RESEARCH-ARTICLE

# Compositional Model-Driven Verification of Weakly Consistent Distributed Systems

**BRYANT J CURTO**, Northeastern University, Boston, MA, United States

**JEONGHYEON KIM**, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

**ALAN WANG**, Northeastern University, Boston, MA, United States

**GIJUNG IM**, Yonsei University, Seoul, South Korea

**JIEUNG KIM**, Yonsei University, Seoul, South Korea

**JEEHOON KANG**

**View all**



**Published:** 13 October 2025



SOSP '25: ACM SIGOPS 31st Symposium on Operating Systems Principles
*October 13 - 16, 2025*
*Seoul, Republic of Korea*

**Conference Sponsors:**
SIGOPS


https://doi.org/10.1145/3764860.3768331
ISBN: 9798400722257

# Compositional Model-Driven Verification of Weakly Consistent Distributed Systems

Bryant J. Curto
curto.b@northeastern.edu
Northeastern University
Boston, USA

Jeonghyeon Kim
jeonghyeon.kim@kaist.ac.kr
KAIST
Daejeon, Republic of Korea

Alan Wang
wang.alan@northeastern.edu
Northeastern University
Boston, USA

Gijung Im
kijeonglim@yonsei.ac.kr
Yonsei University
Seoul, Republic of Korea

Jieung Kim
jieungkim@yonsei.ac.kr
Yonsei University
Seoul, Republic of Korea

Jeehoon Kang
jeehoon.kang@furiosa.ai
FuriosaAI
Seoul, Republic of Korea

Ji-Yong Shin
j.shin@northeastern.edu
Northeastern University
Boston, USA

## Abstract

Despite abundant distributed system verification work, weakly consistent distributed systems have been overlooked as formal verification targets. Verification methodologies starting from the code level face scalability challenges when verifying weakly consistent distributed systems as these systems employ a wide variety of similar semantics and designs, potentially leading to redundant verification work.

This paper presents our ongoing work to develop Moveri, a top-down verification framework for weakly consistent distributed systems. It aims to reduce the effort needed to verify this commonly overlooked class of safety-critical systems. Moveri is based on novel compositional and operational models of sixteen different consistency semantics including eventual consistency, four session guarantees, and causal consistency. The implementation-agnostic semantic models connect to templated distributed protocol models and further to different verified implementations through refinement.

Verification of the safety of weakly consistent distributed system protocols and implementations is made more practical by the flexibility of our templated distributed protocol models. Further, the compositionality of these semantic models and templating of these protocol models enable the framework to efficiently scale to support a range of weak consistency semantics. These claims are evaluated through verification of primary-replica and gossip style protocol implementations configurable to any of six different consistency semantics.

***CCS Concepts:*** • **Software and its engineering → Distributed programming languages**; **Software verification**; • **Theory of computation → Distributed computing models**; *Object oriented constructs.*

***Keywords:*** consistency semantics, formal verification, refinement, bisimulation, Rocq, Coq



*PLOS '25, October 13–16, 2025, Seoul, Republic of Korea*

ACM ISBN 979-8-4007-2225-7/25/10
https://doi.org/10.1145/3764860.3768331

## 1 Introduction

Ensuring the correctness of safety-critical systems is burdensome, and this challenge is exacerbated in distributed systems due to their inherent complexity. As alternatives to debugging and testing, formal verification approaches [20, 22, 30, 33, 35, 37, 47] have been proposed to mathematically guarantee the correctness of distributed systems [29]. The proposed approaches leverage modularity for proofs, separate functional correctness and protocol safety proofs, gradually relax network assumptions, and extend logic for concurrency reasoning for distributed systems. As a demonstration of these methodologies, strongly consistent distributed systems implementing linearizability [11, 16, 28, 32, 44, 49] are used as common targets.

In this paper, we focus on verifying weakly consistent distributed systems that are more widely deployed than strongly consistent distributed systems but are rarely studied for formal verification. There is a wide spectrum of weakly consistent distributed systems, which employ different semantics suited to their application needs. While these systems provide relaxed guarantees, ensuring their correctness is nonetheless crucial.

