A Formally Verified Foundation for Compositional Heterogeneous Coherence

AN QI ZHANG, University of Utah, USA
ANDRÉS GOENS, TU Darmstadt, Germany
DANIEL SORIN, Duke University, USA
VIJAY NAGARAJAN, University of Utah, USA

Modern processors integrate heterogeneous devices to expose unified shared memory. Yet, the de-facto design pattern used to compose their disparate coherence protocols lacks a formal foundation. This leaves the door open for subtle consistency bugs, in a critical gap between practice and correctness. This paper provides the first formal, machine-checked proof that a de-facto design pattern, which we call the Principle of Synchronous Propagation, is correct. Leveraging a new unifying abstraction for coherence protocols, our central theorem (machine checked in Lean) proves that Synchronous Propagation is sufficient to guarantee the Compound Memory Consistency Model for a wide class of protocols. Our work provides long-needed assurance for current designs and delivers a reusable, compositional framework for verifying future heterogeneous systems.

CCS Concepts: • **Hardware** → **Theorem proving and SAT solving**; *Model checking*; • **Computer systems organization** → *Multicore architectures*; • **Theory of computation** → *Axiomatic semantics*.

Additional Key Words and Phrases: Cache Coherence Protocol, Composition, Compound MCM

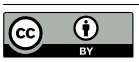
ACM Reference Format:
An Qi Zhang, Andrés Goens, Daniel Sorin, and Vijay Nagarajan. 2026. A Formally Verified Foundation for Compositional Heterogeneous Coherence. *Proc. ACM Program. Lang.* 10, PLDI, Article 272 (June 2026), 25 pages. https://doi.org/10.1145/3808350

1 Introduction

Today's processors are ubiquitously heterogeneous, often fusing different types of devices (e.g., CPUs, GPUs, Network cards) to support unified shared memory, as illustrated in Figure 1. The NVIDIA Grace Hopper superchip, for instance, fuses an ARM CPU with an NVIDIA GPU using the AMBA CHI cache-coherent interconnect [48]. This tight integration raises a fundamental challenge for language and compiler design: how can one program a system where each component has a distinct memory consistency model (MCM)? Recent work has established Compound Memory Consistency Models (CMCMs) as the formal specification for such systems [17]. The CMCM is a critical abstraction, as it allows existing, unmodified compilers for each device to be reused.

This specification, however, leads to the key question: how can it be implemented? How does a hardware designer achieve CMCM? There is a misconception that simply using a global coherence interconnect like CXL [2] or CHI [1] magically solves the problem. It does not. Designers must still create complex interface logic (shims) to fuse local device coherence protocols (e.g., a CPU's or a GPU's) with the global interconnect. Similarly, one might wonder whether these are "merely" academic questions; perhaps, industrial systems have mastered heterogeneous coherence already.

Authors' Contact Information: An Qi Zhang, University of Utah, Salt Lake City, USA, an.qi.zhang@utah.edu; Andrés Goens, TU Darmstadt, Darmstadt, Germany, andres.goens@tu-darmstadt.de; Daniel Sorin, Duke University, Durham, USA, sorin@ee.duke.edu; Vijay Nagarajan, University of Utah, Salt Lake City, USA, vijay@cs.utah.edu.



ACM 2475-1421/2026/6-ART272
https://doi.org/10.1145/3808350

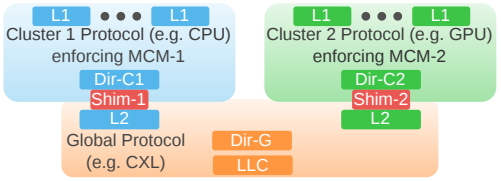


Fig. 1. Heterogeneous processor of two or more devices (clusters)— each cluster using a distinct coherence protocol enforcing a distinct MCM—fused using a global protocol. The central challenge addressed by this work is to formally prove what rules the 'shim' logic must follow to correctly compose these disparate protocols.

We argue not. Our study (Section 2) reveals that even a commercial chip does not enforce a CMCM in an edge case: the handling of TSO/RC execution modes in Apple chips. Even if this edge case was not expected to be exercised, one might still be surprised that a seemingly straightforward composition of devices does not enforce a CMCM. This proves the problem is far from solved, and highlights the critical need for a foundational approach to implementing CMCMs.

This paper thus addresses the formal foundation for implementing CMCMs. We do so by capturing ideas from recent industrial systems [44, 48] and academic proposals [22, 23, 32], which have converged on a hierarchical approach to heterogeneous coherence. We call it the Principle of Synchronous Propagation: When a device-internal protocol propagates a memory operation to other threads within that device, leverage the global protocol to also propagate it to other devices within the system. This design rule is attractive for its modularity and performance, as it leaves internal protocols untouched and avoids unnecessary global coordination.

Synchronous Propagation, however, has been validated only by (litmus) testing [22, 23, 32]—a method fundamentally incapable of providing a full correctness guarantee or identifying the precise environmental conditions required for it to work. Crucial questions therefore remain unanswered: Does the Synchronous Propagation principle actually guarantee the CMCM? Are the widely-deployed global coherence interconnects such as CXL [2] or CHI [1], strong enough to enforce CMCM? For what classes of internal protocols does the guarantee hold?

Answering these questions is critical to building heterogeneous systems, but a foundational approach to composition of coherence protocols has proved elusive. The major impediment to this is that the protocols themselves are fundamentally different: CPUs use strict, invalidation-based protocols that enforce the Single-Writer-Multiple-Reader (SWMR) invariant [29], while accelerators often use weak, software-directed ones —a diversity that has stymied prior formal analysis.

This paper provides the first formal framework for abstracting diverse coherence protocols to enable reasoning about their composition. Our framework achieves this by integrating three key concepts: abstract states (capturing and abstracting coherence permissions), consistency labels (making consistency a first-class citizen in reasoning about coherence), and linearization points (to formally reason about the ordering and composition of protocol events). We show that this abstraction is not a toy: it can capture industrial-strength protocol specifications such as CXL [2] as well as sophisticated academic protocols such as Denovo [12].

Using this abstraction, we prove our central theorem: for any device protocol that satisfies our abstract axioms, its composition using Synchronous Propagation over a global interconnect that satisfies the SWMR invariant (such as CXL or CHI), is guaranteed to enforce CMCM. This 26K-line mechanized proof not only puts heterogeneous shared memory on a firmer foundation, but it also establishes a compositional verification framework; the intractable global problem is now reduced to the manageable tasks of verifying that individual protocols satisfy our abstraction's axioms and that the global protocol satisfies SWMR. The contributions of this paper are as follows:

- A new, unifying abstraction for coherence protocols that is capable of modeling diverse CPU-style as well as GPU-style protocols, as well as their composition with global protocols.

- A formalization of the principle of Synchronous Propagation with a proof that, in combination with a global interconnect that guarantees SWMR, it is sufficient to guarantee the CMCM. This formalization is fully mechanized in the Lean theorem prover and is publicly available [49, 50].
- A reusable verification framework, consisting of a set of well-defined, modular properties of the constituent protocols, that enables architects to design future heterogeneous systems that are correct by construction.

2 Motivating Case Study: Checking Compound Consistency in Apple M-Series Chips

Composing coherence protocols is not a straightforward, solved problem. There are subtle issues that can result in miscompilation and concurrency bugs, even when the individual protocols guarantee the respective memory consistency models. In this section we discuss a concrete case study where we found such subtleties in commercial off-the-shelf hardware, showing this is both relevant and not obvious.

Apple's M-series chips implement the ARMv8 Instruction Set Architecture (ISA), including its MCM [3, 38]. For efficient emulation of x86 architectures, they include a "TSO Mode," which changes the operation of a hardware thread to the TSO memory model [33]. How does such a TSO-mode thread synchronize with a regular ARM thread in an M-series chip?

We investigated this behavior empirically on an Apple M1 mini (2020). To test the MCM, we generated and executed litmus tests using a modification of the diy7 Litmus tool [4]. We used the TSO-enabler kernel extension[1], which enabled us to spawn threads in TSO mode without Rosetta 2.

```
// T1 (ARM)              // T2 (TSO)
m1: store(data, 1)       m3: r1 = load.acq(flag)
DMB St
m2: store(flag, 1)       m4: r2 = load(data)
```

Consider the "Message Passing" (MP) litmus test shown above, where thread T1 is mapped to an ARM core and T2 is mapped to a TSO core. Can thread T2 ever observe the value r1 = 1 and the value r2 = 0? The compound MCM [17, 32] of ARMv8 and TSO— the natural composition of the two MCMs —would disallow this behavior. Nevertheless, we observed this unexpected behavior on the M1 mini!

The preserved program order (ppo) of TSO [41] mode requires the two reads in T2 to (appear to) execute in order. The DMB ST fence in T1 requires the two writes to (appear to) execute in order. In the M1, this TSO-mode ppo does not seem to synchronize with the ARM fence in a different thread.

We see the same issue if we change the producer to TSO and the consumer to ARM, but instead have a DMB LD instruction between the two reads. We found similar issues for other litmus tests, which all consistently show that TSO reads and writes do not synchronize with ARM fences. On the other hand, the litmus tests with only one type of thread—either all TSO-mode or all ARM—behave as expected.

We do not present this case study to imply that Apple has a bug[2], particularly given that TSO mode is not widely publicized or documented. However, the results do illustrate that the composition of coherence protocols respecting compound consistency is complicated. It is not automatically solved by using a strong global protocol, nor is it a solved problem in industry. In the rest of the paper we will show how we can provably compose protocols in a way that respects compound consistency.

[1] https://github.com/saagarjha/TSOEnabler
[2] We have conveyed these results to Apple.

3 Background: Heterogeneous Coherence and Consistency

This section builds the necessary technical context for our work. We begin by outlining our system model of interconnected heterogeneous devices (clusters). We then characterize the different building blocks of these systems by introducing the two dominant classes of coherence protocols used within these clusters. With these components defined, we describe the Principle of Synchronous Propagation, the de facto method used to fuse them together. This leads to the final piece of background: the Compound Memory Consistency Model (CMCM), the formal specification against which the correctness of this fused system is judged.

3.1 System Model

Consider a heterogeneous computer (Figure 1) which includes some number of clusters—a cluster can be a chiplet, socket, or a chip—connected using a global coherence interconnect to expose unified shared memory. A cluster represents a self-contained processing unit, such as a multi-core CPU or a GPU. A key feature of our model is that each cluster manages its own private cache hierarchy: in the example shown in Figure 1, each cluster has L1 caches and a cluster-level last-level-cache that is the L2. The caches within the cluster are kept coherent using an internal coherence protocol enforcing its internal MCM. Our framework treats these internal protocols as black boxes, an abstraction that represents a modular, compositional design.

