Hybrid Verification of Consensus-Based Distributed Services

JAEHYUNG LEE, Seoul National University, Korea
TAEYOUNG YOON, Seoul National University, Korea
YUJIN IM, Seoul National University, Korea
TAEYOUNG RHEE, Seoul National University, Korea
SANGHYUN YI, Seoul National University, Korea
JIEUNG KIM, Yonsei University, Korea
CHUNG-KIL HUR, Seoul National University, Korea

Consensus-backed distributed services call for hybrid verification: formally verifying the safety-critical service while preserving the surrounding application code as executable code for testing, model checking, or further verification. Existing verification efforts for protocols such as Paxos typically bring the protocol, the surrounding service, and application-facing code into the same formal verification framework. Traditional contextual refinement captures a basic form of hybrid verification, but it cannot express the cross-boundary obligations needed when verified and unverified components share resources and rely on assumptions about each other. This is too rigid for realistic distributed systems.

This paper presents the first hybrid verification of a distributed write-once key-value store built on Multi-Paxos over an unreliable network, using CRIS's conditional contextual refinement and imaginary specifications. The implementation combines Paxos, the Write Once Register (WOR) storage layer, shared memory and network infrastructure, application-provided Paxos callbacks, and arbitrary application code. We verify the Paxos/WOR stack, the callbacks, and callback registration discipline while leaving the remaining applications concrete. The resulting refinement replaces Paxos-owned resources, callbacks, and distributed WOR implementations by a centralized atomic WOR model with nondeterministic failure, while preserving ordinary application memory and network behavior in an executable top-level model. In this hybrid verification setting, prophecy reasoning connects the atomic outcomes of the abstract WOR model to future consensus decisions, and helping reasoning accounts for acceptor-side work performed for proposer operations. The resulting executable abstract model gives a compact testing target for unverified applications; in our bank application experiments, it exposes injected bugs within 0.2 seconds, while the full implementation times out in the tested configurations.

1 Introduction

Distributed systems are critical yet challenging to build correctly, as their reliability directly impacts the services built on top of them. These systems are the backbone of modern databases and cloud infrastructure by enabling multiple nodes to coordinate and cooperate despite node failures, unstable network connections, or message reordering. At the core of many such services is a consensus protocol: a protocol that prevents distributed nodes from deciding conflicting values while they try to agree on a collection of operations despite failures and network nondeterminism. Therefore, bugs or safety violations in distributed system implementations can trigger cascading failures—conflicting data across servers, data loss, or complete system unavailability—compromising not just individual components but the integrity of entire systems that depend on them.

Despite their critical role, traditional testing cannot adequately ensure correctness of distributed systems, motivating the need for formal verification. The vast state space created by concurrent node interactions and nondeterministic message delivery makes exhaustive testing infeasible, and formal verification has emerged as a principled response. A substantial body of work now exists

Authors' Contact Information: Jaehyung Lee, Seoul National University, Korea, jaehyung.lee@sf.snu.ac.kr; Taeyoung Yoon, Seoul National University, Korea, taeyoung.yoon@sf.snu.ac.kr; Yujin Im, Seoul National University, Korea, yujin.im@sf.snu.ac.kr; Taeyoung Rhee, Seoul National University, Korea, taeyoung.rhee@sf.snu.ac.kr; Sanghyun Yi, Seoul National University, Korea, sanghyun.yi@sf.snu.ac.kr; Jieung Kim, Yonsei University, Korea, jieungkim@yonsei.ac.kr; Chung-Kil Hur, Seoul National University, Korea, gil.hur@sf.snu.ac.kr.

in this area. Iris-based efforts such as Grove [22] have verified Paxos-based replicated storage systems; CCAL-based efforts such as ADO [8] give a unified abstraction model for various strongly consistent consensus algorithms; and earlier systems like IronFleet [7] and Verdi [27] produced end-to-end verified implementations of consensus protocols and replicated state machines.

Despite their diversity, these efforts share a methodological limitation: the components inside the verified system are all brought into the same formal verification framework, whether Iris-based separation logic [11], CCAL-style certified abstraction layers [6], or another proof infrastructure. This is often stronger than necessary. Even when reusable consensus abstractions are available, they are typically reusable only as proof components for applications verified in the same framework, rather than as executable abstractions for arbitrary concrete application code that can be validated separately by different methods. In practice, the components of a distributed system differ widely in which verification method serves them best. A consensus protocol at the core may warrant the full cost of formal proof, while the surrounding application code, often far larger in volume, is better served by lighter techniques such as testing or static analysis.

We therefore seek hybrid verification: formally verify the safety-critical distributed service, but leave the surrounding application code available for lighter-weight validation such as testing. The desired theorem is not merely that the verified component satisfies an internal invariant. Rather, it should justify replacing the concrete distributed stack by an executable top-level abstraction, so that tests run against the abstraction are faithful to the original implementation.

The basic idea of hybrid verification already appears in contextual refinement, where a verified component can be linked with an unverified context and then replaced by the abstract model that serves as its specification. Traditional contextual refinement, however, is unconditional: the component must refine its specification for any surrounding context. This is too rigid for realistic distributed systems, where the verified subsystem and the surrounding application often share resources and impose assumptions on each other across boundaries that are not clean library-call interfaces.

CCR [24] and its successor CRIS [13] overcome this limitation through conditional contextual refinement, which allows the unverified context and verified component to be mutually dependent through separation-logic-style assumptions. CRIS provides the state-of-the-art framework for hybrid verification. Its imaginary specifications can mix executable code with ownership assertions, allowing a specification to serve two roles at once: it can provide separation-logic-style reasoning principles for verified components, while also remaining executable for testing, model checking, or further verification.

Using ConCRIS [29], an extension of CRIS supporting interleaving execution, this paper realizes this vision for realistic consensus-backed distributed services. Realistic implementations introduce complex interactions among the consensus protocol, the storage layer built on top of it, shared memory and network infrastructure, application-provided callbacks, and ordinary application code that should remain executable. Handling these interactions in a hybrid verification setting is the central technical challenge. We first describe the concrete system and its top-level abstract model, and then present the key innovations needed to address this challenge.

To make these interactions concrete, we verify a realistic Multi-Paxos-based WOR (Write Once Register) key-value service and prove an executable top-level abstraction suitable for application testing. The implementation architecture is shown in Fig. 1. It consists of n proposer nodes, m acceptor nodes, and r client nodes connected by an unreliable network model that may drop, duplicate, and reorder messages. Each proposer node contains arbitrary application code, application-provided Paxos callbacks, a WOR library, a Paxos proposer library, and local memory. Each acceptor node contains arbitrary application code, application-provided callbacks, a Paxos acceptor library, and local memory. Within each proposer or acceptor node, the application may itself run multiple threads

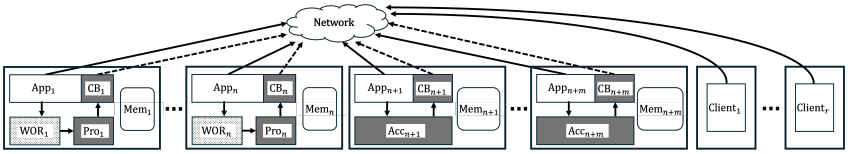

Fig. 1. Architecture of the whole-system implementation. Gray boxes are Paxos-facing implementation components, and dotted boxes are WOR libraries. Solid links indicate call directions, including network calls; memory calls are omitted. Dotted links are network calls used by the Paxos protocol.

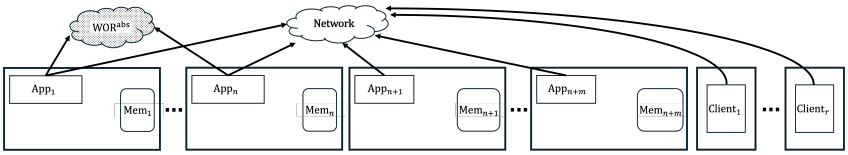

Fig. 2. Architecture of the whole-system abstract model. The dotted WOR^{abs} cloud is a single global atomic WOR module that replaces the dotted and gray components and dotted network communication from Fig. 1.

that share the node-local memory; in this work we model that shared memory as sequentially consistent.[1] The client nodes do not participate in Paxos communication; they interact with the applications running on proposer and acceptor nodes through ordinary non-Paxos communication. WOR is the storage layer exposed to applications: it provides a key-value interface in which each slot may be written at most once, using Multi-Paxos internally to replicate the chosen values. Only applications on proposer nodes use the WOR library directly, but applications may run on both proposer and acceptor nodes and may communicate with each other and with client nodes over sockets unrelated to Paxos.