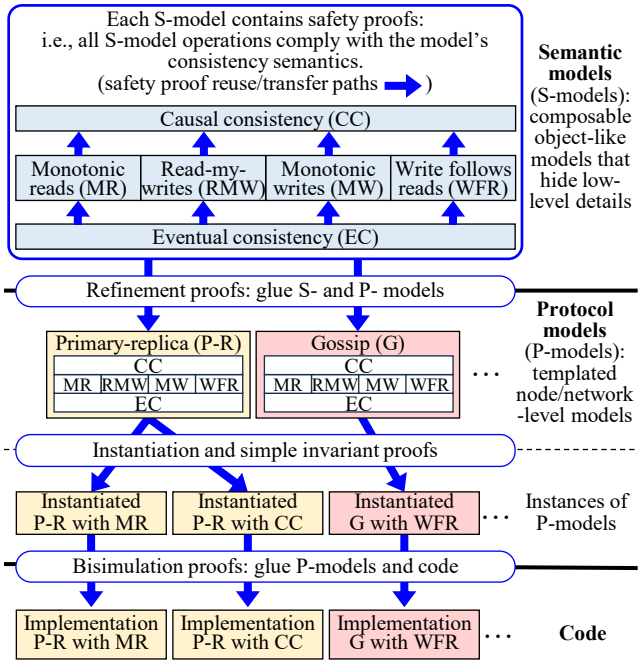


**Figure 1.** Overview of Moveri.

Applying formal verification to this diverse domain can lead to a significant amount of redundant work. The problem arises because most existing approaches focus on the code-level details; this is natural as the ultimate goal of verification is to guarantee the correctness of the implementation. This bottom-up approach, however, leads to a proof structure that closely mirrors the code, and despite modular designs, the proof becomes difficult to port and reuse even for different systems with similar semantics and properties [21, 26]. Consequently, when verifying different systems that exhibit similar or the same characteristics, users must thoroughly understand the theory and reasoning behind how each system works, write a huge amount of verification code to model each system, and prove desired properties for each system.

A top-down verification approach, starting from an implementation-agnostic model capturing the distributed system semantics (*e.g.*, Figure 2), can make the verification more scalable [23, 30]. However, weakly consistent distributed semantics are diverse and their models have not been extensively studied. Further, operational models are favorable for formal verification because they capture code-like transitions, but most distributed system semantics are defined using axiomatic definitions [5, 40, 45]. These axiomatic definitions can be used as correctness properties which a system must satisfy, but they may not clearly define how a system satisfies these properties. The operational models of the semantics can simplify verifying the properties that are not apparent in the code because finding the matching behavior in the code and the operational model (*e.g.*, through refinement) can lead to verifying the properties [22].

To fill in these gaps, we propose Moveri, a compositional model-driven weakly consistent distributed system verification framework (Figure 1). *Semantic models* (S-model) of distributed systems are at the core of our verification. S-models are operational models that capture the safety properties of weakly consistent distributed systems. S-models capture the semantics, or safety-preserving behaviors, and minimal states of distributed systems, while abstracting away low-level implementation details such as network messages. The S-models are proven to enforce the axiomatic definitions and can be used to verify different system designs implementing the same semantics. Although a model-based verification was explored for linearizable semantics [23, 33] and causal consistency [30] for the safety of a single semantics or protocol, the novelty of our S-models comes from their ability to capture a wide variety of distributed weak consistency semantics via composition and templating. These factors play a critical role in modular verification and proof reuse.

Each S-model independently models a distributed system and semantics and carries correctness proofs, and the S-models compose to model a system with stronger semantics. In this paper, we focus on six popular semantics: eventual consistency (EC) [46], the four session guarantees (SG) [40]—monotonic reads (MR), read-my-writes (RMW), monotonic writes (MW), and write-follows-reads (WFR)—and causal consistency (CC) [5]. The SG S-models are built on top of EC S-model, and CC S-model is designed as a composition of the four SG S-models. Properties verified for each S-model are passed along to the composed model for free and S-models can expand to different combinations of SGs, at no cost.

To verify the distributed system code, we connect the S-model to more concrete protocol-level models (P-models) through refinements. P-models include node and network-level details. Similar to the composable S-model design, the P-model employs a templated design: depending on the semantics it observes, P-models are assembled to match the S-model and the code-level implementation. We instantiate the P-models to primary-replica-style [38, 41] and gossip-style protocols [15]. Safety properties of P-models are proved by reusing the proofs of S-models through refinement.

We use Rocq (formerly Coq) [43] to implement Moveri. Distributed system implementations in Go connect to Moveri using Goose [4, 12], and we verify the primary-replica and gossip style implementations that can be configured to follow different consistency semantics.

We make the following contributions:

**(1)** novel, verified, compositional, operational models for distributed weak consistency semantics with which top-down verification is performed;
**(2)** safety proof reuse techniques exploiting the compositionality of S-models;

**(3)** an implementation of Moveri which, through templating and composition, aims to support the formal verification of a variety of distributed systems each enforcing any of a variety of weak consistency semantics;
**(4)** verified distributed system code implementing EC, four SGs, and CC using Moveri.

