer Berlin Heidelberg, 247–262. doi:10.1007/978-3-540-39724-3_22

Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 1145–1168. doi:10.1145/3591267

C. Norris Ip and David L. Dill. 1996. Verifying Systems with Replicated Components in Murϕ. In *Computer Aided Verification*, Rajeev Alur and Thomas A. Henzinger (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 147–158.

Rakesh Komuravelli, Sarita V. Adve, and Ching-Tsun Chou. 2014. Revisiting the Complexity of Hardware Cache Coherence and Some Implications. *ACM Transactions on Architecture and Code Optimization* 11, 4 (Dec. 2014), 1–22. doi:10.1145/2663345

Yongjian Li, Kaiqiang Duan, Yi Lv, Jun Pang, and Shaowei Cai. 2016. A Novel Approach to Parameterized Verification of Cache Coherence Protocols. In *2016 IEEE 34th International Conference on Computer Design (ICCD).* 560–567. doi:10.1109/ICCD.2016.7753341

K. L. McMillan. 2001. *Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking.* Springer Berlin Heidelberg, 179–195. doi:10.1007/3-540-44798-9_17

J. Strother Moore. 1998. *An ACL2 Proof of Write Invalidate Cache Coherence.* Springer Berlin Heidelberg, 29–38. doi:10.1007/bfb0028728

R. Nikhil. 2004. 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.* 69–70. doi:10.1109/MEMCOD.2004.1459818

Nicolai Alexander Oswald. 2023. *Automatic Generation of Highly Concurrent, Hierarchical and Heterogeneous Cache Coherence Protocols From Atomic Specifications.* Ph. D. Dissertation. doi:10.7488/ERA/3375

Seungjoon Park and David L. Dill. 1996. 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 (SPAA '96).* ACM Press, 288–296. doi:10.1145/237502.237573

Joseph Stoy, Xiaowei Shen, and Arvind. 2001. *Proofs of Correctness of Cache-Coherence Protocols.* Springer Berlin Heidelberg, 43–71. doi:10.1007/3-540-45251-6_4

Chengsong Tan, Alastair F. Donaldson, and John Wickerson. 2025. Formalising CXL Cache Coherence. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25).* Association for Computing Machinery, New York, NY, USA, 437–450. doi:10.1145/3676641.3715999

Muralidaran Vijayaraghavan. 2016. *Modular Verification of Hardware Systems.* Ph. D. Dissertation.

Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. 2015. *Modular Deductive Verification of Multiprocessor Hardware Designs.* Springer International Publishing, 109–127. doi:10.1007/978-3-319-21668-3_7

Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 485–501. https://www.usenix.org/conference/osdi22/presentation/yao

Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. 2010. Fractal Coherence: Scalably Verifiable Cache Coherence. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 471–482. doi:10.1109/micro.2010.11

Tony Nuda Zhang, Travis Hance, Manos Kapritsos, Tej Chajed, and Bryan Parno. 2024. Inductive Invariants That Spark Joy: Using Invariant Taxonomies to Streamline Distributed Protocol Proofs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 837–853. https://www.usenix.org/conference/osdi24/presentation/zhang-nuda

Tony Nuda Zhang, Keshav Singh, Tej Chajed, Manos Kapritsos, and Bryan Parno. 2025. Basilisk: Using Provenance Invariants to Automate Proofs of Undecidable Protocols. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2025). https://www.usenix.org/conference/osdi25/presentation/zhang-tony

Received 2025-09-12; accepted 2025-11-13