The verification result is an end-to-end refinement from this distributed implementation to a much simpler executable abstraction. The top-level abstract model is shown in Fig. 2. In this model, the ordinary application code remains on each proposer and acceptor node, and the client nodes remain unchanged, so both applications and clients can still be tested. In particular, the abstract model preserves the sequentially consistent shared-memory behavior used by multithreaded applications inside each node. The Paxos callbacks, Paxos proposer and acceptor libraries, and node-local WOR implementations are abstracted away into the single global module WOR^{abs}. This atomic module manipulates a mathematical key-value map in a few lines of code, while still allowing nondeterministic operation failure to model message loss and failure to reach consensus. The abstract model also preserves the parts of memory and network behavior used by applications and clients outside Paxos: allocation becomes nondeterministic to account for memory blocks removed with the Paxos implementation, and the network continues to model non-Paxos sockets operationally. In contrast, use of Paxos-owned sockets by the remaining application code triggers undefined behavior, expressing the required noninterference condition that applications must not interfere with Paxos traffic except through their verified callbacks. Because the client nodes never participate in Paxos communication, they remain completely unconditional context in the verification of Paxos and WOR.

The first innovation is a callback-aware hybrid verification architecture. Industrial consensus libraries often do not own all of their networking code. Instead, they delegate packet encoding, packet transmission, packet reception, and packet decoding to callbacks supplied by the surrounding application. This architecture is practical because application developers know the concrete networking environment, but it creates a verification problem: the correctness of Paxos depends on

[1] ConCRIS [29] also supports weakly consistent memory models; we use sequential consistency here to focus on Paxos/WOR.

code that is not part of the Paxos library, while verifying the whole application would defeat the purpose of hybrid verification. We address this by giving application developers only small proof obligations for the Paxos-facing callbacks and registration discipline. The rest of the application remains arbitrary code. ConCRIS then lets us compose these callback proofs with the Paxos proof, so that the callbacks and the Paxos library are abstracted together into a Paxos specification.

The second innovation is selective abstraction of shared infrastructure. The memory and network modules are used both by the verified Paxos/WOR stack and by arbitrary application code. A purely ownership-based specification would be useful for verifying Paxos, but would impose unnecessary proof obligations on application networking and application memory accesses. A purely operational specification would remain executable, but would not provide the ownership reasoning needed for Paxos and WOR. Building on CRIS's hybrid memory specification [13], we use hybrid imaginary specifications for memory and network to support this selective abstraction. For Paxos-owned resources, these specifications expose pre- and postcondition-style ownership assertions. For application-owned resources, they preserve the operational model. This lets the proof abstract away Paxos packets, Paxos sockets, and Paxos memory blocks while keeping unrelated application packets and memory blocks in the final testing model.

The third innovation is the refinement of the distributed WOR library to a centralized atomic store. The implementation reaches agreement through future Paxos executions over an unreliable network; the abstract WOR module, however, must decide atomically at the moment an operation is invoked whether the operation succeeds, fails, or observes an already-written value. We bridge this temporal mismatch using prophecy reasoning. A prophecy variable is a logical device that records, at an early point in the proof, a value that will be justified later by the actual execution. In our setting, it lets the abstract WOR operation choose its atomic outcome immediately, while the proof later validates that choice when the corresponding Paxos consensus outcome is eventually observed.

We also use helping reasoning to bridge a spatial mismatch between logical operations and physical execution. The logical WOR operation belongs to the proposer-side application that invoked `read` or `write`, but part of the physical work that makes the operation succeed is performed by acceptor nodes. Helping lets those acceptor-side steps be accounted for as progress toward the proposer-side logical operation. Together, these established reasoning principles let us prove, in a hybrid verification setting, that the Multi-Paxos-based WOR implementation refines the centralized atomic WOR abstraction.

The resulting abstract model is not merely a logical specification; it is a better target for testing the unverified application code and its clients. Testing the full implementation must explore message drops, duplications, reorderings, repeated receipt of old Paxos packets, and races between nodes attempting to reach consensus. Testing the abstraction removes this internal Paxos nondeterminism and exposes a compact sequential interface for instantiated applications and their clients. In our experiments on a replicated bank application, the abstract model found injected application bugs within 0.2 seconds, while the full implementation timed out in 10 minutes without finding the same bugs in the tested configurations.

In summary, this paper makes the following contributions.

- We present the first hybrid verification of a distributed write-once key-value store built on Multi-Paxos over an unreliable network, with an end-to-end theorem parameterized by arbitrary multithreaded applications that share node-local sequentially consistent memory.
- We support a callback-based Paxos architecture in which only the application-supplied callbacks and registration discipline are verified, while the rest of the application remains concrete code.

```
Module Proposer.
  fn init(cb_send: ptr, cb_recv: ptr)
  fn prepare(k: key): quorum_state
  fn commit(k: key, v: value)
Module Acceptor.
  fn init(cb_send: ptr, cb_recv: ptr)
  fn run_once()
```

Fig. 3. Interfaces for Paxos

```
Module WOR.
  fn init(cb_send: ptr, cb_recv: ptr)
  fn read(k: key): result
  fn write(k: key, v: value)
```

Fig. 4. Interfaces for WOR

- We introduce hybrid imaginary specifications for memory and network that provide ownership reasoning for Paxos resources while preserving operational behavior for application resources.
- We verify that the Paxos-based WOR implementation refines a centralized atomic WOR model with nondeterministic failure, adapting prophecy and helping reasoning to relate distributed executions to atomic operations in the hybrid verification setting.
- We show that the resulting abstraction substantially improves application testing by preserving application behavior while removing the Paxos-specific nondeterminism that dominates randomized testing of the full implementation.

2 Background

2.1 Multi-Paxos

Multi-Paxos [15, 26] is a consensus protocol that lets distributed nodes agree on values despite unreliable communication. In this paper, the protocol is used as a collection of independent per-key consensus instances: for each string key, Paxos may decide at most one string value. Different keys do not have a Paxos-level ordering constraint. Higher layers may encode version numbers or log positions as keys, but the Paxos library itself only provides consensus separately for each key.

The protocol has two roles. Proposers try to establish a value for a key, and acceptors maintain the persistent protocol state that makes agreement possible. A proposer interacts with acceptors through two phases. In the prepare phase, it asks enough acceptors for the current state of the key. If it cannot obtain responses from a quorum, the operation may fail. If it obtains a quorum, the replies summarize what the proposer is allowed to do next: it may be free to propose any value, it may be required to preserve a value already observed by the protocol, or it may learn that consensus has already been made on a value. In the commit phase, the proposer submits an allowed value to the acceptors. If this commit is accepted by a quorum, that value is chosen as the consensus value for the key.

The rest of the paper relies only on this abstract structure. The Paxos proof hides the internal bookkeeping of Paxos, the acceptor-local state, and the protocol messages behind the proposer interface described below. Its client-facing safety fact is agreement: even with message loss, duplication, reordering, and competing proposers, two successful consensus observations for the same key cannot disagree. We verify safety, not liveness, so an operation may fail to obtain a quorum and an execution may fail forever.

2.2 Paxos and WOR Libraries

Fig. 3 shows the Paxos library interface. The `Proposer` module is used on proposer nodes, as shown in Fig. 1. Its `init` function installs two application-provided callbacks: one for sending Paxos messages and one for receiving them. The library deliberately delegates packet formatting and socket use to these callbacks, matching an implementation pattern in which the networking environment is owned by the embedding application rather than by the consensus library itself.

The two protocol operations are `prepare` and `commit`. The `prepare(k)` operation runs the prepare phase for key `k` and returns a quorum state. The result is `Fail` if no quorum was reached; `AnyC` if the caller may commit any value; `OnlyC(v)` if a following commit must preserve `v`; and `Cons(v)` if consensus has already been made on `v`. The `commit(k,v)` operation then runs the commit phase for an allowed value `v`; if enough acceptors accept the request, the value becomes the consensus value for `k`.

The `Acceptor` module is used on acceptor nodes, as shown in Fig. 1. After initialization, each call to `run_once` handles one incoming Paxos request if one is available. Acceptor-side effects are observed by applications only through later proposer operations.

Fig. 4 shows the WOR library. WOR stands for Write Once Register. It provides a distributed key-value storage interface in which keys and values are strings, and each key can be fixed at most once. A later write to an already fixed key does not update the store; later reads continue to observe the value that already occupies the key. The implementation of WOR uses the Paxos proposer interface to make all proposer nodes agree on the value associated with each key.

The `read(k)` operation returns a result with three cases. `Failed` means the operation did not obtain a conclusive Paxos result within its bounded attempts. `NoValue` means that no value is currently fixed for the key. `Value(v)` means that `v` is the fixed value for `k`. The `write(k,v)` operation similarly runs a bounded Paxos protocol internally, but it does not return a value to the application. It may fail to make progress, it may leave an already fixed key unchanged, or it may help fix a value for `k`. When Paxos requires preserving a value observed during prepare, WOR commits that value rather than the caller's requested value.

The correctness boundary is therefore not just the Paxos algorithm. The Paxos library must implement prepare and commit correctly; the WOR library must use those operations in a Paxos-respecting way while implementing the write-once storage protocol. For example, after `prepare` returns `OnlyC(v)`, WOR must commit `v` rather than an arbitrary new value; otherwise, it could break Paxos safety even if the Paxos primitives themselves are implemented correctly. Finally, the application-provided callbacks must send, receive, encode, and decode Paxos packets consistently. Our verification reflects this boundary: callback implementations and callback registration discipline are verified, while the rest of the application code remains ordinary executable code in the final abstraction.