## 2 Background and Motivation

### 2.1 Weakly Consistent Distributed Semantics

Distributed weak consistency semantics define how updates propagate and become visible in a distributed system. They allow the system to trade off consistency by returning old data for better performance [14, 17, 31, 41]. Our target weak consistency semantics are eventual consistency, four session guarantees, and causal consistency, which are widely used in industry and academia [1, 3, 13, 36, 41, 42, 45, 48].

Most such systems assume eventual consistency, which guarantees minimal safety properties [7, 39, 46].

- **Eventual consistency (EC):** A user's read returns any version of the data written by prior writes.

Session guarantees (SGs) can be layered on top of EC and they order reads and writes differently per user depending on the individual user's interactions with the system [40–42].

- **Monotonic reads (MR):** A read returns the same or a later version of data than the user last read.
- **Read-my-writes (RMW):** A read returns the same or a later version of data than the user last wrote.
- **Monotonic writes (MW):** A write by a user applies to a node in the system after all prior writes by the same user are applied to the node.
- **Write-follows-reads (WFR):** A write by a user applies to a node in the system after all prior writes read by the same user are applied to the node.

Finally, causal consistency, one of the strongest among weak consistency semantics, enforces causality-preserving ordering [5, 27, 30] of operations across the entire system.

- **Causal consistency (CC):** All users observe causally related operations in a single, common order. That is, if a user observes an operation, then it must also observe all causally dependent operations.

Interestingly, the composition of MR, RMW, MW, and WFR is known to be CC [9]. Leveraging this theory and the practice that EC is the base assumption for most systems, we design S-models that are compositional, by which we achieve reuse.

### 2.2 Motivation and Related Work

Table 1 summarizes efforts to verify executable distributed system code. Chapar [30] and ADO [23] propose abstract models similar to Moveri to verify the safety of causally consistent distributed systems and strongly consistent distributed systems, respectively, in a top-down manner.

| Project | Code | Safety | Model | Model Comp. | Verified Semantics |
|---|---|---|---|---|---|
| Moveri (this work) | Go | ✔ | ✔ | ✔ | 16 weak (EC, SG, CC, etc.) |
| Chapar [30] | OCaml | ✔ | ✔ | ✘ | CC |
| ADO [23] | C | ✔ | ✔ | ✘ | Strong |
| LiDO [33] | OCaml | ✔ | ✔ | ✘ | Byz/Strong |
| Verdi [47] | OCaml | ✔ | ✘ | N/A | Strong |
| IronFleet [22] | C# | ✔ | ✘ | N/A | Strong |
| Grove [35] | Go | ✔ | ✘ | N/A | Strong |
| WormSpace [37] | C | ✔ | ✘ | N/A | Strong |

**Table 1.** Comparison of Moveri and other works. Moveri is the only framework that uses model composition and verifies multiple weak consistency semantics.

LiDO [33] extends ADO to verify the safety and liveness of Byzantine consensus protocols. Each work targets different semantics but their commonality is being able to use their models in verifying a few different implementations satisfying the semantics. However, their models are limited to only one semantics and the scope of applications is limited.

Verdi [47] does not employ models of distributed semantics but uses different network models to simplify the verification. IronFleet [22], Grove [35], and WormSpace [37] introduce methodologies to modularize proofs and reasoning which Moveri also pursues. They verify distributed consensus protocols to demonstrate their generality and applicability. However, when it comes to verifying a wide spectrum of distributed systems, they do not provide concrete reusable proofs like the ones available in model-driven approaches.

Moveri uses model-driven verification for proof reuse (Sections 4.2 and 5.3) and realizes wide applicability at the same time by designing a model composition for a variety of distributed semantics (Section 4.1).

Other than the work summarized in Table 1, EC is widely studied and verified in the context of Conflict-free Replicated Data Types (CRDT) [34]. Bouajjani et al. [8] proposed formal specifications for model checking an optimistic replication system under EC for CRDT. Gomes et al. [18] developed a verification framework to check CRDT algorithms maintain strong eventual consistency. While they analyze and specify EC as a phenomenon under CRDTs, we propose S-models for EC itself and beyond that connect to implementations.

## 3 Overview

As described in Section 1 and Figure 1, Moveri consists of multiple layers. The key difference from other layer-based approaches [20, 22, 23, 37] is that we fill in the layers with composable and templated models reusable for verification.