The device clusters are linked by a global coherence interconnect that enables communication and sharing between the clusters by keeping the last-level-caches of each cluster coherent. We assume this interconnect provides the Single-Writer-Multiple-Reader (SWMR) invariant for any given memory location [29]. This guarantee, which is satisfied by existing standards like CXL [2] and CHI [1], ensures that at any point in time, a memory address can have at most one writer or, in the absence of a writer, multiple readers. Note that Figure 1 shows a global LLC that is integrated with the global directory, but our model also captures systems without this global LLC.

To bridge the semantic gap between the diverse internal protocols and the unified global protocol, each cluster interfaces with the interconnect through a dedicated shim. This shim logic is responsible for translating coherence requests between the cluster's private domain and the global SWMR domain, and it is where the Synchronous Propagation rule is implemented.

3.2 Coherence Protocols

While a vast number of coherence protocols exist in literature and practice, they can be broadly categorized by the invariants they enforce. For the scope of this work, we focus on two dominant classes that are widely implemented in modern processors.

The first class consists of protocols that enforce the Single-Writer-Multiple-Reader (SWMR) invariant. These protocols, which include the MOESIF family and form the basis of industrial specifications like CXL [2] and CHI [1], are typically employed in CPUs. Their defining characteristic is that a write to a memory location eagerly invalidates all other cached copies, ensuring there is only ever one writer; there can, however, be multiple readers that can concurrently read cached locations. It is worth noting that SWMR does not refer to one single protocol, rather it encompasses dozens of protocols that have been proposed in the literature and appeared in real products.

We consider memory systems with individual cluster protocols and a global protocol. Besides caches, each of these individual protocols has a directory which tracks information and coordinates the caches. In each cluster, each core has a cache. In the global protocol, on the other hand, every cache corresponds to an entire cluster. Each entry in a cache consists of a line of data corresponding to its address, and a state that indicates its access permissions. In an SWMR protocol, the state (with corresponding permissions) is tracked by the directory to enforce the SWMR property. Permissions

here refers to read, read-write permissions, or none. If a request does not have permissions in the cache, it will request these from the directory. The directory tracks if there is a Single-Writer (SW) with read-write permissions, or Multiple-Readers (MR) with read permissions. Upon a cache's request for read-write permissions, the SWMR directory downgrades prior SW or MR caches if they exist. Upon a read permission request, the directory downgrades an SW cache, if there is one. The requesting cache enters a transient state upon sending its request, to note it's waiting for a response. Here, we distinguish between *stable states*, where the cache has completed all previous requests, and *transient states*, which indicate the cache is still waiting for a request to be fulfilled.

The second class eschews the strict SWMR invariant in favor of directly enforcing a relaxed memory model. Protocols that enforce Release Consistency are, in particular, commonly employed in GPUs, and variants of these protocols are called RCC (Release-Consistency-directed-Coherence) [29]. Rather than eagerly invalidating on every write, they often buffer writes locally. These buffered writes are guaranteed to be propagated to the wider system only upon explicit synchronization operations (e.g., a release), while caches are self-invalidated on other synchronization operations (e.g., an acquire). Again, RCC is not a single protocol but a broad class of protocols that can be implemented with different design trade-offs. For example, Denovo [12] is an RCC variant that obtains exclusive ownership on writes, but does not send invalidations to other cores, relying instead on software-directed self-invalidation. Another variant, Lazy Release Consistency Coherence (L-RCC) [6], optimizes write propagation by delaying it until a remote `acquire` is detected, as opposed to eagerly propagating writes on a `release`.

An RCC protocol also consists of a directory and caches. To implement buffered writes, in many RCC variants, the directory does not enforce SWMR. Instead, cache entries have states with read and read-write analogues, but they do not need to respect SWMR. A cache may have no permissions, where it must read data from the directory, for both a read and write. A read only returns the value to the core, leaving it clean (with read permissions) in the cache. In contrast, a write then marks the cache entry as dirty, forming an equivalent to read-write permissions. Upon a release operation, the cache writes cache lines back to the directory. Conversely, an acquire operation invalidates clean cache entries. The invalidated lines have no permissions, which forces subsequent reads to fetch a fresh value.

Protocol Assumptions. Our analysis focuses on the protocol classes prevalent in today's high-performance systems. For CPU-style SWMR protocols, we model directory-based designs, as they represent the de facto scalable design pattern for modern many-core chips. For GPU-style RCC protocols, we acknowledge the existence of scoped coherence [29, 42], where operations are synchronized at different levels of the thread and cache hierarchy (e.g., CTA-scope vs. system-scope in NVIDIA's parlance). However, since our work's focus is on the mechanism of inter-cluster coherence, which is governed by operations that propagate to the most inclusive scope (cf. system scope), we do not explicitly model the intra-cluster scope hierarchy. Furthermore, the protocols we have modeled are multi-copy-atomic (MCA). SWMR protocols are always MCA. RCC-variants may be MCA or non-MCA; some commercial variants—such as those from NVIDIA—enforce non-MCA variants. Our formalized RCC variants maintain MCA by avoiding the non-MCA corner case where an acquire on a locally dirty block requires a writeback before the read completes. Modeling non-MCA protocols is future work. Our framework currently requires the global interconnect to enforce SWMR. This assumption is grounded in industrial practice—ARM CHI and CXL are both SWMR-enforcing interconnects and represent the de facto standard. Weakening the interconnect to an RC model would eliminate the guarantee of a single global ordering for cluster-level operations, fundamentally complicating shim design as well as the proof structure; we leave this generalization to future work. We also acknowledge that there are other protocol classes that have been explored in academic proposals, such as those based on timestamps [43] or updates; those fall outside the

scope of this paper. Finally, we only verify safety of protocols, not deadlock-freedom or liveness which is reserved for future work.

3.3 Synchronous Propagation

Connecting back to our system model, this paper focuses on heterogeneous systems that fuse clusters implementing SWMR and RCC protocols. These clusters are connected via a global interconnect that itself enforces the SWMR invariant. This assumption is grounded in the reality of modern industrial designs. For instance, early AMD APUs fused an SWMR-based CPU protocol with a simpler GPU protocol using an SWMR-enforcing global fabric [9]. More recently, the NVIDIA Grace Hopper superchip fuses an ARM CPU with a Hopper GPU, again over the SWMR-compliant CHI interconnect [30].

A key feature of these industrial designs and academic proposals like HieraGen [31] and HeteroGen [32] is that they are compositional—the internal protocols of the clusters are treated as black boxes and remain unmodified. The integration is handled entirely by the shim logic, which translates requests between the internal protocol and the global protocol. These systems are all underpinned by a common design pattern for this translation, which we call the Principle of Synchronous Propagation:

- **Outgoing Requests**: A memory request originating from a core within a cluster is propagated to the global interconnect if and only if that request reaches that cluster's main coherence point (e.g., its directory or last-level cache) and the cluster does not already hold sufficient permissions to service the request locally. This propagation is synchronous: the local request encapsulates the global request within it, ensuring the two are tightly coupled. Requests handled entirely within a core's private caches (e.g., a write to a line in the 'Modified' state), or serviced at the directory with existing permissions are not propagated globally.
- **Incoming Requests**: A request arriving from the global interconnect (e.g., an invalidation or a forwarded read request) is injected into the target cluster's internal protocol as if it were a local request originating from a pseudo-core at the cluster's boundary. Again, this injection happens synchronously.

This two-part rule is both modular and performant. The first rule prevents unnecessary global traffic, while the second ensures that external coherence demands are processed using the cluster's existing, unmodified internal protocol. It is this practical and intuitive rule whose correctness we seek to formally establish.

3.4 Correctness Condition: Compound Consistency

Given a system that fuses disparate SWMR and RCC protocols using the Synchronous Propagation principle, what is the standard for its correctness? A programmer writing code for the CPU part of the chip expects that architecture's memory ordering rules to hold, while a programmer on the GPU expects its distinct, more relaxed model to be enforced. A correct heterogeneous system must somehow satisfy both of these expectations simultaneously.

The formal answer to this question is the Compound Memory Consistency Model (CMCM) [17, 32]. A correct execution trace under CMCM, when projected onto the threads of a single device, remains a valid execution under that device's native MCM. For example, all memory operations issued by the CPU cluster must appear to obey the CPU MCM, while all operations from the GPU cluster must concurrently appear to obey the GPU's MCM.

More precisely, a valid execution can be defined in a *declarative* or *axiomatic* model by defining relations on the set of memory events in a (candidate) execution [5]. Memory events are reads, writes, and memory fences in each of the threads. Between these events, the `rf` relation ("reads

from") relates a write event w with every read event r that sees the written value. The co relation ("coherence") relates two writes to the same location by the order in which they are observed. Finally, the po relation orders all events in every thread by their program order, and ppo is the subset of po that is preserved by the memory model. From these base relations, memory models usually derive other relations: The fr relation ("from-reads") is defined as $\texttt{fr} = \texttt{rf}^{-1}; \texttt{co}$, where $^{-1}$ denotes the inverse relation and ; the sequential composition. It relates all read events with the write that overwrites the value they saw. Similarly, if we restrict the rf relation to "external" events, i.e. where the read and write are in different threads, we get $\texttt{rfe} = \texttt{rf} \setminus \text{same-thread}$.

Memory models are then defined by predicates on these relations. For an MCA memory model, the defining "axiom" states that the union of these relations is acyclic:

$$\text{acyclic}(\texttt{ppo} \cup \texttt{rfe} \cup \texttt{fr} \cup \texttt{co}) \tag{1}$$

For a compound memory model (in the MCA case) thus, the memory-model-specific relation ppo is defined as $\texttt{ppo} = \cup_i \texttt{ppo}_i$, where $\texttt{ppo}_i$ is the ppo of the memory model on thread i [32].

The guarantee provided by CMCM is not merely a theoretical construct; it has a critical practical implication: independent compilation. If the local memory semantics are preserved for each device within the larger fused system, then the assumptions made by the compilers for those devices remain valid. This allows architects to leverage existing, highly-mature, and unmodified compilers for each component [17].

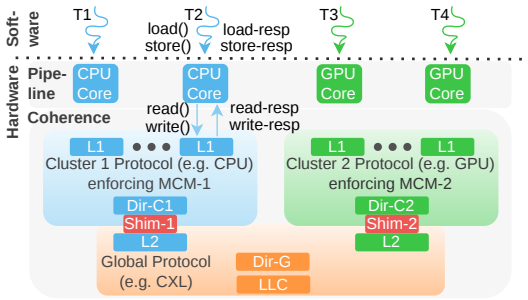


Fig. 2. The consistency effects of the coherence interface: Normally the consistency model enforced by the processor is a function of both the processor pipeline and the coherence protocol combined. Our goal here is to reason about the correctness of the 'Coherence box'. To this end, we isolate the consistency effects of a protocol.