2.3 CRIS and Imaginary Specifications

CRIS [13] is a framework for proving contextual refinements between executable modules and abstract specifications. In the direction used in this paper, a refinement proves that every behavior of the concrete implementation is represented by the abstract model. Contextual refinement makes this replacement compositional: the component can be linked with surrounding code, and the refinement still relates the linked systems.

Ordinary contextual refinement is unconditional: the component must refine its specification in every context. This is too strong for our Paxos/WOR stack. The Paxos library calls application-provided callbacks, the callbacks and libraries share memory buffers, and the final abstraction assumes that ordinary application code does not directly interfere with Paxos-owned sockets.

CRIS handles such dependencies through *conditional* contextual refinement. A component specification may state assumptions that the surrounding context must satisfy and guarantees that the component provides back. Later, when separately verified components are composed, matching assumption/guarantee pairs are removed by cancellation, leaving the executable abstraction amenable to testing, model checking, or further analysis.

The key CRIS device is an *imaginary specification*: a program-like specification that interleaves ordinary executable commands with proof-only logical commands. CRIS imaginary specifications

$$\frac{\texttt{tgt} \lesssim (st, \textbf{take}(X) \gg\!= K)}{\forall x \in X.\ \texttt{tgt} \lesssim (st, K\,x)}\ \textsc{take-src} \qquad \frac{\texttt{tgt} \lesssim (st, \textbf{choose}(X) \gg\!= K)}{\exists x \in X.\ \texttt{tgt} \lesssim (st, K\,x)}\ \textsc{choose-src}$$

$$\frac{(st, \textbf{take}(X) \gg\!= K) \lesssim \texttt{src}}{\exists x \in X.\ (st, K\,x) \lesssim \texttt{src}}\ \textsc{take-tgt} \qquad \frac{(st, \textbf{choose}(X) \gg\!= K) \lesssim \texttt{src}}{\forall x \in X.\ (st, K\,x) \lesssim \texttt{src}}\ \textsc{choose-tgt}$$

$$\frac{\texttt{tgt} \lesssim (st, \textbf{Assume(}\mathrm{P}\textbf{)}; s)}{P \mathrel{-\!\!*} \texttt{tgt} \lesssim (st, s)}\ \textsc{assume-src} \qquad \frac{\texttt{tgt} \lesssim (st, \textbf{Guaran(}\mathrm{P}\textbf{)}; s)}{P * \texttt{tgt} \lesssim (st, s)}\ \textsc{guarantee-src}$$

$$\frac{(st, \textbf{Assume(}\mathrm{P}\textbf{)}; t) \lesssim \texttt{src}}{P * (st, t) \lesssim \texttt{src}}\ \textsc{assume-tgt} \qquad \frac{(st, \textbf{Guaran(}\mathrm{P}\textbf{)}; t) \lesssim \texttt{src}}{P \mathrel{-\!\!*} (st, t) \lesssim \texttt{src}}\ \textsc{guarantee-tgt}$$

Fig. 5. Selected CRIS simulation rules for the imaginary commands. The target program is on the left of $\lesssim$, and the source/specification program is on the right.

use four logical commands:

$$\textbf{take}(X), \quad \textbf{choose}(X), \quad \textbf{Assume(}\mathrm{P}\textbf{)}, \quad \textbf{Guaran(}\mathrm{P}\textbf{)}.$$

The commands **take** and **choose** are forms of nondeterminism. The commands **Assume(P)** and **Guaran(P)** transfer separation-logic resources at the exact program point where they appear.

Fig. 5 shows the simulation rules for these commands. The target program is on the left of $\lesssim$, and the source program, usually the specification in this paper, is on the right. Thus, when proving that concrete code refines an imaginary specification, the most common case is a concrete target simulated by a source-side imaginary command.

On the source side, **take**(X) is universal: the proof must continue for every $x \in X$. It is used to receive an arbitrary logical value from the context or invariant. For example, the hybrid network specification in §4 uses **take** to name the current logical packet set of a Paxos socket before assuming ownership of it. Source-side **choose**(X) is existential: the proof may pick some $x \in X$. It is used when the abstract model may choose a witness or nondeterministic outcome, such as the result of an abstract receive or the branch taken by the atomic WOR model.

The target-side rules are dual: target-side **take**(X) is existential, so the proof may pick some $x \in X$, while target-side **choose**(X) is universal, so the proof must handle every $x \in X$. These dual rules are especially useful when a later proof uses an already verified library through its imaginary specification: after inlining a call to that library, the imaginary commands appear on the target side of the new simulation. For example, the WOR proof in §7 inlines the Proposer specification from §6, so the target-side rules explain how the proof consumes the **take**, **choose**, **Assume()**, and **Guaran()** commands inside `prepare` and `commit`. The same duality is also used when removing proof-only structure later in the refinement chain.

The rules for **Assume()** and **Guaran()** explain how CRIS embeds separation-logic assertions into code-shaped specifications. In a source specification, **Assume(P)** gives the proof access to the resource P: operationally, the context is responsible for providing P at that point. Conversely, **Guaran(P)** is an obligation for the proof to produce P: operationally, the component guarantees P back to the context. Target-side assumptions and guarantees are dual.

For example, the left-hand definition below calls `foo` and stores its return value in the module-local variable `X`; the right-hand definition is one possible imaginary specification for it:

```
def f() ≡ i ← foo();
          X := i;
```

```
def f() ≡ v ← take(Val);
          Assume(X ↦ v);
          i ← foo();
          Guaran(X ↦ i);
```

In the imaginary specification on the right, the first line captures a logical value `v` supplied by the context. The second line assumes ownership of `X` initially containing this `v`; the surrounding context must provide this ownership at that program point. The third line is ordinary executable code, so the specification obtains the same return value `i` from `foo` as a concrete execution would. The fourth line guarantees ownership of `X` containing `i`. This guarantee may mention `i` because logical commands are interleaved with executable computation; they are not restricted to function entry and exit.

This code-shaped view is the reason CRIS is useful for the hybrid verification in this paper. Specifications can contain executable state updates, such as the final atomic WOR map in §7.1, and also contain proof-only ownership transfers, such as the shaded initialization and call-discipline assertions in Fig. 15. The ordinary commands form the executable skeleton of the abstraction. The logical commands record the obligations needed for the proof and disappear from the final executable model once their matching assumptions and guarantees have been cancelled.

The later sections use this pattern repeatedly. The hybrid memory and network specifications in §4 preserve ordinary executable behavior for application-owned resources, but use **`Assume()`** and **`Guaran()`** to expose ownership reasoning for Paxos-owned memory blocks and sockets. The callback specifications in §5 use the same commands to express that callbacks read and write message buffers and update logical Paxos mailboxes. The Paxos proposer specification in §6 uses logical resources such as PaxosState and QuorumMap to hide low-level Paxos bookkeeping, acceptor-local state, and protocol messages from WOR. Finally, the WOR abstraction in §7 combines an executable key-value map with logical initialization tokens that enforce the application-side calling discipline.

ConCRIS [29] extends CRIS to concurrent and distributed systems by giving an interleaving semantics for scheduled threads or nodes that execute code from modules. Some modules are local to one node, while others, such as network modules, are shared across nodes. The command $\mathcal{Y}$ explicitly yields control to the scheduler; commands between two yields execute atomically with respect to other scheduled threads or nodes. Later figures use this convention directly. When a specification accesses a logical mailbox, Paxos state, or the abstract WOR map with no intervening $\mathcal{Y}$, that access is the logical atomic point of the operation, even though the concrete implementation may take many smaller steps.

ConCRIS also provides the prophecy and helping reasoning principles used later in the Paxos and WOR verification. Prophecy reasoning lets the proof predict a future event and use that prediction to decide where an abstract atomic action should occur. In §6, this is needed because the right atomic point for `prepare` depends on the quorum result that will be observed only later. In §7, prophecy chooses which retry iteration of a WOR read will become the single atomic read of the abstract map. Helping reasoning handles the complementary spatial mismatch: a concrete step performed by one node can complete part of the source-side specification for another node. Paxos uses this when an Acceptor step helps linearize a Proposer operation.

3 Proof Outline and Paper Structure

Fig. 6 gives the proof composition for the full-system refinement. The figure shows one representative proposer node i and one representative acceptor node j; the actual theorem composes the same arguments over all n proposers and all m acceptors. Client nodes are omitted from the figure because they are not involved in the Paxos/WOR verification. They remain unchanged throughout the proof and are handled only as unconditional context around the displayed system.