***Layered Design*** Each S-model represents a distributed consistency semantics. S-models encapsulate the distributed system state as an object and the transitions as mutations to the object. Individual S-models carry the safety properties and proofs which can be reused to prove properties of composed S-models and verify the P-models and the code. We assume that all of our target semantics are built on top of

EC. The SG S-models are realized on top of the EC S-model, and the CC S-model is composed of four SG S-models.

P-models reflect how distributed protocols implement the distributed consistency semantics by concretizing how data gets propagated on an asynchronous network. P-models are modularly templatized such that they can be instantiated to a protocol that follows any semantics in the S-model layer and that is close to the implementation. The safety properties of the P-models are verified through refinement of corresponding S-models.

Users are responsible for implementing and connecting the code with the target P-model. We verify Go implementations of primary-replica style [38, 41] and gossip style [15] protocols with our target semantics. The code is verified through refinement of P-models and, transitively, S-models, thereby reusing the safety proofs of the S-models.

***Assumptions*** The network is assumed unreliable: it can reorder, drop, and duplicate messages. We assume a client can interact with any server in the system, unlike the common CC implementations and verifications which pin the client to a dedicated server [19, 30, 31]. We use this more generic assumption, because the semantics do not restrict client-to-server mappings and clients often talk to multiple servers in the same replicated system in practice [24]. To extend the semantics across different servers, we assume nodes reliably store their state despite a crash and eventually come back up with the same state.

## 4 S-model Layer

An S-model abstractly captures minimum distributed system states and actions – common to all such distributed systems – for replicating a generic object and enforcing a consistency semantics. Details, like an asynchronous network, are abstracted away. It is represented as an object with simple atomic *read* and *write* APIs where non-determinism is abstracted and captured within API calls.

Weak consistency semantics can be partially ordered by their strength [45]. This fact is leveraged to enable S-models to be designed compositionally. Composition enables code and proof reuse; this is vital for any framework that aims to scalably verify a variety of correctness conditions.

### 4.1 From Eventual to Causal S-models and Beyond

***Eventual Consistency (EC) S-model*** The EC S-model is the base model for all S-models. All S-models maintain an object with some *read* and *write* interface. The EC S-model captures how *read* and *write* operations are performed while updates are replicated across nodes in the system.

Figure 2a illustrates an EC S-model instantiated with an object that contains a register and counter, and allowing sets and gets to the prior and increments and reads of the latter. It consists of *op-set*, the set of all update operations added to the system, and *vis-sets*, visible subsets of *op-set*. *Vis-sets* are

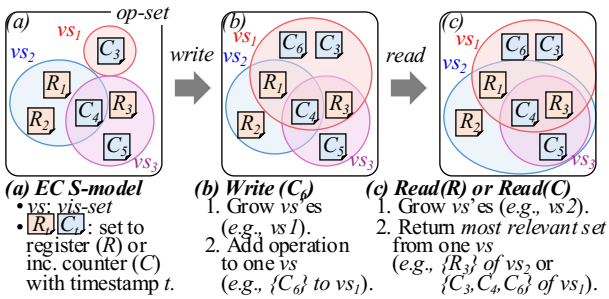


**Figure 2.** S-model for eventual consistency.

analogous to individual nodes in the system such that, if all updates in the *vis-set* are applied, the result is the data state of a node (*i.e.*, a register value and counter value).

*Write* adds a new operation with a new timestamp to *op-set* and at least one *vis-set* (Figure 2b). Updates are ordered using timestamps and a timestamp ordering relation. The ordering relation is a strict partial order, allowing for commutative writes (*i.e.*, register set and counter increment) to be unordered while non-commutative writes (*i.e.*, register sets) are ordered. EC permits updates to be applied at a replica node regardless of timestamp. However, if non-commutative updates are applied, their in-order evaluation is enforced to ensure convergence. To reflect asynchronous update propagation within a distributed system, *vis-sets* non-deterministically grow by adding more operations from the *op-set* when processing each *write* and *read*.

*Read* chooses one of the *vis-sets* and returns the *most relevant set* [40] of operations for the *read* in the *vis-set* (Figure 2c). The *most relevant set* consists of minimal operations responsible for the *read*. For example, the *most relevant set* for a read of the register is the latest register set $R_t$ in the *vis-sets* but no counter increments $C_t$. The *most relevant set* for a read of a counter is all the counter increments $C_t$ in the *vis-set* but no register sets $R_t$. The *most relevant set* allows S-models to model the semantics independent of the replicated object.