3.5 Defining the Proof Scope: The Coherence Box

Proving that a real system satisfies CMCM, however, requires reasoning about all of its components. It is therefore crucial to distinguish between two layers of behavior, as illustrated in Figure 2. The first is the architectural MCM visible to software (the solid line in Figure 2), which is the ultimate contract with the programmer. This model's behavior is a function of both the coherence protocol and core-microarchitectural effects like pipeline reordering and store buffering. The second, lower layer is the "coherence box" itself.

This paper's primary focus is on proving the correctness of the coherence box. To do this, we analyze the MCM enforced by the protocol itself [29, 32]. We achieve this by analyzing the coherence fabric's behavior.

Consider a simplified pipeline-coherence interface modeled as delivering memory operations to the coherence box one at a time and in program order. Thus, an SWMR-enforcing protocol is said to "enforce SC", and an RCC protocol is said to "enforce RC".

Note that the MCM enforced by the Coherence Box is a standard definitional tool to isolate the guarantees of the coherence fabric [29]. In particular, the consistency labels (e.g., SC) characterize the interface the Coherence Box exposes to its client (the pipeline)—not the MCM enforced by the pipeline itself. The Coherence Box enforcing SC does not mean the pipeline will have to enforce

SC as well. The pipeline retains full ability to buffer, reorder, and overlap requests while honoring the preserved program order (ppo) of the architectural MCM. More broadly, modern pipelines tightly integrate pipeline execution with coherence for optimizations such as speculative load reordering: a core executes a load speculatively and monitors the cache line for invalidations. Our framework does not restrict such optimizations. The correctness of the pipeline-coherence interface is a separate problem and can be addressed by tools such as CCI-Check [24]. In sum, the validity of the consistency model framing of the Coherence Box rests on the principle that the fusion of two coherence protocols should be independent of the specific pipeline designs attached to them; for instance, whether a CPU pipeline implements the ARM or x86-TSO model should not fundamentally alter how the underlying coherence protocols must be combined.

Therefore, the central goal of this paper is to formally prove that the Synchronous Propagation rule correctly produces a coherence fabric that upholds the requirements of the appropriate Compound Consistency Model (e.g., an SC/RC CMCM).

4 Intuition

This section builds the intuition for our formal development. We begin by defining the precise property we need to prove, then explore the core challenges that make such a proof difficult. Finally, we present the key ideas behind our formal abstraction and proof strategy, providing a conceptual roadmap for the formal sections that follow.

To be precise, the central goal we must prove is that for any cluster protocol from the SWMR or RCC classes, its composition with a global SWMR protocol via a shim satisfying the Principle of Synchronous Propagation is sufficient to enforce the corresponding Compound Memory Consistency Model (CMCM). The type of the resulting CMCM is determined by the protocols being fused; for instance, combining an SWMR protocol (which enforces SC at 'coherence' box level) with an RCC protocol (which accordingly enforces RC) yields an SC/RC CMCM.

Our proof strategy achieves this by introducing three key formal concepts:

- **Abstract State:** To handle the challenge of local cache hits, where the state of a cache line itself must serve as a historical record of prior coherence events.
- **Consistency Labels:** To formally specify the complete set of ordering guarantees—the "contract"—that a given protocol is expected to enforce.
- **Linearization Points:** To formally reason about when operations become visible across different clusters and to connect the local ordering of events to the global order.

We now use our motivating example to explore each of these concepts in detail.

4.1 A Motivating Example: Message Passing

To build intuition, let us consider a concrete system and a classic litmus test. Our system fuses a CPU cluster (C_{CPU}) implementing a directory-based MSI (SWMR) protocol with a GPU cluster (C_{GPU}) implementing a basic RCC protocol. The global interconnect is also MSI-based.

We analyze the "message passing" (MP) litmus test shown below (a variant of the test from Section 2). A thread T_1 on C_{CPU} writes data and then a flag, while a thread T_2 on C_{GPU} waits to read the flag before reading the data. Initially, `data` and `flag` are both `0`.

```
// T1 on CPU (SC)        // T2 on GPU (RC)
m1: store(data, 1)       m3: r1 = load.acq(flag)
m2: store(flag, 1)       m4: r2 = load(data)
```

The CMCM correctness condition requires that if `m3` reads the new value 1, then `m4` *must also* read 1. The outcome where `r1=1` and `r2=0` must be forbidden. Our goal is to show how Synchronous Propagation guarantees this.

To appreciate the subtlety inherent in shim design, consider a naive but plausible performance optimization. An architect might reason that invalidations from the CPU to the GPU are potentially too slow and should be sent "off the critical path"—that is, buffered and sent asynchronously without requiring immediate acknowledgment. This seemingly reasonable design can violate the litmus test. Assume `data` and `flag` are initially in a readable state in both clusters. If a CPU writes to `data` and then to `flag`, the invalidation for `flag` could arrive at the GPU before the invalidation for `data` due to network reordering. A GPU thread could then suffer a cache miss on `flag`, fetch its new value (1), but immediately after, hit on its still-present stale copy of `data`, incorrectly reading `0`. This *forbidden outcome* demonstrates that intuitive optimizations can be wrong, proving that the precise timing and ordering guarantees of the shim—which we now analyze—are essential for correctness.

4.2 Linearization Points

Let's again assume both `data` and `flag` are initially in a readable state in both clusters. When T_1 executes `m1` (write `data`), its L1 has the line in state *S*. The request must go to the C_{CPU} directory to get exclusive ownership. Per the Synchronous Propagation (Outgoing) rule, this cluster-level event triggers a global request that invalidates the copy of `data` in C_{GPU}'s L2 cache. When this global invalidation reaches C_{GPU}, the Incoming Request rule is invoked, treating the event as stimulus to the cluster's internal RCC protocol. However, because a typical RCC protocol does not eagerly forward invalidations on a write, no further coherence action is taken within the cluster at this time. The line is invalidated at the cluster's L2, but any copies residing in the GPU's L1 caches may persist until a future synchronization operation. Importantly, because the propagation happens synchronously, the GPU cluster's copy of `data` is invalidated before `m1` completes.

A similar sequence occurs for `m2` (write `flag`), causing the copy of `flag` in C_{GPU}'s L2 cache to be invalidated before `m2` completes. Later, when T_2 executes its `acquire` read of `flag`, the acquire forces the L1 cache to be bypassed, and also causes the self-invalidation of the L1 and so `data` is self-invalidated in the GPU's L1. So when `data` is accessed, it misses the L1 and the subsequent miss in its L2 triggers a global read request; when the global read reaches C_{CPU}'s L2, the Incoming rule is invoked causing a forwarded read request in the MSI protocol that eventually returns the value 1 from the CPU.

It is important that T2 uses an `acquire` read for `flag`. A regular read could hit the stale copy of flag still in the L1 since the global invalidation only reached the L2, observing the old value—or worse, it could fetch the updated flag from L2 while data remains stale in the L1. But note, this behavior does not violate RC. Regular reads carry no ordering guarantee; the programmer must use an Acquire read for flag to guarantee that the subsequent read to data observes the updated value.

This trace seems correct, but to prove it, we need to formalize the notion of visibility and connect the local ordering within C_{CPU} to the global ordering observed by C_{GPU}. This motivates the first component of our abstraction: **linearization points**, which are logical times at which an operation's effects become visible to a set of threads.

The key insight is to distinguish between local and global visibility. An operation like `m1` achieves its *local linearization point* within C_{CPU} when it is ordered by its cluster directory. It also achieves a *global linearization point* when its effects are made visible to all clusters. The Principle of Synchronous Propagation can now be elegantly modeled as a formal constraint relating these two points: for an outgoing request, its global linearization must occur no later than its local linearization.

Since both `m1` and `m2` are ordered by the C_{CPU} directory, we know that `m1` is locally linearized before `m2`. Because the Synchronous Propagation rule creates this tight coupling between local and global linearization for both operations, it follows that `m1` is also globally linearized before `m2`. It is

this formal reasoning about the ordering of linearization points that allows us to prove the local ordering is reflected globally, which is precisely what CMCM requires.

4.3 The Challenge of Locality and Abstract State

Now consider a harder case: what if `data` and `flag` were already in an writable (*Modified*) state in T_1's L1 cache? In this scenario, both `m1` and `m2` are local L1 hits. They do not generate traffic to the C_{CPU} directory, so per Synchronous Propagation, no global invalidations are sent. How can we prove that the order between `m1` and `m2` is preserved globally?

This motivates the second part of our abstraction: tracking an **abstract state** (e.g., `read-coherent`, `write-coherent`) for each memory location. The key insight is that for T_1 to have acquired *Modified* status for a line, it *must have* previously issued a local coherence request—and consequently owing to the Outgoing Request rule of Synchronous Propagation—a global coherence request. The abstract state of the line itself—in this case, `write-coherent`—is effectively a cached proof of prior global linearization. Thus, even though the writes `m1` and `m2` are local, we can reason that the lines they modify were made globally exclusive before the writes occurred, preserving the necessary ordering. This explains why we employ abstract coherence states in our abstraction.

4.4 From Litmus Tests to General Correctness: Consistency Labels

Proving correctness for this one litmus test is illustrative but not sufficient. We must prove that *all* orderings required by the native cluster protocols are preserved globally. This forces a critical question: how do we formally capture the complete set of ordering guarantees that a given protocol like MSI or RCC is expected to provide?

This motivates the final component of our abstraction: consistency labels. Instead of reasoning about individual litmus tests, we associate every protocol request (e.g., a `GetModified` in MSI) with a label (e.g., `SC-write`) that formally specifies its required consistency semantics. These labels serve as a comprehensive specification of the entire set of orderings that a protocol must preserve. Our overall proof goal is thus transformed from verifying a potentially infinite set of litmus tests to proving a single, more general property: that for any pair of operations, the ordering dictated by their consistency labels is correctly enforced by the fused system.

This label-based approach provides a powerful separation of concerns. While assigning and verifying the labels for a complex industrial protocol is a non-trivial task in itself, it is a self-contained, local problem that can be solved with existing techniques. Our framework then takes these locally-verified components and provides the formal guarantee that they will be correctly composed across the entire system.

With all components of our abstraction now motivated, our overall proof strategy can be stated clearly. We must prove that for any pair of operations whose consistency labels dictate a preserved ordering, the Synchronous Propagation rule guarantees that their global linearization points respect that order, regardless of the initial abstract state of the data. The two scenarios we walked through represent two key cases in this general proof.