The bottom row is the concrete implementation, consisting of applications, application-supplied callbacks, Paxos proposer and acceptor libraries, WOR libraries on proposer nodes, node-local memory modules, and the unreliable network. The top row is the executable abstract model: the ordinary applications remain, acceptor-side Paxos work has become Nop, memory and network have

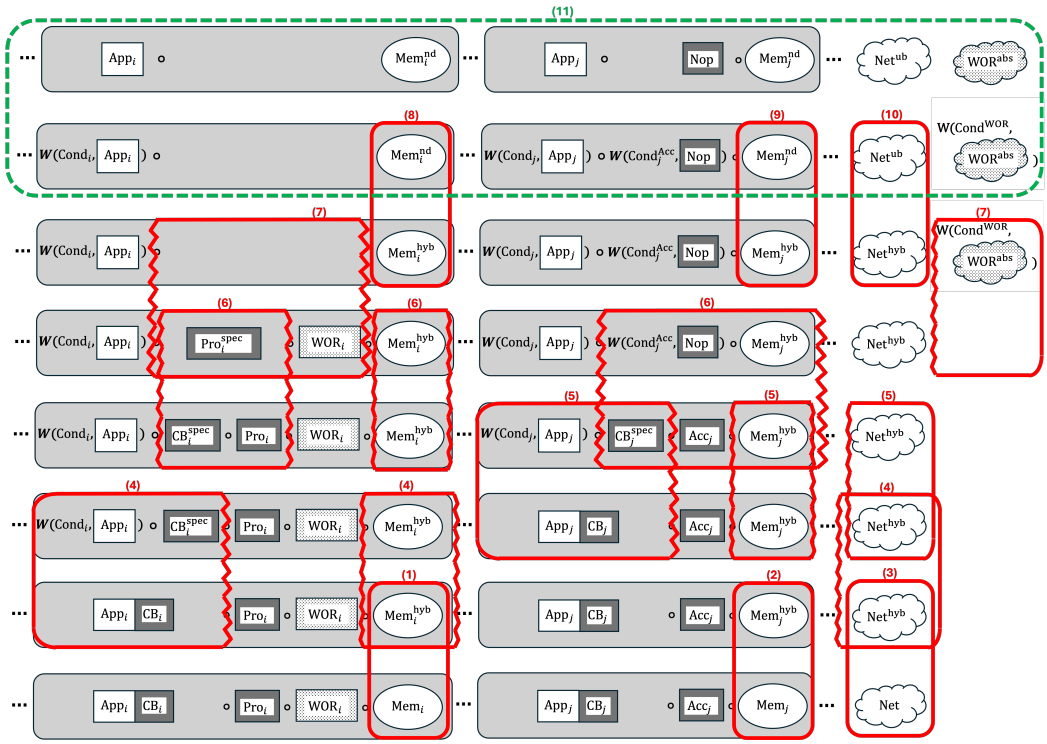

Fig. 6. Composition of the end-to-end Paxos/WOR refinement. The figure shows representative proposer and acceptor nodes; client nodes are omitted because they remain unchanged unconditional context.

been reduced to their final executable abstractions, and all Paxos/WOR behavior is represented by one global atomic WOR^{abs} module. The red boxes mark the contextual refinements proved separately and then composed; the dashed green box is the final cancellation step.

Hybrid Memory and Network (1)–(3). The first three refinements replace the concrete shared infrastructure by hybrid imaginary specifications. Refinements (1) and (2) replace the memory modules on proposer and acceptor nodes by Mem^{hyb}. This hybrid memory specification supports ownership-style reasoning for memory blocks used by the verified Paxos/WOR stack, while preserving an operational memory model for ordinary application code. Refinement (3) similarly replaces the unreliable network by Net^{hyb}, which gives pre/postcondition-style reasoning principles for Paxos sockets but keeps non-Paxos sockets executable. These infrastructure refinements are explained in §4.

Application Callbacks (4)–(5). Refinements (4) and (5) are the proof obligations assigned to application developers. On a proposer node, refinement (4) abstracts the application-supplied callbacks CB_i into their callback specification $\text{CB}_i^{\text{spec}}$, while leaving the rest of the application code executable as $\mathcal{W}(\text{Cond}_i, \text{App}_i)$. The wrapper $\mathcal{W}$ does not replace the application by a mathematical model; it only inserts ownership assertions at the points where the application registers callbacks and invokes the WOR interface. Thus the ordinary application code remains concrete, while the refinement proof requires the application to install suitable callbacks and use the proposer-side library interfaces according to their required discipline.

Refinement (5) performs the analogous abstraction on acceptor nodes. The acceptor-side application remains as $\mathcal{W}(\mathrm{Cond}_j, \mathrm{App}_j)$, while its Paxos-facing callbacks are replaced by $\mathrm{CB}_j^{\mathrm{spec}}$. Since the concrete applications and callbacks may allocate, read, and write node-local memory and send or receive packets over the network, both refinements involve $\mathrm{Mem}^{\mathrm{hyb}}$ and $\mathrm{Net}^{\mathrm{hyb}}$. These application obligations are explained in §5.

Paxos Abstraction (6). Refinement (6) is the Paxos safety proof. On proposer nodes, the callback specification and the concrete proposer library are abstracted together into the proposer specification $\mathrm{Pro}_i^{\mathrm{spec}}$. This specification exposes the Paxos reasoning principles needed by WOR, such as the fact that once a value has been chosen for a key, later successful observations for the same key must agree with it. On acceptor nodes, the callback specification and the concrete acceptor library are abstracted into $\mathcal{W}(\mathrm{Cond}_j^{\mathrm{Acc}}, \textsc{Nop})$: after the Paxos proof, the acceptor code has no direct application-visible behavior, but its wrapper records the invariant preservation needed by the rest of the composition. This refinement still involves $\mathrm{Mem}^{\mathrm{hyb}}$, because the concrete proposer and acceptor libraries store Paxos state in node-local memory, and the proof uses the ownership assertions provided by the hybrid memory specification for Paxos-owned blocks. In contrast, the network module itself is not involved in this refinement proof: all Paxos network effects exposed to the libraries are already captured by the callback specifications established by refinements (4) and (5). §6 gives the Paxos proof.

WOR Abstraction (7). Refinement (7) abstracts the distributed WOR implementation into a single global atomic $\mathrm{WOR}^{\mathrm{abs}}$ module, wrapped as $\mathcal{W}(\mathrm{Cond}^{\mathrm{WOR}}, \mathrm{WOR}^{\mathrm{abs}})$ until the final cancellation step. This proof uses only the proposer specifications $\mathrm{Pro}_i^{\mathrm{spec}}$ as the interface to Paxos; the acceptor implementations and Paxos network traffic have already been hidden by refinement (6). The main proof obligation is to relate the distributed WOR behavior exposed through the proposer specifications to the atomic read/write behavior of $\mathrm{WOR}^{\mathrm{abs}}$. The WOR proof is detailed in §7.

Removing Proof-Only Infrastructure (8)–(10). After refinements (6) and (7), the Paxos-owned memory blocks and Paxos-owned sockets have been abstracted away, so the final executable model no longer needs the ownership-bearing hybrid specifications. Refinements (8) and (9) replace $\mathrm{Mem}^{\mathrm{hyb}}$ by $\mathrm{Mem}^{\mathrm{nd}}$ on proposer and acceptor nodes. The latter preserves ordinary application memory behavior but allocates addresses nondeterministically, accounting for the fact that memory blocks internal to Paxos/WOR have disappeared from the abstraction. Refinement (10) replaces $\mathrm{Net}^{\mathrm{hyb}}$ by $\mathrm{Net}^{\mathrm{ub}}$: non-Paxos sockets still execute operationally, while attempts by any remaining code (*i.e.,* ordinary application code or client code omitted from Fig. 6) to use Paxos-owned sockets trigger undefined behavior. Thus the final network model expresses the noninterference assumption that applications communicate with Paxos only through their verified callbacks, and that clients do not communicate with Paxos directly. These are explained in §4.

Cancellation (11). The final step, refinement (11), is the dashed green box in Fig. 6. Unlike the preceding contextual refinements, this is a global refinement over the whole linked system; hence the client nodes omitted from the figure are also included in this step, although they remain unchanged. It removes the remaining wrappers $\mathcal{W}(\mathrm{Cond}, -)$ around applications, acceptor-side $\textsc{Nop}$, and $\mathrm{WOR}^{\mathrm{abs}}$ by CRIS's cancellation theorem. Each condition contains both assumptions and guarantees. For example, the WOR condition guarantees that WOR uses the Paxos interfaces according to their required discipline, while assuming the Paxos facts exposed by the proposer specifications. In the composed system, every assumption in one condition is discharged by the corresponding guarantee in another condition. Once these assumption/guarantee pairs are matched, CRIS can automatically cancel the proof-only conditions from the final executable model. The

```
Module Net.
var net : socket → Set string := λ _, {}
def sendto (s: socket, pkt: string) ≡
  net := net[s ↦ net(s) ∪ {pkt}];

def recvfrom (s : socket) : option string ≡
  pkts ← net(s);
  opkt ← choose Opt(pkts); return opkt;
```

where $\mathrm{Opt}(M) = \{\ \mathsf{None}\ \} \cup \{\ \mathsf{Some}(m) \mid m \in M\ \}$