***Session Guarantees (SG) S-models*** Session guarantees (SGs)—Monotonic reads (MR), read-my-writes (RMW), monotonic writes (MW), and writes-follow-reads (WFR)—are defined based on individual user sessions; a user interacts with the system and thereby creates dependencies for that user's future interactions. Specifically, for each SG, the dependencies are either the user's previously read or written updates, which restrict either the user's future *read* or *write* calls. Each SG is independently and widely used in industry and academia [1, 3, 13, 36, 41, 42, 48].

All SG S-models are built on the EC S-model which has been augmented to become the dependent EC module that selectively enforces given dependencies to the API calls and with a client that keeps track of its own dependencies (Figure 3). Each SG S-model uses this augmented EC S-model to restrict what dependencies a client should track and which

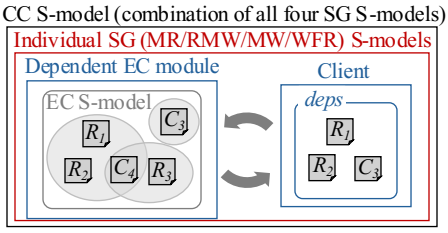


**Figure 3.** Composed S-models for session guarantees and causal consistency.

API calls should enforce the dependencies: *e.g.*, MR S-model restricts the client to track reads and enforces these dependencies only for *read*, whereas MW S-model restricts the client to track writes and enforces them only for *write*.

The client model represents the user session and remembers the dependencies in a dependency set (*deps*). The client interacts with the dependent EC module by calling the *read* and *write* with dependencies. Asynchrony between the client and the system is modeled in SG S-models through nondeterministic failures of the API calls. The failure represents timeouts due to message drops or extended delays. The failed operations may or may not be processed by the dependent EC module and any results are ignored by clients.

The dependent EC module sits atop the EC S-model and expands the *read* and *write* APIs to take user dependencies and ensures that they are satisfied. *Read* and *Write* are serviced using a *vis-set* of the EC S-model that satisfies all dependencies. This also applies during the growing of *vis-sets*.

All of our SG S-models are designed with generality and modularity in mind. The dependent EC module and the client model are reused across the four different S-models.

***Causal Consistency (CC) S-model*** The definition of causal consistency (CC) commonly relies on Lamport's happens before relations, but enforcing the four SG at the same time also leads to CC [9]. That is, CC can be defined based on SGs: a client sees the same or later version of data than what it last read and wrote, and the client's write applies after all of its prior writes and the writes that it has seen.

We leverage this definition by composing the four SG S-models to create the CC S-model (Figure 3). While each SG S-model keeps track of either read or write dependencies and enforces the dependencies during either *read* or *write*, the CC S-model collects both read and write dependencies in the client and passes them to both *read* and *write*. Our modular and compositional CC S-model simplifies reasoning about CC and differs from prior work that only models CC [19, 30].

***Other Semantics*** Similar to how CC S-model is achieved, Moveri can trivially model any semantics that combines SGs (*i.e.*, powerset of {MR, RMW, MW, WFR}; $2^4 = 16$ cases); PRAM consistency is one of them (*i.e.*, {MR, RMW, MW}) [10].

### 4.2 S-model Safety Proofs

To ensure that our operational S-models comply with existing axiomatic definitions [45], we prove them safe against these definitions. Existing definitions restrictively define reads and writes as directly reading or writing an object/memory-address and not on a set of updates (*i.e.*, *vis-sets* in our S-models). Thus, we expand the definitions to account for this change.

Our modular and compositional S-models allow for proof reuse (Figure 1). The proofs for the SG S-models reuse those on the dependent EC module, client model, and the EC S-model. Further, the proofs for the CC S-model also reuse the proofs of individual SG S-models and the EC S-model. Such reuse allowed us to reduce the proof efforts especially for CC compared to prior work that directly verifies CC [19, 30].

## 5 P-model Layer

The P-model layer concretely captures distributed system states and actions for replicating a generic object and enforcing a consistency semantics. While capturing more detail than the implementation-agnostic S-models, P-models template states and functions which can be flexibly instantiated to a range of protocols. The node states and functionality, network message contents, and network assumptions are all templated to allow for a wide variety of instantiations. Still, for any instantiation that preserves the template properties, safety is verified through refinement. We exercise their flexibility by instantiating the P-models to two widely used replication protocol styles: primary-replica and gossip.

### 5.1 P-model Components

The P-model includes individual server and client nodes which interact over the network following a templated protocol and semantics.