5 A Formal Abstraction for Coherence Protocols

The previous section provided the intuition behind our approach, motivating the need for a formal language that can connect low-level coherence events to high-level consistency guarantees. This section now provides the precise definitions for this formal abstraction. To make the formal concepts concrete and easy to follow, we will use the simple MSI and RCC protocols from our motivating example as running illustrations for each definition. In the next section, we show we can handle other non-trivial protocols.

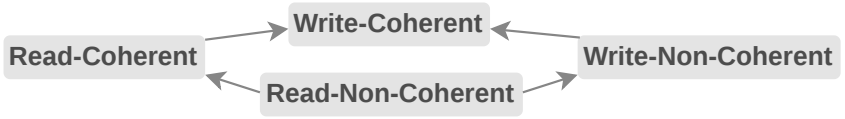


Fig. 3. The state lattice defined by the partial order: An arrow from state A to B indicates that A ≤ B.

5.1 Abstract Protocol State

We begin by defining the abstract state of a memory location. Our model simplifies the multitude of states found in concrete protocols (e.g., Modified, Shared, Invalid, Valid etc.) into a few key properties essential for reasoning about correctness. The abstract state describes a local cache's permissions for a line and whether accesses are guaranteed to be visible to other threads within that same protocol's domain.

This gives us four abstract states:

- **Write-Coherent:** Grants permission to write to a local copy. The protocol guarantees that this write is immediately made visible to all other threads it manages (e.g., by invalidating their copies). This corresponds to a `Modified` state in an MSI protocol, as well as `Exclusive` state in an MESI protocol.
- **Read-Coherent:** Grants permission to read from a local copy that is consistent with the latest visible write within the protocol's domain. This corresponds to a `Shared` state in an MSI protocol, as well as `Owned` state in an MOSI protocol.
- **Write-Non-Coherent:** Grants permission to write to a local copy, but the protocol does **not** guarantee immediate visibility to other threads. The updated value is buffered locally, and its effects are only propagated by a subsequent, explicit synchronization operation. This state is essential for modeling weak stores in RCC protocols.
- **Read-Non-Coherent:** Grants permission to read from a local copy, but the value is **not** guaranteed to be consistent with the latest visible write within the protocol. This allows for relaxed/weak reads, while stronger operations (like an `acquire` read) must bypass a copy in this state. Weak writes to a cache entry on `Read-Non-Coherent` silently upgrade to `Write-Non-Coherent`.

More formally, we define our abstract states with a tuple of (read-write) permissions and whether or not the state tuple has *coherent* read-write permissions.

Definition 5.1 (State). $\langle permissions, coherent \rangle$

We define our abstract states using a tuple:

- **permission** is a set with a total order: `none` < `read-only` < `read-write`.
- **coherence** is a boolean with the order: `non-coherent` < `coherent`.

Using this formalism, we can precisely define our four abstract states:

- **Write-Coherent** $\triangleq$ ⟨`read-write, coherent`⟩
- **Read-Coherent** $\triangleq$ ⟨`read-only, coherent`⟩
- **Write-Non-Coherent** $\triangleq$ ⟨`read-write, non-coherent`⟩
- **Read-Non-Coherent** $\triangleq$ ⟨`read-only, non-coherent`⟩

This structure defines a state lattice (Figure 3). We define a partial order (≤) on states such that for any two states s_1, s_2, we have $s_1 \leq s_2$ if s_1.`permission` ≤ s_2.`permission` and s_1.`coherence` ≤ s_2.`coherence`.

We use this partial order to determine whether a cache line has sufficient state to satisfy a given request. For each request type, we can define a minimal required state; any state greater than

or equal to that minimum in the partial order can also satisfy the request locally. For example, a request to read data in an MSI protocol requires at least the **Read-Coherent** state. Since our partial order defines that **Read-Coherent** ≤ **Write-Coherent**, a read request can also be satisfied locally if the cache line is already in the **Write-Coherent** state.

5.2 Abstract Requests and Consistency Labels

Next, we define the abstract requests that protocols issue. The core of our abstraction is the **consistency label** assigned to each request, which acts as a formal contract specifying the ordering guarantees that request must provide. We consider the following:

- `SC-Read` and `SC-Write`: These capture the strong ordering semantics of typical SWMR protocols.
- `RC-Acquire`, `RC-Release`, `Weak-Read`, and `Weak-Write`: These capture the more relaxed ordering semantics of RCC protocols.

Formally, an abstract request is a tuple consisting of three components: the operation type, its consistency label, and whether the request must obtain coherent permissions.

Definition 5.2 (Abstract Request). ⟨`op, label, coherent`⟩ where:

- `op` ∈ {`read, write`}
- `label` ∈ {`SC, Acquire, Release, Weak`}
- `coherent` ∈ {`true, false`}

Not all combinations of the ⟨`op, label, coherent`⟩ tuple are meaningful. The set of valid abstract requests is constrained by the underlying semantics of the consistency labels and the coherence property itself.

First, `Release` is a property of writes and `Acquire` is a property of reads; this invalidates combinations such as ⟨`write, Acquire, . . .`⟩. Second, **non-coherent requests** buffer operations whose visibility is delayed, which is characteristic of RCC protocols; these requests cannot carry an `SC` label, which demands immediate visibility. This leaves `Weak` and `Release` as the valid labels for non-coherent writes, and `Weak` and `Acquire` as the valid labels for non-coherent reads.

Conversely, **coherent requests** (`coherent=true`) represent operations that are immediately visible. While typically associated with SWMR protocols and their `SC` labels, they can also model synchronizing operations in advanced RCC variants. For example, a `Release` in a protocol like L-RCC that obtains coherent permissions, or a write in an ownership-based protocol like RCC-O, would be modeled as a coherent request (e.g., ⟨`write, Release, true`⟩).

For a protocol to service a request locally (i.e., from its cache without contacting the cluster directory), the cache line must be in an appropriate state. We formalize this by defining each request's **Required State**—the minimum abstract state a cache must hold.

Definition 5.3 (Request's Required State). For any abstract request r:

- For a **coherent write**, the *ReqState* is **Write-Coherent**.
- For a **coherent read**, the *ReqState* is **Read-Coherent**.
- For a **non-coherent Release write** or **Acquire read**, the *ReqState* is still **Write-Coherent** or **Read-Coherent**, respectively. Because simple RCC protocols typically hold lines in non-coherent states, these synchronizing operations are **forced to the cluster directory**.
- For **non-coherent weak** operations, the *ReqState* is **Read-Non-Coherent**, as they can act on locally cached, potentially stale data.

5.3 Linearization Points and Visibility

As motivated in Section 4, linearization points are the formal mechanism for reasoning about when an operation's effects become visible. The key is to distinguish between when an operation is visible *locally* (within its own cluster) and *globally* (to all clusters).

We define the local linearization point of a request by distinguishing between operations that are immediately visible upon a cache access and those whose visibility is determined by an interaction with the cluster directory.

Definition 5.4 (Local Linearization Point). For a request r on a line held in abstract state s by its local cache:

(1) The operation is **linearized at the cache** if and only if the state s is sufficient for the request *and* s.`coherence` is `true` (i.e., the state is **Write-Coherent** or **Read-Coherent**).
(2) The operation is **linearized at the cluster directory** if the request must be sent to the cluster directory to be serviced. This includes both requests on insufficient states (e.g., a write to a **Read-Coherent** line) and strongly consistent requests (e.g., `RC-Acquire`, `RC-Release`).
(3) The linearization of a **Weak-Write** to a non-coherent state is **deferred** to a subsequent synchronizing operation (e.g., `RC-Release`) that makes its value visible at the directory. Similarly, a **Weak-Read** on `Write-Non-Coherent` state reads a value that will be made visible later, and its linearization is deferred.
(4) The linearization of a **Weak-Read** from a **Read-Non-Coherent** state is defined as the linearization point of the preceding operation that brought the line into that state.

The global linearization point is defined by how the Synchronous Propagation rule propagates local requests globally.

Definition 5.5 (Global Linearization Point). For a request r:

(1) If r's local linearization is at the cache, its **global linearization point is also at the cache**.
(2) If r's local linearization is at the cluster directory, its **global linearization point** is when the corresponding global request, triggered by Synchronous Propagation, is serviced by the global protocol.

The validity of Case 1 in Definition 5.5 is a key consequence of our model. The global SWMR invariant, in combination with the Synchronous Propagation shim rules, guarantees that if a cluster holds a line in a locally coherent state (e.g., **Write-Coherent**), that state is consistent with the global state of the system. With these definitions, we can now clearly describe our running examples.

MSI Protocol. For a write request in our MSI example, the linearization point is determined by the cache state. If the line is held in the `Modified` state (**Write-Coherent**), the state is sufficient and coherent. Per Definition 5.4(1), the write is **locally linearized at the cache**. Its global linearization point is also at the cache, as per Definition 5.5(1). If the line is `Shared` (**Read-Coherent**), the state is insufficient for a write. Per Definition 5.4(2), the request is sent to the directory and is **locally linearized at the cluster directory**, which in turn triggers a global request that is globally linearized at the global protocol.

RCC Protocol. The RCC example highlights the importance of the deferred linearization rule. Consider a `Weak-Write` that hits a `Read-Coherent` line, transitioning it to **Write-Non-Coherent**. Because this is a weak operation whose resulting state is not coherent, per Definition 5.4(3), its local linearization is **deferred**. The effects of this write only become ordered with respect to other threads upon a subsequent `RC-Release` operation, which *is* linearized at the directory. In contrast, an `RC-Acquire` read is a strong synchronizing operation. Per Definition 5.4(2), it is always **linearized at the directory**, ensuring it observes the effects of all previously linearized writes.

6 A Compositional Verification Framework

With these definitions, we are ready to formalize our main result, that the principle of synchronous propagation for composing protocols that satisfy a set of given requirements guarantees compound consistency. For this, we assume that the individual protocols are given as state machines or, more precisely, labeled transition systems [8], describing their specification. The concrete states in the protocol should then be mapped to the abstract states specified in Section 5.1. In this section we give the conditions on those abstract states.

More concretely, our main theorem's validity relies on the assumption that the individual protocols being composed are well-behaved and correct, as well as conditions on how they are composed. This section formalizes what "well-behaved" and correct means by presenting a set axioms that serve to characterize the formal abstractions of Chapter 5.1. We also formalize the principle of synchronous propagation. This framework provides the formal preconditions for our main theorem (Theorem 6.1), establishing a compositional workflow: architects need only verify that their individual components satisfy these axioms to leverage our general proof and guarantee the correctness of the entire composed system.

6.1 Modeling Temporal Relationships

The fused system is a state machine (more precisely, a labeled transition system) whose state includes the abstract states of all memory locations in all clusters, as defined in Section 5.1. The transitions of this machine are driven by abstract coherence requests.