Fig. 7. Net: The underlying network module

```
Module Net.
var net : socket → Set string := λ _, {}
def sendto (s : socket, pkt : string) ≡
  if (s ∈ paxos_sockets)
  then pkts ← take(Set string);
       Assume(s ↦[net] pkts);
       Guaran(s ↦[net] pkts ∪ {pkt});
  else net := net[s ↦ net(s) ∪ {pkt}];

def recvfrom (s : socket) : option string ≡
  if (s ∈ paxos_sockets)
  then pkts ← take(Set string);
       Assume(s ↦[net] pkts);
       Guaran(s ↦[net] pkts);
  else { pkts ← net(s); }
  opkt ← choose Opt(pkts); return opkt;
```

Fig. 8. $\mathrm{Net}^{\mathrm{hyb}}$: Hybrid specification of the network module.

```
Module Net.
var net : socket → Set string := λ _, {}
def sendto (s : socket, pkt : string) ≡
  if (s ∈ paxos_sockets) then take(∅);
  else net := net[s ↦ net(s) ∪ {pkt}];

def recvfrom (s : socket) : option string ≡
  if (s ∈ paxos_sockets) then take(∅);
  else { pkts ← net(s); }
  opkt ← choose Opt(pkts); return opkt;
```

Fig. 9. $\mathrm{Net}^{\mathrm{ub}}$: top-level specification of the network module.

resulting top-level abstraction contains the ordinary applications, $\mathrm{Mem}^{\mathrm{nd}}$, $\mathrm{Net}^{\mathrm{ub}}$, acceptor-side Nop, and the centralized atomic $\mathrm{WOR}^{\mathrm{abs}}$.

Testing over the Abstraction. Because all remaining modules in the final row are executable, the top-level abstraction can be used as a testing target for applications and their clients. Compared with testing the full implementation, this model removes the internal nondeterminism of Paxos message loss, duplication, reordering, repeated receipt of old Paxos packets, and races for consensus, while preserving the ordinary application memory and non-Paxos network behavior relevant to clients. §8 reports our testing results.

4 Memory and Network Model

This section defines the shared infrastructure modules used in refinements (1)–(3) and (8)–(10) of §3. The first group replaces the concrete modules by hybrid specifications: refinements (1) and (2) replace each node's Mem by $\mathrm{Mem}^{\mathrm{hyb}}$, and refinement (3) replaces Net by $\mathrm{Net}^{\mathrm{hyb}}$. The second group removes the proof-only assertions from these hybrid specifications: refinements (8) and (9) replace $\mathrm{Mem}^{\mathrm{hyb}}$ by $\mathrm{Mem}^{\mathrm{nd}}$, and refinement (10) replaces $\mathrm{Net}^{\mathrm{hyb}}$ by $\mathrm{Net}^{\mathrm{ub}}$. We describe the network modules first, followed by the memory modules.

Network. Fig. 7 defines the concrete network module Net. The module state maps each socket to a set of packets. Sending a packet inserts it into the destination socket's set. Receiving from a socket nondeterministically returns either None or any packet currently in that set. Since receiving does not remove the selected packet, a packet may be observed more than once; since any packet in the set may be chosen, packets may be observed out of order; and since the choice may keep returning None or other packets, a sent packet may never be observed. Thus Net models packet loss, duplication, and reordering.

Fig. 8 gives the hybrid network specification $\mathrm{Net}^{\mathrm{hyb}}$. It is parameterized by `paxos_sockets`, the set of sockets used for Paxos communication. For sockets outside this set, $\mathrm{Net}^{\mathrm{hyb}}$ uses the same

operational behavior as Net. These sockets therefore remain available to ordinary applications and clients. For Paxos sockets, Net^{hyb} replaces the operational state update by ownership-based assumptions and guarantees over the socket predicate $s \overset{\text{net}}{\mapsto} pkts$. A send on a Paxos socket assumes ownership of the current packet set and guarantees ownership of the set extended with the sent packet. A receive assumes and returns ownership of the same packet set, then nondeterministically chooses a possible receive result from it. This split is the reason the network specification is hybrid. Paxos and callback proofs can reason about Paxos sockets using separation-logic ownership, while unrelated network traffic remains ordinary executable code.

The refinement proof relates Net and Net^{hyb} with an invariant that stores the concrete packet sets for Paxos sockets in the logical socket resources and keeps the concrete and hybrid module states equal on non-Paxos sockets. For Paxos sockets, the proof opens the logical socket resource, performs the corresponding concrete network update or read, and re-establishes the resource required by the guarantee. For non-Paxos sockets, the two modules take the same operational steps.

Fig. 9 defines the final executable network model Net^{ub}. After the Paxos-owned sockets have been abstracted away, the final model no longer needs the ownership assertions in Net^{hyb}. On non-Paxos sockets, Net^{ub} behaves like the concrete network, so applications and clients can still communicate and be tested. On Paxos sockets, Net^{ub} executes **take**($\emptyset$), which triggers undefined behavior and therefore allows arbitrary behavior. Hence the Paxos-socket cases of the refinement from Net^{hyb} to Net^{ub} are trivial, while the non-Paxos-socket cases use the same operational steps on both sides. The undefined behavior is also the executable form of the noninterference assumption used in the top-level abstraction: after the callbacks have been verified and Paxos has been hidden, the remaining application and client code must not directly use Paxos-owned sockets.

Memory. The memory refinements follow the hybrid memory construction of CRIS [13], so we do not repeat the proof details here (see Sec. 7 of the CRIS paper). The concrete memory module Mem is first replaced by a hybrid memory specification Mem^{hyb}. At each allocation, Mem^{hyb} lets the proof choose between the usual ownership assertions and the usual operational model. We use ownership assertions for memory blocks owned by verified components, and the operational model for ordinary application code. After Paxos and WOR have been abstracted away, the hybrid memory specification Mem^{hyb} is replaced by Mem^{nd}. This final memory model keeps ordinary reads and writes executable, but makes allocation choose fresh addresses nondeterministically. The nondeterminism accounts for memory blocks that were present in the implementation but have disappeared because Paxos/WOR-owned state has been abstracted away.

5 Verification of Applications

This section explains the application-side proof obligations, refinements (4) and (5) in Fig. 6. These refinements verify the Paxos-facing callbacks supplied by applications and prove that the application code satisfies the required registration discipline at the points where it interacts with Paxos or WOR. This requires reasoning about only a small fraction of the application code: the callback implementations, callback registration calls, and Paxos/WOR interface calls. The rest of the application is not replaced by an abstract model; after the refinement, it remains executable, wrapped only with the proof-only conditions $\mathcal{W}(\text{Cond}, -)$.

Callback Specification. Fig. 10 shows the callback specification CB^{spec} used by the Paxos proof. The specification hides the concrete packet format and socket operations performed by the implementation. Instead, it exposes a logical Paxos mailbox resource $nid \overset{\text{paxos}}{\mapsto} msgs$, where *msgs* is the set of Paxos messages currently available for node *nid*. Here messages are lists of values before serialization, whereas the packets in §4 are strings sent through the network module.

```
def cb_send_nid(dest: ℕ,
                buffer: ptr, size: ℕ) ≡
  msg ← take(list val);
  Assume(⌜len(msg) = size⌝ ∗
         buffer ↦^mem_nid msg);
  𝒴;
  msgs ← take(Set (list val));
  Assume(dest ↦^paxos msgs);
  Guaran(dest ↦^paxos msgs ∪ {msg});
  𝒴;
  Guaran(buffer ↦^mem_nid msg);
```

```
def cb_recv_nid (buffer: ptr, size: ℕ) : 𝔹 ≡
  Assume(∃ m, ⌜len(m) = size⌝ ∗ buffer ↦^mem_nid m);
  𝒴;
  msgs ← take(Set (list val));
  Assume(nid ↦^paxos msgs);
  Guaran(nid ↦^paxos msgs);
  𝒴;
  ret ← choose 𝔹;
  Guaran(∃ m, ⌜len(m) = size⌝ ∗ buffer ↦^mem_nid m ∗
    ∃ msg, ⌜(ret = false ∨ msg ∈ msgs) ∧ m =_prefix msg⌝);
  return ret;
```

Fig. 10. $\mathrm{CB}^{\mathrm{spec}}$: Imaginary specification of callbacks.

The send callback $\mathrm{cb\_send}_{nid}(\mathrm{dest}, \mathrm{buffer}, \mathrm{size})$ owns the memory block containing the message to be sent. Its specification reads that message from the buffer, transfers it to the logical mailbox of `dest`, and returns the buffer ownership unchanged. A concrete callback implementation may serialize the message into a packet and send that packet over the network using $\mathrm{Net}^{\mathrm{hyb}}$, but its proof must show that this concrete behavior refines the mailbox update in $\mathrm{CB}^{\mathrm{spec}}$.

The two $\mathcal{Y}$ points in the specification are also part of the abstraction. A concrete callback may yield between every physically atomic command while it serializes the message and sends the resulting packet. In $\mathrm{CB}^{\mathrm{spec}}$, however, the segment between the two yields contains no yield point at which another thread can interfere. Thus the update to the destination mailbox is the callback's logically atomic linearization point.