***Server Nodes*** Each server node implementing a replication protocol has three main roles: maintaining the node's copy of the replicated object, propagating updates to other nodes, and communicating with clients. Servers are divided into three corresponding templated pieces: object manager (OM), propagation manager (PM), and client manager (CM).

The OM is in charge of maintaining the node's own copy of the replicated object. This entails judging when and how requests should be performed on the state to enforce the consistency semantics (*e.g.*, deferring reads to enforce MR or writes to enforce MW) and actually executing the requests. The PM ensures that server nodes in the system learn of writes to achieve convergence. It propagates new writes that are accepted by the OM, and depending on the protocol it may also forward the writes from other server nodes to others. Finally, the CM interacts with clients.

Figure 4 presents simplified pseudo-code capturing how the server node executes using the managers. On the server node receiving input, the managers are informed of this

```
1 function repl_node_step(Managers mgs, Message in)
2     mgs := inform(mgs, in)
3     outs := []
4     cont := true
5     while cont
6         mgs, outs', cont := act(mgs)
7         outs := append(outs, outs')
8     return mgs, outs
```

**Figure 4.** Simplified pseudo-code defining the behavior of the templated server node model in the P-model layer.

input (line 2). The OM is then allowed to iteratively perform actions (lines 5-7). An action is performing a read or write request, or applying propagated updates to the replicated state. For some input, zero or more actions can be performed; the OM defers acting on input in order to enforce consistency semantics (*e.g.*, waiting for dependencies to arrive before performing a read under MR). During each call to `act()`, the PM and CM can generate messages to send based on the action performed.

Types and functions `Managers`, `Message`, `inform()`, and `act()` are templated such that they can be instantiated to achieve a range of protocols. To model primary-replica, the servers are instantiated such that only the OM of the primary server receives clients' write requests, and the propagation managers of replica servers do not propagate data. Our gossip model lifts these restrictions and treats all nodes equally: all nodes can receive client requests and propagate data.

***Client Node*** The client includes a session manager which is the only templated piece in the client. The session manager decides communication patterns (*e.g.*, whether to retry after timeout), keeps track of session state (*e.g.*, dependencies), and adds any additional metadata (*e.g.*, request ids and dependencies) to the API calls.

### 5.2 P-model Template Properties

The P-models templated design lowers the proof burden required to verify a new protocol, so long as there exists corresponding template arguments such that the instantiated P-models express the protocol. Verification of the uninstantiated templated models is performed only once. Thus, users verifying a new protocol need only verify that their instantiated P-models satisfy a small set of properties. These template properties are defined for each consistency and relative to each node's manager(s).

For example, to ensure that the safety property of EC is enforced, the result of a read request produced by the OM must be the expected value. Formally:

**Lemma 5.1** (EC - Read Expected Value)**.** *Assume that there exists a set of operations (each a write and timestamp pair) representing all of the unique writes applied to the replica node's copy of the replicated object. If the OM produces a value to fulfill a read request, then this value must be the result of evaluating the read request on the object produced by evaluating the set of operations (recursively on some initial object value in timestamp order).*

### 5.3 Connecting P-models to S-models

Assuming the set of template properties associated with a given consistency, the templated models in the P-model layer are verified by proving that each refines the corresponding model of that consistency in the S-model layer. For example, assuming the template properties of EC, we prove that the distributed system model in the P-model layer, formed from a collection of replica nodes, refines the EC S-model. Similarly, assuming the template properties of MR, we prove that the corresponding P-model refines the dependent EC module.

In the same way that some models of the S-model are formed from a composition of others, so too are the proofs of refinement from the P-models to the S-models. For example, the previously described proof of refinement with the dependent EC module reuses the proof of refinement with the EC S-model. This is possible since the dependent EC module is a composition containing the EC S-model. Further, the template properties of the prior imply those of the latter. Therefore, completing this proof requires only proving refinement with the other components of the dependent EC module (*e.g.*, client).

The key novelty compared to other work using refinement proofs is that we reuse multiple pre-built components in a compositional way to maximize the proof reuse. The top-down approach from the S-model simplifies P-model proofs.

## 6 Code layer

Using Moveri, we verify distributed systems that implement EC, four SGs, and CC under primary-replica style and gossip style protocols. We implement them in Go and use Goose [4, 12] to connect the code to Moveri which is based on Rocq.