To reason about the composed system dynamically, we model an execution as a history of **Events**, where each event represents a discrete protocol action (e.g., a `Cache Event` or a `Directory Event`). The relationships between these events are governed by two key relations: **Ordered-Before**, which models their temporal sequence, and **Encapsulates**, which models a causal dependency through temporal containment.

For example, a cache miss that triggers a directory request results in a `Cache Event` that encapsulates a `Directory Event`. This event-based formalism provides a precise language for defining the Synchronous Propagation shim.

Formally, we model each event e as a time interval $(t_{\text{start}}, t_{\text{end}})$. The two key relations are then defined over these intervals for any two events e_1, e_2:

- **Ordered-Before:** Event e_1 is ordered before e_2, written $e_1 \prec_{\text{OB}} e_2$, if and only if $e_1.t_{\text{end}} < e_2.t_{\text{start}}$.
- **Encapsulates:** Event e_1 encapsulates e_2, written $e_1 \supset e_2$, if and only if $e_1.t_{\text{start}} < e_2.t_{\text{start}}$ and $e_2.t_{\text{end}} < e_1.t_{\text{end}}$.

6.2 The Verification Axioms

The formal guarantees of correct composition are based on a set of axioms of the cluster protocol and shim designs that fall into four main categories.

- **Well-Formedness Axioms:** These are basic sanity checks on the transition system of the protocol, ensuring that events are correctly ordered and structured. For example, we require that events at a single directory entry are ordered.
- **Abstract State Invariants:** These axioms ensure that the protocol's use of abstract states is consistent with their definitions. For instance, an SWMR protocol must verify that at most one cache can hold a line in the **Write-Coherent** state at any time.
- **Consistency Label Axioms:** These are the most critical axioms, connecting coherence actions to consistency semantics. They specify the behavior required for each consistency label. For example, a protocol using a `Non-Coherent Release` must verify that this operation forces all prior `Write-Non-Coherent` entries to be written back to the directory before it can linearize.

- **Shim Correctness Axioms:** These axioms verify that the shim logic correctly implements the two rules of Synchronous Propagation.

We first define abbreviations to aid in the definition of the Axioms. Our abbreviations in Table 1a and 1b state properties of an event, and relations between events. Figures 4-5 are short excerpts of the full list of axioms in their categories. They give a sense of the properties these axioms capture. A full list is in the appendix for the interested reader. The key consistency label axioms and shim correctness axioms are given fully in Figures 6-7.

Well-Formedness Axioms. Axiom `Cache-to-Dir` in Figure 4 captures the property that if an operation (cache event) $e_{\$}$ does not have a sufficiently strong state, then $e_{\$}$ accesses the directory through a directory event $e_{\texttt{dir}}$. This models the directory-based nature of the protocol. Similarly, Axiom `Inv.-Read` captures an RCC-style property. After an `inval`(idation) $e_{\texttt{inval}}$ at a cache entry, for any subsequent read $e_{\texttt{read}}$, $e_{\texttt{read}}$ linearizes after $e_{\texttt{inval}}$. The rest of the well-formed axioms capture similar properties, ensuring the temporal ordering of events makes sense for the protocol.

Abstract State Invariants. Axiom `Dir-WCoherent-W` in Figure 5 is an example of a SWMR-style property. It states if a directory event $e_{\texttt{dir}}$ arrives on `Write-Coherent` state, then $e_{\texttt{dir}}$ encapsulates a downgrade to the `owner`, enforcing the "Single-Writer" part of SWMR.

Axiom `Write-lin` is an example of one abstract state invariant that captures an RCC-style property. Specifically, it states that a `write` operation $e_{\texttt{write}}$ doesn't linearize before $e_{\texttt{lin}}$. This captures the behavior of write-through and write-back writes in RCC-style protocols.

We use our SWMR-property Abstract State Invariants to capture the behavior of the global interconnect. The global interconnect's role is specifically to enforce the **Single-Writer-Multiple-Reader (SWMR)** invariant. For any given memory address, the global protocol guarantees that either **(a)** at most one cluster has write permission (holding the line in a **Write-Coherent** state), or **(b)** any number of clusters have read permission (holding the line in a **Read-Coherent** state), but never both simultaneously.

Consistency Axioms. Axiom `Acquire` in Figure 6 captures an RCC style property. `Acquire` states that for an `Acquire` operation $e_{\texttt{acquire}}$ that accesses the directory in $e_{\texttt{dir}}$, $e_{\texttt{acquire}}$ subsequently `invals` other cache entries in $e_{\texttt{inval}}$ after $e_{\texttt{dir}}$. Similarly, Axiom `Release` captures when a `Release` operation $e_{\texttt{Release}}$ writes back all other cache entries in `Write-Non-Coherent` state before writing to the directory. Axiom `L-RCC` captures L-RCC's behavior that writes back `Write-Non-Coherent` cache entries upon a downgrade to a cache entry in O state. Note that the $\supset$ operator in the figure denotes encapsulation—a sub-event relationship—not temporal sequencing.

Shim Axioms. Shim Axioms `To-global-shim` and `To-cluster-shim` in Figure 7 state that the shim implements synchronous propagation. `To-global-shim` captures that if any directory access event $e_{\texttt{dir}}$ has insufficient permissions, it is translated to an access ($e_{\texttt{greq}}$) in its corresponding global cache. The key insight it captures is that any request reaching the global interconnect must be translated into an SC operation to comply with the global SWMR invariant.

`To-cluster-shim` captures that any global coherence event sent to a global cache is then translated to an appropriate local `proxyReq` cluster event. In our abstraction, the `proxyReq` is generated from a fictitious "proxy core" at the cluster boundary. The translated `proxyReq` depends on the local protocol's type:

- **Translation to an SWMR cluster:**
  - A global write-downgrade (invalidation) event $e_{\texttt{gdown}}$ is translated into a local `Coherent SC-Write` event from the proxy core. This naturally triggers the protocol's internal invalidation mechanism. Formally: $e_{\texttt{gdown}} \supset e_{\texttt{local-write}}$.

Table 1. Abbreviations of event properties and relations.

Abbrev	Description
down	Indicates e is a downgrade.
req	Refer to e's request.
MRS	Refer to e.req's minimal required state (from Section 5.1).
prevState	Refers to the state before e. i.e. the state e's request is made on.
nextState	Refers to the state after e. i.e. the state resulting from e's request.
gcacheState	For a directory event $e_{\texttt{dir}}$, refers to the state of the global cache corresponding to $e_{\texttt{dir}}$.
d_id	An auxiliary field of an event e. If an event results from a directory event de, e.d_id = de.d_id.
weak	A proposition on an event e. e has consistency label Weak.
release	A proposition on an event e. e has consistency label Release.
acquire	A proposition on an event e. e has consistency label Acquire.
write	A proposition on an event e. e has label of Weak and op of Write.
read	A proposition on an event e. e has consistency label Weak and op of Read.
wb	A proposition on an event e. e evicts a Write-Non-Coherent cache entry. e.down and e.write.
inval	A proposition on an event e. e evicts a Read-Non-Coherent cache entry. e.down and e.read

(a) Property abbreviations on a single event e.

Relation	Description
$\gtrless$	Events e_1 and e_2 are either $e_1 \prec_{OB} e_2$ or $e_2 \prec_{OB} e_1$.
linAt	For events e_1 and e_2, e_1 linAt e_2 indicates e_2 is e_1's linearization event.
sameCacheEntry	For events e_1 and e_2, e_1 sameCacheEntry e_2 indicates e_1 and e_2 are both at the same cache entry.
proxyReq	For events e_1 and e_2, where $e_2 \in$ Global, e_1.ProxyReq e_2 indicates e_1.req corresponds to e_2's global request and e_1's cluster. e_1.req is elaborated on below in the description of the Shim Axioms.

(b) Relation abbreviations between two events e_1 e_2.

  - A global read-downgrade is translated into a local Coherent SC-Read, which correctly transitions local Write-Coherent lines to Read-Coherent.
- **Translation to an RCC cluster:**
  - A global write-downgrade (invalidation) event e_{gdown} is translated into a local Non-Coherent Acquire Read event from the proxy core. In advanced RCC protocols such as RCC-O and L-RCC, caches may hold blocks in a Write-Coherent state. When a global invalidation arrives, these Write-Coherent permissions must be relinquished. Translating the invalidation into a local Acquire triggers the protocol's native state-downgrade sequence, which revokes Write-Coherent permissions. Thus: $e_{\text{gdown}} \supset e_{\text{local-acquire}}$.
  - A global read-downgrade is also translated into a local Non-Coherent Acquire Read for the same reason.

$$\frac{e_{\$}.\mathsf{MRS} \not\leq e_{\$}.\mathsf{prevState}}{\exists e_{\mathsf{dir}} : e_{\mathsf{dir}} \in \mathrm{Dir} \wedge e_{\mathsf{dir}}.\mathsf{req} = e_{\$}.\mathsf{req} \wedge e_{\$} \supset e_{\mathsf{dir}}} \text{ Cache-to-Dir}$$

$$\frac{e_{\mathsf{inval}}.\mathsf{inval} \quad e_{\mathsf{read}}.\mathsf{read} \quad e_{\mathsf{inval}} \ \mathsf{sameCacheEntry} \ e_{\mathsf{read}} \quad e_{\mathsf{inval}} \prec_{\mathsf{OB}} e_{\mathsf{read}}}{\exists e_{\mathsf{read\_lin}} : e \ \ \mathsf{linAt} \ \ e_{\mathsf{read\_lin}} \wedge e_{\mathsf{inval}} \prec_{\mathsf{OB}} e_{\mathsf{read\_lin}}} \text{ Inv.-Read}$$

Fig. 4. An excerpt of the Well-Formedness Axioms

$$\frac{e_{\mathsf{write}}.\mathsf{write} \quad e_{\mathsf{write}} \ \mathsf{linAt} \ e_{\mathsf{lin}}}{e_{\mathsf{write}} \not\prec_{OB} e_{\mathsf{lin}}} \text{ Write-lin}$$

$$\frac{e_{\mathsf{dir}}.\mathsf{prevState} = \mathtt{Write\text{-}Coherent} \quad e_{\mathsf{dir}}.\mathsf{req.coh} = \mathtt{Write\text{-}Coherent}}{\exists e_{\mathsf{down}} : e_{\mathsf{down}}.\mathsf{down}, e_{\mathsf{down}}.\mathsf{req} = e_{\mathsf{dir}}.\mathsf{req}, e_{\mathsf{dir}} \supset e_{\mathsf{down}}, e_{\mathsf{down}}.\mathsf{id} = e_{\mathsf{dir}}.\mathsf{owner}} \text{ Dir-WCoherent-W}$$