The receive callback $\mathrm{cb\_recv}_{nid}(\mathrm{buffer}, \mathrm{size})$ owns the destination buffer and consults the logical mailbox of node *nid*. As with cb_send, $\mathrm{CB}^{\mathrm{spec}}$ accesses this mailbox logically atomically between its two $\mathcal{Y}$ points. It may return `false`, reflecting failure to receive a useful Paxos message. If it returns `true`, the buffer must contain a message compatible with some message in the mailbox. A concrete callback implementation may receive a packet through $\mathrm{Net}^{\mathrm{hyb}}$ and deserialize it into a message, but its proof must justify the returned buffer contents against the logical mailbox. The relation $m_1 =_{\mathrm{prefix}} m_2$ means that either m_1 is a prefix of m_2 or m_2 is a prefix of m_1. This accommodates the usual fixed-buffer behavior: a message longer than the buffer may be truncated, while a message shorter than the buffer may occupy only a prefix of the buffer contents.

Application Obligations. Refinements (4) and (5) replace the concrete callbacks on each node by $\mathrm{CB}^{\mathrm{spec}}$, but leave the rest of the application in place. On a proposer node i, the refinement turns the concrete application and callbacks into $\mathcal{W}(\mathrm{Cond}_i, \mathrm{App}_i)$ linked with $\mathrm{CB}_i^{\mathrm{spec}}$. On an acceptor node j, it similarly turns the concrete application and callbacks into $\mathcal{W}(\mathrm{Cond}_j, \mathrm{App}_j)$ linked with $\mathrm{CB}_j^{\mathrm{spec}}$. Both refinements involve $\mathrm{Mem}^{\mathrm{hyb}}$ and $\mathrm{Net}^{\mathrm{hyb}}$, because concrete callbacks manipulate memory buffers and communicate over the network, serializing and deserializing Paxos messages as network packets.

Fig. 11 illustrates the proposer-side obligation. The conditions inserted on the right match the shaded assertions in the WOR specification of Fig. 15. In the WOR specification, $\mathtt{WOR}^{\mathsf{Abs}}$`.init` assumes that the two function pointers resolve to `cb_send` and `cb_recv` and that the caller owns $\mathtt{WORUninit}(nid)$. Thus, immediately before calling `init`, the wrapped application must guarantee exactly those facts. Conversely, `init` guarantees $\mathtt{WORInit}(nid)$ when it returns, so the wrapped application may assume $\mathtt{WORInit}(nid)$ after the call. The `read` and `write` conditions follow the same pattern: the application guarantees $\mathtt{WORInit}(nid)$ before the call and receives it back after the call.

```
... code1 ...                                  ... code1 ...
                                               Guaran(⌜get(genv,fptr1) = cb_send⌝ ∗
                                                      ⌜get(genv,fptr2) = cb_recv⌝ ∗
                                                      WORUninit(nid));
WOR.init_nid(fptr1, fptr2);                    WOR.init_nid(fptr1, fptr2);
                                               Assume(WORInit(nid));
... code2 ...                                  ... code2 ...
                                   ⊑ctx        Guaran(WORInit(nid));
WOR.write_nid(0,42);                           WOR.write_nid(0,42);
                                               Assume(WORInit(nid));
... code3 ...                                  ... code3 ...
                                               Guaran(WORInit(nid));
v ← WOR.read_nid(0);                           v ← WOR.read_nid(0);
                                               Assume(WORInit(nid));
... code4 ...                                  ... code4 ...
```

Fig. 11. Application verification: implementation App on the left and $\mathcal{W}$(Cond, App) on the right.

These assertions should not be read as conventional pre/postconditions describing the behavior of WOR. The behavior of WOR is the executable stateful model inside the function bodies of Fig. 15: reads and writes inspect and update the abstract map `kvmap` and may nondeterministically fail. The shaded assertions instead state only what the application must provide in order to call the WOR interface. This separation is deliberate. Since the abstract behavior is written as executable code rather than as separation-logic pre- and postconditions, the final abstract system can still be run to test applications.

The simple conditions capture two subtle obligations. First, the application must initialize WOR with the verified callbacks. Because callbacks are passed through function pointers, the proof must connect the concrete pointers used by the program to the callback names in $\mathrm{CB}^{\mathrm{spec}}$. CRIS models this with a global environment, and the `init` condition requires `get(genv,fptr1)` = `cb_send` and `get(genv,fptr2)` = `cb_recv`. In typical code, these facts are immediate because the application obtains the pointers by taking the addresses of `cb_send` and `cb_recv`.

Second, WOR operations must be globally serialized: across all application threads, there may be at most one active WOR call at a time. This is not merely a per-node obligation. Since application code runs concurrently, two threads could try to call WOR at the same time. `WORInit`(*nid*) prevents this in the proof: it is a non-duplicable resource, and every `read` or `write` call takes the resource at entry and returns it only when the call finishes. Therefore two concurrent calls cannot both satisfy their pre-call guarantees unless the application explicitly synchronizes them and transfers this single resource between the callers.

The proof in Fig. 11 is straightforward because all WOR calls in the displayed code are made by the same thread. At the beginning of the program, the application proof is given `WORUninit`(*nid*). The identical fragment `code1` is advanced on both sides by CRIS's simulation rule for identical code, while framing `WORUninit`(*nid*). The proof does not need to analyze the behavior of this fragment. Even if the fragment contains an application bug such as division by zero, the same code appears on both sides, so the simulation matches it directly.

The first real proof obligation appears at `WOR.init`. At this point the proof establishes the pointer equalities from the way `fptr1` and `fptr2` were obtained and gives away `WORUninit`(*nid*). The function call itself is identical on the two sides, so the simulation steps over the call and obtains `WORInit`(*nid*) from the post-call assumption. The proof then frames `WORInit`(*nid*) through `code2`,

```
def init_nid(send_ptr: ptr, recv_ptr: ptr) ≡
  Assume(⌜get(genv,send_ptr) = cb_send ∧ get(genv,recv_ptr) = cb_recv⌝ ∗ ProUninit(nid))
  𝒴;
  Guaran(QuorumMap(nid, ∅));
def prepare_nid(k : key): quorum_state ≡
  qmap ← take (key ⇀^fin quorum_state);
  Assume(QuorumMap(nid, qmap));
  𝒴;
  ps ← take (Set value × option value);
  Assume(PaxosState(k, ps));
  Guaran(PaxosState(k, ps));
  𝒴;
  q ← choose quorum_state;
  Guaran(⌜q ∈ valid_quorums(ps)⌝ ∗
         QuorumMap(nid, qmap[k ↦ q]));
  return q
;
def commit_nid(k : key, v: value) ≡
  qmap ← take (key ⇀^fin quorum_state);
  Assume(⌜qmap(k) ∈ {AnyC, OnlyC(v)}⌝ ∗
         QuorumMap(nid, qmap));
  𝒴;
  ps ← take (Set value × option value);
  Assume(PaxosState(k, ps));
  Guaran(∃ ns, ⌜ns ∈ next_states(ps,v)⌝ ∗
         PaxosState(k, ns));
  𝒴;
  Guaran(QuorumMap(nid, qmap[k ↦ ⊥]));
```

$$\texttt{valid\_quorums}(ps) \triangleq \begin{cases} \{\texttt{Fail}, \texttt{AnyC}\} \cup \{\texttt{OnlyC}(v) \mid v \in ps.1\} & \text{if } ps.2 = \texttt{None} \\ \{\texttt{Fail}, \texttt{OnlyC}(v), \texttt{Cons}(v)\} & \text{if } ps.2 = \texttt{Some}(v) \end{cases}$$

$$\texttt{next\_states}(ps, v) \triangleq \{\, (ps.1 \cup \{v\}, ps.2) \,\} \cup \{\, (ps.1 \cup \{v\}, \texttt{Some}(v)) \mid ps.2 = \texttt{None} \,\}$$

Fig. 12. Pro^{spec}: Imaginary specification of Proposer module

(1) $\text{QuorumMap}(nid, qm) * \ulcorner qm(k) = \text{Cons}(v) \urcorner \vdash \text{Consensus}(k, v)$
(2) $\text{Consensus}(k, v) \vdash \text{Consensus}(k, v) * \text{Consensus}(k, v)$
(3) $\text{Consensus}(k, v) * \text{Consensus}(k, v') \vdash \ulcorner v = v' \urcorner$

Fig. 13. Properties about $\text{Consensus}(k, v)$

gives it away for the `write` call, receives it back after the call, and repeats the same pattern for `code3` and `read`.

In our bank example, all WOR calls are made by a dedicated thread, so the ownership of $\texttt{WORInit}(nid)$ remains in that thread and the application-side proof is almost entirely routine. If different threads can call WOR, the proof becomes less direct: the implementation must globally synchronize those calls, for example with locks, and the refinement proof must use the synchronization invariant to pass $\texttt{WORInit}(nid)$ from one caller to another. This is exactly the discipline that WOR requires from applications. Once the callbacks are verified and the application establishes the callback-registration and sequential-call obligations above, the contextual refinement in Fig. 11 has no further application-specific requirements.