### 6.1 Go Implementation

The primary-replica and gossip style implementations follow a design that combines Bayou [42] and COPS [31]. It uses lists of committed and tentative updates like Bayou, and inserts updates that propagate late into the lists. Ordering of operations and dependency tracking are simplified by using the COPS approach of using vector timestamps and ensuring ordered delivery of updates from the same source node.

For the primary-replica style protocol, all write requests go to one primary node and read requests can be serviced by any node. The primary is in charge of propagating all updates. Note that this implementation is not strongly consistent: clients can see different versions of outdated data by reading from replica nodes. For the gossip style protocol, all nodes service reads and writes and propagate updates. The client session state is maintained at the client program and clients

can communicate with any node unlike COPS. Due to the dependency tracking using vector timestamps, clients only need to maintain $8n$ bytes for dependencies where $n$ is the number of server nodes.

The verified code consists of functions (and types) which, given the current node state and a received message, return the updated node state and messages to send. We glue this code to a trusted shim networking layer, which manages calling into these verified functions, serializes and deserializes messages, and interacts with the OS. Our consolidated implementation is configurable to use one of the communication patterns (*i.e.*, primary-replica or gossip) and consistency semantics (*i.e.*, EC, SG, or CC). Besides the glue code, our trusted computing base includes Rocq, Goose, Go runtime, OS, and hardware.

### 6.2 Connecting Code to P-models

Using Goose, our Go implementation translates to Perennial [12] that extends Iris [2, 25] which is implemented in Rocq. We write a functional specification of our Go implementation in Rocq and prove bisimulation with the translated code. The specification is then connected to the instantiated P-models using refinement. The template of the P-model layer enables us to obtain an instantiation almost identical to the functional specification.

Once connected to the P-models, the code is proven to transitively connect to the S-models through refinement, guaranteeing the safety of the code.

## 7 Discussion and Future Work

Correctness of a system decomposes into a conjunction of a safety property (nothing "bad" happens) and a liveness property (something "good" eventually happens) [6]. This paper has spoken solely about safety properties. We are in the process of enabling Moveri with the ability to verify liveness properties of code following a similar approach to that used for safety properties. Besides LiDO [33] and IronFleet [22], prior works in Table 1 have not demonstrated support for the verification of liveness properties. Further, because of their focus on strong consistency, prior works have not fully explored the liveness properties specific to weakly consistent distributed systems.

The focus of Moveri has been on verifying EC, SG, and CC for good reason. Weak consistency semantics are partially ordered [45]. Our compositional approach to building S-models necessitates creating new S-models from weaker to stronger. Having started from EC (a very weak consistency semantics), future work can and will expand Moveri to support any of a wide variety of stronger consistency semantics not addressed by prior works.

## 8 Summary

In this paper, we presented Moveri, a compositional model-driven verification framework for weakly consistent distributed systems. Moveri employs a top-down verification approach where operational models of distributed system semantics embed safety proofs. The models capture 16 different semantics including eventual consistency, session guarantees, and causal consistency. The semantic models are designed to compose with each other to create stronger semantics and reduce proof burdens. In addition to templating, these characteristics of the framework enable it to scale to support the verification of a variety of systems enforcing a variety of consistency semantics. Distributed system protocols and implementations can reuse the proofs to verify their correctness through refinement. Using Moveri, we verify Go implementations of primary-replica style and gossip style systems exhibiting six different semantics.

## Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This work is supported by NSF awards #2019285 and #2442888. Further, it is supported by the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant and the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2024-00459026 and RS-2025-00556905, respectively).

## References

[1] 2024. Azure Cosmos DB. https://azure.microsoft.com/en-us/products/cosmos-db/.

[2] 2024. Iris Project. https://iris-project.org.

[3] 2024. Sequential consistency without borders: how D1 implements global read replication. https://blog.cloudflare.com/d1-read-replication-beta/.

[4] 2025. Goose: a subset of Go with a semantics in Coq. https://github.com/goose-lang/goose.

[5] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distrib. Comput.* 9, 1 (March 1995), 37–49. https://doi.org/10.1007/BF01784241

[6] Bowen Alpern and Fred B Schneider. 1985. Defining liveness. *Information processing letters* 21, 4 (1985), 181–185.

[7] Peter Bailis and Ali Ghodsi. 2013. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM* 56, 5 (May 2013), 55–63. https://doi.org/10.1145/2447976.2447992

[8] Ahmed Bouajjani, Constantin Enea, and Jad Hamza. 2014. Verifying eventual consistency of optimistic replication systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 285–296. https://doi.org/10.1145/2535838.2535877

[9] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. 2004. From session causality to causal consistency. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004. Proceedings.* 152–158. https://doi.org/10.1109/EMPDP.2004.1271440

[10] Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. 2004. Session Guarantees to Achieve PRAM Consistency of Replicated Shared Objects. In *Parallel Processing and Applied Mathematics*, Roman

Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–8.