Fig. 5. An excerpt of the Abstract-State Axioms

$$\frac{e_{\mathsf{acq}}.\mathsf{acq} \quad e_{\mathsf{dir}} \in \mathrm{Dir} \quad e_{\mathsf{acq}} \supset e_{\mathsf{dir}} \quad a \neq e_{\mathsf{acq}}.\mathsf{addr}}{\exists e_{\mathsf{inval}} : e_{\mathsf{inval}}.\mathsf{inval} \wedge e_{\mathsf{inval}}.\mathsf{addr} = a \wedge e_{\mathsf{acq}} \supset e_{\mathsf{inval}} \wedge e_{\mathsf{dir}} \prec_{OB} e_{\mathsf{inval}}} \text{ Acquire}$$

$$\frac{e_{\mathsf{rel}}.\mathsf{rel} \quad e_{\mathsf{dir}} \in \mathrm{Dir} \quad e_{\mathsf{rel}} \supset e_{\mathsf{dir}} \quad a \neq e_{\mathsf{rel}}.\mathsf{addr}}{\exists e_{\mathsf{wb}} : e_{\mathsf{wb}}.\mathsf{wb} \wedge e_{\mathsf{wb}}.\mathsf{addr} = a \wedge e_{\mathsf{wb}} \supset e_{\mathsf{dir}} \wedge e_{\mathsf{rel}} \supset e_{\mathsf{wb}} \wedge e_{\mathsf{wb}} \prec_{OB} e_{\mathsf{dir}}} \text{ Release}$$

$$\frac{e_{\mathsf{down}}.\mathsf{prevState} = \mathtt{Write\text{-}Coherent} \quad a \neq e_{\mathsf{down}}.\mathsf{addr}}{\exists e_{\mathsf{down}}, e_{\mathsf{wb}} : e_{\mathsf{down}}.\mathsf{down} \wedge e_{\mathsf{wb}}.\mathsf{wb} \wedge e_{\mathsf{wb}}.\mathsf{addr} = a \wedge e_{\mathsf{down}} \supset e_{\mathsf{wb}}} \text{ L-RCC}$$

Fig. 6. The Consistency Axioms.

$$\frac{e_{\mathsf{dir}} \in \mathrm{Dir} \quad e_{\mathsf{dir}} \in \mathrm{Cluster} \quad e_{\mathsf{dir}}.\mathsf{gcacheState} \not\leq e_{\mathsf{dir}}.\mathsf{nextState}}{\exists e_{\mathsf{greq}} \in \mathrm{Cache}, e_{\mathsf{greq}} \in \mathrm{Global}, e_{\mathsf{dir}} \supset e_{\mathsf{greq}}, e_{\mathsf{greq}}.\mathsf{req} = \langle e_{\mathsf{dir}}.\mathsf{req.op}, \mathsf{SC}, \mathtt{Coherent} \rangle} \text{ To-global}$$

$$\frac{e_{\mathsf{gdown}} \in \mathtt{Cache} \quad e_{\mathsf{gdown}} \in \mathtt{Global} \quad e_{\mathsf{gdown}}.\mathsf{down}}{\exists e_{\mathsf{creq}} : e_{\mathsf{creq}}.\mathsf{proxyReq} \ \ e_{\mathsf{gdown}} \wedge e_{\mathsf{creq}} \in \mathrm{Cluster} \wedge e_{\mathsf{gdown}} \supset e_{\mathsf{creq}}} \text{ To-cluster}$$

Fig. 7. The Shim Axioms.

6.3 The Main Theorem

Our central theorem states that any system constructed according to these rules is correct with respect to the Compound Memory Consistency Model.

Theorem 6.1 (Correctness of Synchronous Propagation). *For any two protocols, P_1 and P_2, that are valid instantiations of our abstraction with corresponding consistency models M_1 and M_2, their composition using a Synchronous Propagation shim over a global interconnect satisfying the SWMR invariant correctly enforces the Compound Memory Consistency Model of M_1 composed with M_2, CMCM(M_1, M_2).*

6.4 Proof Sketch

Here, we provide a high-level sketch of the argument. The proof's core strategy is to show that the ordering of **global linearization points** always respects the ordering requirements dictated by the **consistency labels** of the original requests. This is sufficient because the sequence of global linearization points defines the system's unified, global execution trace. By proving that this trace respects the consistency contract of every individual operation, we guarantee that any projection of the trace onto a single device will be a valid execution under its native memory model—which is the core property of Compound Consistency [17, 32]. More precisely, assuming a contradiction to Axiom 1, there is a cycle among the linearization points (abbreviated to LPs) in $\mathtt{ppo} \cup \mathtt{rfe} \cup \mathtt{fr} \cup \mathtt{co}$. We define a relation between the LPs called the Linearization-Link (abbreviated Lin-Link, as $\rightsquigarrow$), defining a Global Memory Order (GMO) between them. Lin-Link is the irreflexive subset of the union of $\prec_{OB}$, $\supset$, $\subset$. $e_1 \subset e_2$ is the reverse direction of $\supset$, where $e_2 \supset e_1 = e_1 \subset e_2$. We prove this cannot happen by contradiction, by showing that $\mathtt{ppo} \cup \mathtt{rfe} \cup \mathtt{fr} \cup \mathtt{co} \subseteq \rightsquigarrow$, i.e. these relations respect the linearization order $\rightsquigarrow$. Since `rfe`, `fr` and `co` are model-independent observations in the execution and are ordered by assumption, showing they are in $\rightsquigarrow$ is a straight-forward sanity-check on the formalism. We thus need to show that $\mathtt{ppo} \subseteq \rightsquigarrow$, which is the memory-model-specific part of the relation. The main theorem shows ppo related request events' LPs are $\prec_{OB}$. By definition, $\prec_{OB} \subseteq \rightsquigarrow$, meaning it is satisfied.

The proof proceeds by analyzing the two ways an operation can be locally linearized, as defined in Section 5.4.

Case 1: Directory-Linearized Operations. For an operation that is linearized at its cluster directory (e.g., a cache miss or an `RC-Acquire`), the **Synchronous Propagation** rule dictates that the local directory `Event` **encapsulates** a corresponding global `Event`. Note that the local protocol correctly orders its directory events according to its consistency labels. Because of the tight causal link from encapsulation, this correct local ordering is directly translated into a correct global ordering of the corresponding global linearization points.

Case 2: Cache-Linearized Operations. For an operation linearized at a local cache, no new global event is generated. This can only occur if the cache line is already in a **coherent abstract state** (e.g., `Write-Coherent`). The key invariant we prove is that, due to the interplay between Synchronous Propagation and the global SWMR protocol, a locally coherent state is a faithful reflection of a globally consistent reality. Therefore, an operation's linearization against this state correctly places it on the global timeline relative to the prior event that established that coherent state.

Our full proof demonstrates the validity of this argument by applying this two-case logic to an exhaustive analysis of every required ordering pair for each protocol variant we consider, confirming the theorem holds universally.

To illustrate our proof's structure, let's consider a pair of operations from the litmus test in Section 4.1. Let's show the GPU thread's operations (an `Acquire` on `flag` followed by a `Load` on `data`) linearize in order. First we consider the local linearization events $e_{\mathtt{acq\_lin}}$ and $e_{\mathtt{load\_lin}}$ of the `Acquire` $e_{\mathtt{acq}}$ and `Load` $e_{\mathtt{load}}$ respectively:

$e_{\mathtt{acq\_lin}}$ either linearizes at the directory, or cache. In the case where $e_{\mathtt{acq\_lin}}$ linearizes in cache, it has coherent permissions. With coherent permissions, there is no consistency ordering to guarantee, as no other cache has updated the entry. So we are left with the case where $e_{\mathtt{acq\_lin}}$ is at the directory. Similarly in our GPU's RCC protocol, $e_{\mathtt{load\_lin}}$ will linearize at the directory.

Now that we know $e_{\mathtt{acq\_lin}}$ and $e_{\mathtt{load\_lin}}$ linearize at the directory, we show they are ordered ($e_{\mathtt{acq\_lin}} \prec_{OB} e_{\mathtt{load\_lin}}$). By Def 5.4, we know that $e_{\mathtt{acq}} \supset e_{\mathtt{acq\_lin}}$ (case 2 in Def 5.4). Then, again by Def 5.4, we see that either (1) $e_{\mathtt{load\_lin}} \prec_{OB} e_{\mathtt{load}}$ (case 4 in Def 5.4), (2) $e_{\mathtt{load}} \supset e_{\mathtt{load\_lin}}$ (case 2 in Def 5.4), or (3) $e_{\mathtt{load}} \prec_{OB} e_{\mathtt{load\_lin}}$ (case 3 in Def 5.4).

$e_{\texttt{load\_lin}}$'s case (2) and (3) are trivial. Because $e_{\texttt{acq}} \prec_{OB} e_{\texttt{load}}$, and $e_{\texttt{acq}} \supset e_{\texttt{acq\_lin}}$, we know transitively $e_{\texttt{acq\_lin}} \prec_{OB} e_{\texttt{load\_lin}}$. Then by Axiom `To-global-shim`, we know the local linearization events encapsulate global linearization events. Transitively, the encapsulated global linearization events are still ordered.

We proceed to prove case (1) by using Axiom `Acquire` and `Inv.-Read`. By Axiom `Acquire`, because we have an `Acquire` operation $e_{\texttt{acquire}}$ we know: $e_{\texttt{acq}} \supset e_{\texttt{inval}}$ and $e_{\texttt{acq\_lin}} \prec_{OB} e_{\texttt{inval}}$. By Axiom `Inv.-Read` we know from $e_{\texttt{inval}} \prec_{OB} e_{\texttt{load}}$ that $e_{\texttt{inval}} \prec_{OB} e_{\texttt{load\_lin}}$. Then again by transitivity, because $e_{\texttt{acq\_lin}} \prec_{OB} e_{\texttt{inval}}$ and $e_{\texttt{inval}} \prec_{OB} e_{\texttt{load\_lin}}$ we know $e_{\texttt{acq\_lin}} \prec_{OB} e_{\texttt{load\_lin}}$. Again, by Axiom `To-global-shim`, the local linearization events result in ordered global linearization events.

The rest of the proof proceeds with the same style of reasoning.

6.5 Mechanization in Lean

To provide the highest degree of confidence in our formal results, we have mechanized our entire framework—including the abstraction, the formal system model, and the proof of the main theorem—in the Lean 4 theorem prover. This was a non-trivial effort, spanning approximately 26,000 lines of code. The mechanization ensures our results are not only rigorously checked but also reproducible.