6 Verification of Paxos Library

We first introduce the imaginary specifications Pro^{spec} for Proposer (§6.1) and $\mathcal{W}(\text{Cond}^{\text{Acc}}, \text{Nop})$ for Acceptor (§6.2). We then explain the proof architecture: the module-local invariant that connects the concrete Paxos code to these specifications, and the prophecy/helping refinements used to linearize `commit` and `prepare`.

6.1 Imaginary Specification for Proposer

Fig. 12 presents the imaginary specification for the Proposer module. It has the same public operations as the implementation, but it hides the Paxos machinery that clients should not have to reason about: proposal numbers, Acceptor promises, protocol messages, retransmission, and

quorum collection. Instead, the specification exposes a small abstract state machine described by two primary resources.

The first resource is the global Paxos state. The assertion PaxosState(k, ps) is the key-indexed piece of this global resource, where ps has type $\mathcal{P}(\text{value}) \times \text{option value}$. The first component ps.1 records the values that have been submitted to Paxos for key k. The second component ps.2 records the value for which consensus has been made. Thus ps.2 = None means that consensus has not yet been made for k, while ps.2 = Some(v) means that consensus has been made on v. This is the state that a client of Paxos sees; the many Acceptor-local states that implement it are hidden behind the refinement proof.

The second resource is node-local. For each Proposer node *nid*, QuorumMap(*nid*, qmap) records what that node currently knows from its latest prepare calls. The map qmap is a finite partial map from keys to abstract quorum states: Fail, AnyC, OnlyC(v), or Cons(v). We write qmap(k) = $\bot$ when qmap is undefined at k. This resource is local to the Proposer node, so it can enforce the usual Paxos discipline without exposing proposal numbers to clients. After initialization, init returns QuorumMap(*nid*, $\emptyset$), so qmap is initially undefined for every key. Before owning this resource (*i.e.,* without calling init), the node has no authority to call prepare or commit.

The prepare specification is a read of this simplified Paxos state. It takes the node's current QuorumMap, atomically reads PaxosState(k, ps) without changing it, and then returns some quorum state q allowed by valid_quorums(ps). If ps.2 = None, prepare may fail, return AnyC to say that the caller may commit any value, or return OnlyC(v) to say that the caller may commit only v. If ps.2 = Some(v), it may fail, return OnlyC(v), or return Cons(v) to say that consensus has already been made on v. From the caller's viewpoint, Fail gives no permission to commit, and Cons(v) means no further commit call is needed for k. After choosing such a result q, the specification stores it in the node-local map as qmap[k $\mapsto$ q] and returns the result q. The client therefore reasons only about the returned abstract case, not about which Acceptors responded.

The commit specification is the corresponding update. To call $\texttt{commit}_{nid}$(k, v), the node-local map must show that the latest usable prepare result for k is either AnyC or OnlyC(v). The first case permits committing any value because no consensus constraint was observed. The second case enforces the Paxos rule that, if prepare says only v may be submitted, the following commit must use that same value. The operation then atomically updates PaxosState(k, ps) to some $ns \in$ next_states(ps, v). This transition always adds v to the submitted-value set. If no value had reached consensus, the transition may either leave consensus unset or set it to Some(v); if consensus was already set, it cannot be changed. Finally, the specification changes qmap so that it is undefined at k again. Thus qmap(k) = $\bot$ happens both initially and after commit uses the prepare result for k.

As in the callback specifications of §5, the absence of a $\mathcal{Y}$ inside a resource access expresses logical atomicity. Here, both prepare and commit access PaxosState(k, ps) without an intervening $\mathcal{Y}$, so clients observe the read and update of the shared Paxos state as single atomic effects. The specification also abstracts away the implementation's finer interleavings: the concrete Paxos code may yield after each physically atomic instruction, but clients reason about the coarse-grained effects shown in Fig. 12.

Fig. 13 packages the stable fact exposed by a successful consensus observation. When a node-local map contains Cons(v) for key k, rule (1) derives the helper resource Consensus(k, v). This helper is persistent: rule (2) allows it to be duplicated and retained after the local QuorumMap continues to change. Rule (3) states the client-facing safety property: two persistent consensus facts for the same key must agree on the value. Thus, once a client has derived Consensus(k, v), it can treat the choice of v as a permanent fact about key k, without knowing how Paxos established it.

```
def init_nid(send_ptr, recv_ptr : ptr) ≡
  Assume(⌜get(genv,send_ptr) = cb_send⌝ ∗
         ⌜get(genv,recv_ptr) = cb_recv⌝ ∗
         AccUninit(nid));
  𝒴;
  Guaran(AccInit(nid));
```

```
def run_once_nid() ≡
  Assume(AccInit(nid));
  𝒴;
  Guaran(AccInit(nid));
```

Fig. 14. $\mathcal{W}(\mathsf{Cond}^{\mathsf{Acc}}, \mathsf{Acc}^{\mathsf{Nop}})$: Imaginary specification of Acceptor module

6.2 Imaginary Specification for Acceptor

Fig. 14 presents the imaginary specification for the Acceptor module. It is a no-op module, $\mathsf{Acc}^{\mathsf{Nop}}$, wrapped by the condition $\mathsf{Cond}^{\mathsf{Acc}}$. The wrapper supplies the shaded logical assertions in the figure; the executable body itself exposes no Paxos protocol state. $\mathsf{Cond}^{\mathsf{Acc}}$ corresponds exactly to these shaded logical assertions. As discussed in §3, this condition is proof-only structure and is removed later by the final cancellation step.

The `init` specification checks that the supplied callback pointers are the verified send and receive callbacks, consumes the uninitialized resource $\mathsf{AccUninit}(nid)$, and returns $\mathsf{AccInit}(nid)$. This records only the public lifecycle fact that Acceptor node *nid* has been initialized. After that, `run_once` simply consumes and returns $\mathsf{AccInit}(nid)$ across one $\mathcal{Y}$. Thus the abstract Acceptor interface says that an initialized Acceptor may be scheduled, but scheduling it has no direct client-visible Paxos effect.

This is deliberate. Concrete Acceptor steps receive prepare and commit messages, update promises and accepted values, send acknowledgments, and may help linearize a Proposer operation. The specification hides all of these internal actions. Their only externally useful consequence is that the Proposer specification remains sound: clients observe Paxos through PaxosState, QuorumMap, `prepare`, `commit`, and the persistent Consensus resource, not by inspecting Acceptor-local state.

6.3 Proof Architecture

We now present the proof architecture of the Paxos verification: a refinement between the concrete Proposer and Acceptor modules and their corresponding imaginary specifications, which corresponds to refinement (6) in the proof outline of Fig. 6:

$$(\mathsf{CB}_1^{\mathsf{spec}} \circ \mathsf{Pro}_1 \circ \mathsf{Mem}_1^{\mathsf{hyb}}) \circ \cdots \circ (\mathsf{CB}_n^{\mathsf{spec}} \circ \mathsf{Pro}_n \circ \mathsf{Mem}_n^{\mathsf{hyb}}) \circ (\mathsf{CB}_{n+1}^{\mathsf{spec}} \circ \mathsf{Acc}_{n+1} \circ \mathsf{Mem}_{n+1}^{\mathsf{hyb}}) \circ \cdots \circ (\mathsf{CB}_{n+m}^{\mathsf{spec}} \circ \mathsf{Acc}_{n+m} \circ \mathsf{Mem}_{n+m}^{\mathsf{hyb}})$$
$$\sqsubseteq_{\mathsf{ctx}} (\mathsf{Pro}_1^{\mathsf{spec}} \circ \mathsf{Mem}_1^{\mathsf{hyb}}) \circ \cdots \circ (\mathsf{Pro}_n^{\mathsf{spec}} \circ \mathsf{Mem}_n^{\mathsf{hyb}}) \circ (\mathcal{W}(\mathsf{Cond}_{n+1}^{\mathsf{Acc}}, \mathsf{Nop}) \circ \mathsf{Mem}_{n+1}^{\mathsf{hyb}}) \circ \cdots \circ (\mathcal{W}(\mathsf{Cond}_{n+m}^{\mathsf{Acc}}, \mathsf{Nop}) \circ \mathsf{Mem}_{n+m}^{\mathsf{hyb}})$$

This refinement builds on the imaginary specifications of the callback functions and the hybrid specification of the memory module. The Proposer and Acceptor modules run the Multi-Paxos protocol to reach consensus on a string value for each key. We start from the `prepare` operation of the Proposer module. Since the protocol treats different keys independently, the following discussion fixes an arbitrary key k and describes the Paxos instance for that key.

The* `prepare` *operation. The `prepare` operation is the phase in which a Proposer prepares for a subsequent `commit` operation. The Proposer sends a *prepare request* to all Acceptors to ask whether it may commit a value for key k. It then collects the Acceptor responses and chooses its next action from their summary.

The resulting choices are as follows. If it fails to receive responses from a quorum, *i.e.,* from more than half of the Acceptors, it returns the network-failure result `Fail`. If the responses show that a value v' has already reached consensus, then it returns the consensus value $\mathtt{Cons}(v')$ to the client. If no Acceptor in the quorum has accepted a value, it returns `AnyC`. Finally, if some Acceptors in the quorum have accepted values but no consensus exists, the Proposer chooses the value v determined by the Paxos ordering among those accepted values and returns $\mathtt{OnlyC}(v)$.