[11] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, Berkeley, CA, USA, 173–186. http://dl.acm.org/citation.cfm?id=296806.296824

[12] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 243–258. https://doi.org/10.1145/3341301.3359632

[13] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.* 1, 2 (aug 2008), 1277–1288. https://doi.org/10.14778/1454159.1454167

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (oct 2007), 205–220. https://doi.org/10.1145/1323293.1294281

[15] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, British Columbia, Canada) *(PODC '87)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/41840.41841

[16] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Financial Cryptography and Data Security*, Ittay Eyal and Juan Garay (Eds.). Springer International Publishing, Cham, 296–315.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. *SIGOPS Oper. Syst. Rev.* 37, 5 (oct 2003), 29–43. https://doi.org/10.1145/1165389.945450

[18] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 109 (oct 2017), 28 pages. https://doi.org/10.1145/3133933

[19] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic. *Proc. ACM Program. Lang.* 5, POPL, Article 42 (jan 2021), 29 pages. https://doi.org/10.1145/3434323

[20] Jeremiah Griffin, Mohsen Lesani, Narges Shadab, and Xizhe Yin. 2020. TLC: temporal logic of distributed components. *Proc. ACM Program. Lang.* 4, ICFP, Article 123 (aug 2020), 30 pages. https://doi.org/10.1145/3409005

[21] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 653–669.

[22] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. ACM, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

[23] Wolf Honoré, Jieung Kim, Ji-Yong Shin, and Zhong Shao. 2021. Much ADO about Failures: A Fault-Aware Model for Compositional Verification of Strongly Consistent Distributed Systems. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 97 (oct 2021), 31 pages. https://doi.org/10.1145/3485474

[24] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 167–181. https://doi.org/10.1145/2517349.2522722

[25] Ralf Jung, David Swasey, Filip Sieczkowski, Ksper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*. 637–650.

[26] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[27] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[28] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229

[29] Butler Lampson. 2021. Hints and Principles for Computer System Design. arXiv:2011.02455 [cs.DC]

[30] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 357–370. https://doi.org/10.1145/2837614.2837622

[31] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

[32] Diego Ongaro and John K Ousterhout. 2014. In Search of an Understandable Consensus Algorithm.. In *USENIX Annual Technical Conference*. 305–319.

[33] Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, Wolf Honoré, and Zhong Shao. 2024. LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs. In *Proceedings of the 45th ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Copenhagen, Denmark) *(PLDI 2024)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3519939.3523444

[34] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

[35] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: a Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles* (, Koblenz, Germany,) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 113–129. https://doi.org/10.1145/3600006.3613172

[36] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, Jakub Szefer, and Hakim Weatherspoon. 2016. Towards Weakly Consistent Local Storage Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 294–306. https://doi.org/10.1145/2987550.2987579

[37] Ji-Yong Shin, Jieung Kim, Wolf Honoré, Hernán Vanzetto, Srihari Radhakrishnan, Mahesh Balakrishnan, and Zhong Shao. 2019. WormSpace: A Modular Foundation for Simple, Verifiable Distributed Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. ACM, New York, NY, USA, 299–311. https://doi.org/10.1145/3357223.3362739

[38] Andrew S. Tanenbaum and Maarten van Steen. 2006. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc.

[39] Doug Terry. 2013. Replicated Data Consistency Explained through Baseball. *Commun. ACM* 56, 12 (dec 2013), 82–89. https://doi.org/10.1145/2500500

[40] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. 1994. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. 140–149. https://doi.org/10.1109/PDIS.1994.331722

[41] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 309–324. https://doi.org/10.1145/2517349.2522731

[42] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.* 29, 5 (Dec. 1995), 172–182. https://doi.org/10.1145/224057.224070

[43] The Rocq development team. 2025. The Rocq prover. https://rocq-prover.org.

[44] Robbert Van Renesse and Fred B Schneider. 2004. Chain Replication for Supporting High Throughput and Availability.. In *OSDI*, Vol. 4. 91–104.

[45] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1, Article 19 (jun 2016), 34 pages. https://doi.org/10.1145/2926965

[46] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (jan 2009), 40–44. https://doi.org/10.1145/1435417.1435432

[47] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958

[48] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. 2018. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 401–412. https://doi.org/10.1109/ICDE.2018.00044

[49] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591