The definitions and theorems in this paper correspond directly to their counterparts in our formal development. The final statement of our main theorem, which proves that the required linearization order holds for any two events that have a required ordering, is shown below:

```
theorem CompoundProtocol.enforce_compound_consistency
  {b : Behaviour n} {e₁ e₂ : Event n} {init : InitialSystemState n}
  (cmp : CompoundProtocol n) (hsame_protocol : e₁.sameProtocol n e₂)
  (he₁_not_downgrade : ¬e₁.down) (he₂_not_downgrade : ¬e₂.down)
  (he₁_cache : e₁.isCacheEvent) (he₂_cache : e₂.isCacheEvent)
  (he₁_in_b : e₁ ∈ b) (he₂_in_b : e₂ ∈ b) (hsame_cache : e₁.cid = e₂.cid)
  (hdiff_addr : e₁.addr ≠ e₂.addr)
  : e₁.OrderedBefore n e₂ → cmp.CompoundLinearizationOrder n b init e₁ e₂
```

This theorem states that for any two events, `e1` and `e2`, in an execution history `b`, if a set of preconditions hold, then their temporal ordering (`e1.OrderedBefore`) implies that their formal linearization order (`CompoundLinearizationOrder`) is also correct. The block of hypotheses (`hsame_protocol`, `hdiff_addr`, etc.) constrains the theorem to a key scenario: two distinct cache events within the same cluster. This represents one of the core proof obligations: showing that local orderings within a single device are correctly preserved in the global system.

The process of rigorous mechanization yielded valuable lessons. We found that defining properties declaratively (by their invariants) was far more robust for proof development than using an operational or algorithmic style. Defining a concept like the "latest state" by its required properties, rather than by a complex algorithm to compute it, proved more modular and resilient to changes in other parts of the proof. Finally, we found that maintaining a high-level, on-paper proof sketch was invaluable. When engaged in proving hundreds of smaller, interdependent lemmas, it is easy to lose track of the overall argument. The paper proof served as a crucial roadmap to structure the formal development and track our progress towards the main theorem.

7 Applying the Framework: Case Studies

To demonstrate the practicality of our framework, we modeled and successfully verified two new protocols against these axioms, in addition to the MSI and RCC protocols which we used as running examples. While each case study individually verifies one protocol class (CXL for SWMR; RCC-O

for RC), the heterogeneous composition CXL+RCC-O is formally covered by construction through Theorem 6.1. The theorem is universally quantified over all combinations of protocol instances satisfying our abstraction axioms. Therefore, proving CXL and RCC-O each satisfy the axioms independently is sufficient to guarantee correctness of their heterogeneous composition. We do not require a separate, monolithic, heterogeneous verification step because compositional reasoning eliminates that need—that is precisely the payoff of our framework.

Our main theorem is proven in Lean, which provides the necessary generalized mathematical reasoning. The case studies below use Murphi to verify that CXL and RCC-O satisfy the abstraction axioms. We manually ensure that the concrete properties checked in Murphi strictly mirror the axioms assumed in Lean. While this manual translation extends the trusted computing base, it represents a tooling gap rather than a conceptual flaw: the axioms are mathematically defined and their verification reduces to a standard model-checking problem. Bridging this gap via a certified translation is reserved for future work.

Industrial Protocol (CXL). We modeled a representative subset of the CXL specification and successfully verified it against the SWMR-related axioms. CXL is important because it is one of the candidates for a suitable global protocol for tying different clusters together.

CXL specifies a MESI protocol variant that guarantees the SWMR invariant between writeable states Modified and Exclusive, and shared readable state Shared. We choose a subset of the CXL protocol family (CXL.cache) that provides a strongly ordered write and read, and forwarded requests.

- **RdOwn:** request coherent read-write permissions in Exclusive or Modified state.
- **RdShared:** request coherent read Shared permissions.
- **SnpData:** forwarded RdShared downgrade to the owner.
- **SnpInvM:** forwarded RdOwn downgrade to the owner.
- **SnpInvS:** forwarded RdOwn downgrade to sharers.

We map the key requests of our CXL subset to our abstraction. A `RdOwn` request, which seeks exclusive read-write permissions, is modeled as a **Coherent SC Write**. A `RdShared` request, which seeks shared read-only permissions, is modeled as a **Coherent SC Read**. Consequently, the concrete CXL states map directly to our abstract states: the `Exclusive` and `Modified` states map to **Write-Coherent**, while the `Shared` state maps to **Read-Coherent**. Note that CXL implements a MESI protocol, and allows a cache to silently upgrade from E to M state. We capture CXL's silent cache upgrades as they maintain the same *abstract* permissions (e.g., E and M are both `Write-Coherent`). Our abstraction captures all transient states of the protocol during a request *implicitly* within the start and end of the corresponding event. Therefore, any concurrency bug related to them (e.g., a race condition in the protocol) would necessarily manifest as a violation of our axioms, as the bug would produce an incorrect final state or violate a required temporal ordering.

We verify in the Murphi model checker [16] that our CXL subset passes all well-formed sanity checks, and enforces SWMR. Lastly, we verify our CXL shim translations are fit to use CXL as a cluster *and* global protocol. We verify directory accesses produce correct global requests, global downgrades reduce CXL cluster protocol state accordingly, and produce correct global to cluster requests. Model checking CXL's 14 axioms took between 1.36s to 5.34s to explore 63,354 to 190,062 states, with symmetry reduction reducing the number of explored states. We verified our CXL model using a configuration of three caches and a single memory address. A single-address model is sufficient for verifying SWMR protocols [29]. Since protocol logic is decoupled and operates independently on memory addresses, a correctness proof for one address generalizes to all addresses.

Advanced RCC Protocol (RCC-O). Next, we model an advanced RCC variant in which writes can obtain exclusive ownership, a feature of the academic Denovo protocol [12]. This case study

demonstrates our framework's flexibility in handling protocols that blend SWMR and baseline RCC-like features.

As opposed to the baseline RCC protocol, RCC-O utilizes an exclusive ownership (O) state that indicates it has the sole copy of a cache line. If a line is in O state, Acquire requests can enjoy the optimization of not invalidating cache entries in Valid state. This is possible because O state indicates no other cache has written a new value to the cache line. Additionally, we again capture transient states implicitly between the start and end of request events.

For verification, we map RCC-O's requests to our abstract requests:

- **Release** $\triangleq$ Coherent Release Write
- **Write** $\triangleq$ Coherent Weak Write
- **Acquire** $\triangleq$ Non-Coherent Acquire Read
- **Read** $\triangleq$ Non-Coherent Weak Read

We verify RCC-O cache and directory events are well-formed. From our mapping, RCC-O uses `Write-Coherent` and `Read-Non-Coherent` states from its write and read requests respectively. As such, we verify RCC-O enforces SW instead of SWMR. As RCC-O uses an acquire request, we verify it accesses the directory on non-O state, followed by invalidating any Read-Non-Coherent cache entries. Lastly, we verify our shims correctly translate local directory accesses to global cache requests, and downgrades reduce protocol state, ensuring SW (Single-Writer). Model checking RCC-O's 14 axioms took between 9 to 720 seconds to explore between 651,504 to 32,239,191 states, with state symmetry reduction reducing the number of searched states. We verified our RCC-O model using a configuration of two caches and two memory addresses. Unlike SWMR protocols, consistency-directed protocols like RCC-O are not address-independent; verifying at least two addresses is therefore necessary to cover these cross-address orderings.

These case studies show our axioms are general. They model real, industrial-strength and advanced academic protocols, and guarantee their correct composition (Theorem 6.1).

8 Related Work

Our work builds upon and contributes to two primary areas of research: the formal verification of coherence protocols and the design of heterogeneous computing systems.

8.1 Coherence Protocol Verification

There is a rich, decades-long history of applying formal methods to verify cache coherence protocols [10, 17, 32, 35, 45, 46]. These efforts have largely followed two paths: automated techniques like model checking [21, 26, 40, 51, 52] and interactive theorem proving [27, 28, 34, 47]. While foundational, these works have overwhelmingly focused on verifying single, monolithic protocols.

This tradition of mechanized proof is best exemplified by frameworks like Kami [14], which provides a platform in the Coq for the end-to-end specification, implementation, and modular verification of a specific, complex hardware design. Our work shares this commitment to mechanized proof. However, our focus is subtly different. Where Kami provides a powerful framework to prove the correctness of a specific implementation, we additionally provide a general, compositional theorem about the rules of interaction between protocols that satisfy our abstract axioms.

Almost all of these works focused on verifying CPU protocols that enforce SWMR. One exception is `CCICheck` [24] that introduced a novel coherence protocol abstraction for handling eager SWMR as well as non-SWMR lazy coherence protocols, but relied on extensive litmus testing and focused on one protocol, not hierarchy or heterogeneity.

More recent work has explored compositional verification of *homogeneous* systems, where multiple identical protocol clusters are connected [11, 13, 25, 47]. Hemiola [13] proposed a correct-by-construction approach to hierarchical protocol composition. Its key contribution is guaranteeing by construction that any protocol defined in the DSL satisfies serializability. Our work differs by tackling the more complex challenge of **heterogeneous composition**: proving the correctness of fusing distinct protocol families.

8.2 Heterogeneous Computing and Coherence

Several key academic and industrial efforts have addressed the challenge of heterogeneous shared memory. The Heterogeneous System Architecture (HSA) [20] initiative was an early industrial effort to define a specification for unified memory [6, 18, 19, 36, 37, 39]. Indeed, systems like AMD's APUs were among the first to implement a fused CPU-GPU design using a Synchronous Propagation-like rule without a formal proof linking implementation to specification [9].

In academia, Spandex [7] proposed a unified hardware-software interface connecting hardware mechanisms to language-level DRF models, but its mechanisms were not formally verified and it did not address hierarchical global interconnects. HieraGen and HeteroGen pioneered the automated synthesis of heterogeneous coherence protocols via Synchronous Propagation [31, 32]: HieraGen combines multiple SWMR cluster protocols with a global SWMR protocol, while HeteroGen extended this to non-SWMR cluster protocols but did not model a global protocol. More recently, C3 [23] and vCXLGen [22] support heterogeneous cluster protocols like HeteroGen while also modeling an SWMR subset of CXL as the global protocol, again using Synchronous Propagation. MemGlue [15] proposed a new global protocol designed to enforce C11, verified with litmus testing and a manual pen-and-paper proof; however, the correctness of its shim translations has neither been generalized nor verified. All of these efforts rely on litmus testing and model checking—methods bounded to finite configurations that cannot provide a formal correctness guarantee.

Our work is complementary. While prior efforts have proposed interfaces, synthesized implementations, and used testing to validate them, our paper provides the first formal, machine-checked proof that a practical, compositional implementation strategy (Synchronous Propagation) correctly enforces a formal specification (Compound Consistency) in a realistic, hierarchical system model. We do not propose a new design pattern; rather, we provide the missing formal assurance for the de facto design pattern used today.

9 Conclusion

The history of shared-memory multiprocessors has been one of specifications struggling to catch up with complex implementations. It took decades of research, extensive empirical study, and formal analysis to precisely define the consistency models of processors that were already in wide use. Today, history is repeating itself in the heterogeneous era. Processors that fuse CPUs and GPUs are now commonplace, exposing unified shared memory through standards like CXL and CHI, but they lack a rigorous, formal understanding of their end-to-end consistency guarantees.

This paper provides that formal foundation. We introduced a new abstraction for coherence protocols and used it to deliver the first machine-checked proof that the widely-used Principle of Synchronous Propagation is correct when used with an SWMR interconnect. Our work provides immediate assurance for a central design pattern in modern heterogeneous design. In doing so, it also establishes a powerful, compositional verification methodology: architects can now verify their components locally with a formal guarantee that the composed system is correct. This replaces intuition with proof, enabling correct-by-construction design of future heterogeneous processors.

Acknowledgments

This work is supported in part by the National Science Foundation under grant CCF-2525271 and ARM. We thank Rohaan Almeida for work on the implementation of the M1 case study, and Cees de Laat for generously allowing us to use his M1 hardware for it. Generative AI tools (Claude, Anthropic) were used to assist with copyediting the text and with developing portions of the mechanized Lean proofs.

References

[1] [n. d.]. AMBA Specifications. https://www.arm.com/architecture/system-architectures/amba/amba-specifications. [Accessed 03-07-2025].
[2] [n. d.]. Compute Express Link. https://computeexpresslink.org/. [Accessed 03-07-2025].
[3] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 8:1–8:54. doi:10.1145/3458926
[4] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running tests against hardware. *Lecture Notes in Computer Science* 6605 LNCS (2011). doi:10.1007/978-3-642-19835-9_5
[5] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats. *ACM TOPLAS* 36, 2 (jul 2014), 1–74. doi:10.1145/2627752
[6] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. 2016. Lazy Release Consistency for GPUs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 26, 13 pages. doi:10.1109/MICRO.2016.7783729
[7] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. 2018. Spandex: A flexible interface for efficient heterogeneous coherence. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 261–274. doi:10.1109/ISCA.2018.00031
[8] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT Press.
[9] Bradford M Beckmann and Anthony Gutierrez. 2015. The AMD gem5 APU simulator: Modeling heterogeneous systems in gem5. In *Tutorial at the International Symposium on Microarchitecture (MICRO)*.
[10] Gregor Bochmann and Carl Sunshine. 2003. Formal methods in communication protocol design. *IEEE transactions on Communications* 28, 4 (2003), 624–631. doi:10.1109/TCOM.1980.1094685
[11] Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan, and Ching-Tsun Chou. 2010. Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. *Formal Methods in System Design* 36, 1 (2010), 37–64. doi:10.1007/s10703-010-0092-y
[12] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*. doi:10.1109/PACT.2011.21
[13] Joonwon Choi, Adam Chlipala, and Arvind. 2022. Hemiola: A DSL and verification tools to guide design and proof of hierarchical cache-coherence protocols. In *International Conference on Computer Aided Verification*. Springer, 317–339. doi:10.1007/978-3-031-13188-2_16
[14] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.* 1, ICFP (2017), 24:1–24:30. doi:10.1145/3110268
[15] Rachel Cleaveland and Caroline Tripperl. 2024. Memory Consistency Model-Aware Cache Coherence for Heterogeneous Hardware. In *Proceedings of Formal Methods in Computer-Aided Design*. doi:10.34727/2024/isbn.978-3-85448-065-5_22
[16] David L Dill. 1996. The Murphi Verification System. In *CAV*, Vol. 1102. doi:10.1007/3-540-61474-5_86
[17] 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 (2023), 1145–1168. doi:10.1145/3591267
[18] Joel Hestness, Stephen W Keckler, and David A Wood. 2015. GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU-GPU processors. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 87–97. doi:10.1109/IISWC.2015.15
[19] Derek R Hower, Blake A Hechtman, Bradford M Beckmann, Benedict R Gaster, Mark D Hill, Steven K Reinhardt, and David A Wood. 2014. Heterogeneous-race-free memory models. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 427–440. doi:10.1145/2541940.2541981
[20] HSA Foundation. 2012. Heterogeneous System Architecture: A Technical Review.
[21] C Norris Ip and David L Dill. 1996. Verifying systems with replicated components in Murϕ. In *International Conference on Computer Aided Verification*. Springer, 147–158. doi:10.1007/3-540-61474-5_65

[22] Anatole Lefort, Julian Pritzi, David Carpentieri, Nicolòand Schall, Simon Dittrich, Soham Chakraborty, Nicolai Oswald, and Pramod Bhatotia. 2026. vCXLGen: Automated Synthesis and Verification of CXL Bridges for Heterogeneous Architectures. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2026, Pittsburgh, PA, USA, March 22-26, 2026*, Benjamin C. Lee, Harry Xu, Mark Silberstein, and Bingyao Li (Eds.). ACM, 2177–2196. doi:10.1145/3779212.3790245

[23] Anatole Lefort, David Schall, Nicolò Carpentieri, Julian Pritzi, Soham Chakraborty, Nicolai Oswald, and Pramod Bhatotia. 2026. C^3: CXL Coherence Controllers for Heterogeneous Architectures. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2026, Sydney, Australia, January 31 - Feb. 4, 2026*. IEEE, 1–17. doi:10.1109/HPCA68181.2026.11408469

[24] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2015. CCICheck: using μhb graphs to verify the coherence-consistency interface. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, Milos Prvulovic (Ed.). ACM, 26–37. doi:10.1145/2830772.2830782

[25] Kenneth McMillan. 2016. Modular specification and verification of a cache-coherent interface. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 109–116. doi:10.1109/FMCAD.2016.7886668

[26] KL McMillan and J Schwalbe. 1990. Formal verification of the Gigamax Cache Consistency Protocol. *manuscript, Sept* 27 (1990).

[27] Kenneth L. McMillan. 2001. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK, September 4-7, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2144)*, Tiziana Margaria and Thomas F. Melham (Eds.). Springer, 179–195. doi:10.1007/3-540-44798-9_17

[28] J Strother Moore. 1998. An acl2 proof of write invalidate cache coherence. In *International Conference on Computer Aided Verification*. Springer, 29–38. doi:10.1007/BFb0028728

[29] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence* (2 ed.). Morgan & Claypool Publishers. doi:10.1007/978-3-031-01764-3

[30] NVIDIA Corporation 2024. *NVIDIA GH200 Grace Hopper Superchip Architecture Whitepaper*. NVIDIA Corporation. v1.21.

[31] Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. 2020. HieraGen: Automated Generation of Concurrent, Hierarchical Cache Coherence Protocols. In *Proceedings of the 47th Annual International Symposium on Computer Architecture*. doi:10.1109/ISCA45697.2020.00077

[32] Nicolai Oswald, Vijay Nagarajan, Daniel J. Sorin, Vasilis Gavrielatos, Theo Olausson, and Reece Carr. 2022. HeteroGen: Automatic Synthesis of Heterogeneous Cache Coherence Protocols. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 756–771. doi:10.1109/HPCA53966.2022.00061

[33] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: X86-TSO. In *Lecture Notes in Computer Science*, Vol. 5674 LNCS. doi:10.1007/978-3-642-03359-9_27

[34] 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*. 288–296. doi:10.1145/237502.237573

[35] Fong Pong and Michel Dubois. 1995. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems* 6, 8 (1995), 773–787. doi:10.1109/71.406955

[36] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 457–467. doi:10.1145/2540708.2540747

[37] Jason Power, Joel Hestness, Marc S Orr, Mark D Hill, and David A Wood. 2014. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters* 14, 1 (2014), 34–36. doi:10.1109/LCA.2014.2299539

[38] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. doi:10.1145/3158107

[39] Phil Rogers. 2013. Heterogeneous system architecture overview. In *2013 IEEE Hot Chips 25 Symposium (HCS)*. 1–41. doi:10.1109/HOTCHIPS.2013.7478286

[40] Divjyot Sethi, Muralidhar Talupur, and Sharad Malik. 2014. Using flow specifications of parameterized cache coherence protocols for verifying deadlock freedom. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 330–347. doi:10.1007/978-3-319-11936-6_24

[41] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. doi:10.1145/1785414.1785443

[42] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th International Symposium on Microarchitecture*. 647–659.

doi:10.1145/2830772.2830821

[43] I. Singh, A. Shriraman, W.W.L. Fung, M. O'Connor, and T.M. Aamodt. 2013. Cache Coherence for GPU Architectures. In *HPCA*. doi:10.1109/HPCA.2013.6522351

[44] Alan Smith, Gabriel H Loh, Michael J Schulte, Mike Ignatowski, Samuel Naffziger, Mike Mantor, Mark Fowler, Nathan Kalyanasundharam, Vamsi Alla, Nicholas Malaya, et al. 2024. Realizing the amd exascale heterogeneous processor vision. In *ISCA'24: Proceedings of the 51st Annual International Symposium on Computer Architecture*. Association for Computing Machinery. doi:10.1109/ISCA59077.2024.00068

[45] Daniel J Sorin, Manoj Plakal, Anne E Condon, Mark D Hill, Milo MK Martin, and David A Wood. 2002. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE transactions on parallel and distributed systems* 13, 6 (2002), 556–578. doi:10.1109/TPDS.2002.1011412

[46] Ulrich Stern and David L Dill. 1995. Automatic verification of the SCI cache coherence protocol. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 21–34. doi:10.1007/3-540-60385-9_2

[47] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. 2015. Modular deductive verification of multiprocessor hardware designs. In *International Conference on Computer Aided Verification*. Springer, 109–127. doi:10.1007/978-3-319-21668-3_7

[48] Felix Werner, Marcel Weisgut, and Tilmann Rabl. 2025. Towards Memory Disaggregation via NVLink C2C: Benchmarking CPU-Requested GPU Memory Access. In *Proceedings of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems*. 8–14. doi:10.1145/3723851.3723853

[49] An Qi Zhang, Andrés Goens, Daniel Sorin, and Vijay Nagarajan. 2026. compositional-protocol-proof. https://github.com/userAZ/compositional-protocol-proof. doi:10.5281/zenodo.20408667

[50] An Qi Zhang, Andrés Goens, Daniel Sorin, and Vijay Nagarajan. 2026. A Formally Verified Foundation for Compositional Heterogeneous Coherence. Zenodo. doi:10.5281/zenodo.19087136

[51] Meng Zhang, Jesse D Bingham, John Erickson, and Daniel J Sorin. 2014. PVCoherence: Designing flat coherence protocols for scalable verification. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 392–403. doi:10.1109/HPCA.2014.6835949

[52] 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

Received 2025-11-14; accepted 2026-04-03