The imaginary specification for this implementation is the `prepare` operation of Fig. 12, which atomically reads PaxosState(k, ps) without changing it. Specifically, for the chosen quorum summary q, `prepare` guarantees $\ulcorner q \in \texttt{valid\_quorums}(ps) \urcorner$, which implies two important properties. First, if $ps.2$ is `Some`(v), then `prepare` cannot return `AnyC`, nor can it return `OnlyC`(v') or `Cons`(v') for any value $v' \neq v$. Second, if $ps.2$ is `None`, then `prepare` does not return consensus (*i.e.,* `Cons(_)`).

***The proof of* `prepare`.** The refinement proof for `prepare` is carried out as a local CRIS simulation between the `prepare` implementation and the imaginary specification above. Within this simulation, the most important obligation is to *identify the atomic point* at which the source-side round trip, consisting of **`Assume(`**PaxosState(k, ps)**`)`** followed immediately by **`Guaran(`**PaxosState(k, ps)**`)`**, should happen. The reason is that the Proposer request and the corresponding Acceptor responses take multiple, non-atomic implementation steps, while the source-side round trip must occur atomically.

A natural candidate atomic point is when the Proposer collects all responses from the Acceptors via the callbacks. After observing all the responses, the proof knows the exact summary of the quorum. Although this works in some cases, it does not work in general. Specifically, this late atomic point can justify the quorum summary `Cons`(v). However, it does not justify `AnyC` or `OnlyC`(v) in general: at that point, PaxosState may already record consensus on a different value v'.

The opposite choice is to take the beginning of `prepare` as the atomic point. This can justify `AnyC`, because observing the quorum summary `AnyC` at the end of `prepare` implies that no consensus had been made at the beginning. However, this point is too early in general: it does not justify the quorum summaries `OnlyC`(v) or `Cons`(v), because at the beginning of `prepare` the value v may not have been submitted yet.

Indeed, no single fixed point handles all cases. Instead, the proof chooses the atomic point by case analysis on the quorum summary that `prepare` will eventually return. The remaining question is where to place the atomic point for `OnlyC`(v). Its position depends on when v becomes submitted relative to the execution of `prepare`. If v has already been submitted (*i.e.,* $v \in ps.1$) before `prepare` starts, the atomic point can be the beginning of the method. Otherwise, the source-side round trip can be placed at the point where v is first accepted by an Acceptor.

Prophecy and Helping. The discussion above assumes that the proof can branch on the quorum summary that `prepare` will eventually return. However, this information is not available to the local simulation in advance: at the beginning of `prepare`, the simulation does not yet know which quorum summary the implementation will observe.

To address this, we use prophecy variables, which provide proof-only information about the future. They let the proof first perform a case analysis over predicted future behaviors and then continue the simulation separately for each prediction. For a given predicted future, the proof only needs to consider executions that resolve the prophecy consistently; executions inconsistent with the prediction are ruled out. For technical convenience, instead of merely predicting the final quorum summary directly, it predicts the sequence of prepare and commit requests processed by Acceptors, from which the proof can determine the future quorum summary returned by any `prepare` call. See ConCRIS [29] for how prophecy variables are supported.

Given each predicted future, the simulation can proceed as discussed above. If the prediction is `AnyC`, the proof performs the source-side round trip at the beginning of `prepare`. If the prediction is `Cons`(v), the proof can instead wait until the end of `prepare` and perform the round trip there.

However, the `OnlyC`(v) case exposes another problem. As discussed above, if $v \in ps.1$ at the beginning, the proof can perform the round trip immediately. Otherwise, the round trip must be performed when v is first accepted by an Acceptor. The predicted future tells the proof exactly when this happens. The remaining issue is spatial rather than temporal: the right point occurs while

an Acceptor has control, not while the Proposer running `prepare` has control. In other words, the round trip must be performed during the Acceptor's simulation proof rather than the Proposer's.

We address this problem using ConCRIS's support for helping. Helping allows source-side code for one node or thread, *i.e.,* specification-side code, to be executed during another node or thread's execution. In the tricky case above, the Proposer-side proof registers the round trip as a helping job, and the first Acceptor that accepts v completes that job at its own control point. Together, prophecy and helping let the simulation follow the case-analysis strategy above. See ConCRIS [29] for how helping is supported.

***The proofs of* `commit` *and* `run_once`.** The proof of $\texttt{commit}(k, v)$ is simpler. The atomic update of PaxosState can always be placed at the beginning of the method. Given a predicted future, the proof performs a case analysis. If the commit request issued by this `commit` is the one that eventually establishes consensus, the proof updates PaxosState to record consensus on v. Otherwise, it only adds v to the submitted-value component of PaxosState.

For `run_once`, the key point is how the helping job described above is realized. When a `prepare` is prophesied to return $\texttt{OnlyC}(v)$, the proof may leave a pending help obligation for the Acceptor that will accept v. The proof of `run_once` discharges this obligation immediately after that Acceptor accepts v. Since the body of `run_once`'s imaginary specification in Fig. 14 is `Nop`, `run_once` itself does not need an atomic point for its own specification.

7 WOR Verification

This section explains the last library abstraction in the proof: the distributed write-once register (WOR) implementation is replaced by a single logically atomic storage service. The implementation uses the Proposer imaginary specification from §6; the abstraction hides the Paxos messages, Acceptor state, callback scheduling, and Paxos-specific network nondeterminism that were needed to implement WOR.

7.1 Specification of WOR^{Abs}

Fig. 15 gives the abstract WOR model. Its only mutable state is the module-local location `kvmap`. While this location is local to the abstract WOR module, the module itself is globally shared: a write performed through one node updates the same store that any later read or write from any node consults.

For each key, `kvmap` stores either $\texttt{inl}\ S$ or $\texttt{inr}\ v$. The state $\texttt{inl}\ S$ means that no value has been fixed yet, and S is the finite set of values that have been submitted for this key. The state $\texttt{inr}\ v$ means that the write-once value has been fixed to v; after this point, the abstract model never changes the value for this key again. Initially, every key maps to $\texttt{inl}\ \emptyset$.

The operation $\texttt{init}_{nid}$ checks the callback pointers and turns $\mathsf{WORUninit}(nid)$ into $\mathsf{WORInit}(nid)$; it does not modify `kvmap`. The token $\mathsf{WORInit}(nid)$ is then required by both storage operations and returned afterwards. A read may fail nondeterministically and return `Failed`. Otherwise, it reads `kvmap`: $\texttt{inl}\ S$ gives `NoValue`, and $\texttt{inr}\ v$ gives $\mathsf{Value}(v)$.

A write has no return value. It may also fail nondeterministically, leaving `kvmap` unchanged. If it runs while $\texttt{kvmap}(k) = \texttt{inl}\ S$, it first records the submitted value by forming $S' = S \cup \{v\}$. It then either keeps the key pending as $\texttt{inl}\ S'$ or chooses one value from S' and fixes the key as $\texttt{inr}\ x$. If the key is already fixed, the write leaves it unchanged.

These nondeterministic failures summarize failures of the underlying Paxos execution. In particular, a Proposer's `prepare` call may fail to receive enough responses from Acceptors to form a quorum; the abstract model exposes such a failed attempt as `Failed` for a read, or as a write that leaves `kvmap` unchanged.

```
var kvmap: key → ((Set value) + value) := λ_,inl ∅
def init_nid(send_ptr: ptr, recv_ptr: ptr) ≡
  Assume(⌜get(genv,send_ptr) = cb_send ∧ get(genv,recv_ptr) = cb_recv⌝ ∗ WORUninit(nid));
  𝒴;
  Guaran(WORInit(nid));
def read_nid(k: key): result ≡
  Assume(WORInit(nid));
  𝒴;
  if choose 𝔹 then ret ← Failed;
  else match kvmap(k) with
       | inl _ => ret ← NoValue;
       | inr v => ret ← Value(v); end;
  𝒴;
  Guaran(WORInit(nid));
  return ret;
```

```
def write_nid(k: key, v: value) ≡
  Assume(WORInit(nid));
  𝒴;
  if choose 𝔹 then skip;
  else match kvmap(k) with
    | inl S =>
      S' ← S ∪ {v};
      e ← choose {inl S'} ∪ {inr x | x ∈ S'};
      kvmap := kvmap[k ↦ e];
    | inr _ => skip; end;
  𝒴;
  Guaran(WORInit(nid));
```

Fig. 15. $\mathcal{W}(\mathrm{Cond}^{\mathrm{WOR}}, \mathrm{WOR}^{\mathrm{Abs}})$: Atomic WOR model.

```
def init_nid(send_ptr: ptr, recv_ptr: ptr) ≡ Pro_nid.init_nid(send_ptr, recv_ptr);
def read_nid(k: key): result ≡
  repeat num_try {
    q ← Pro_nid.prepare_nid(k);
    match q with
    | Fail => skip
    | AnyC => return NoValue
    | OnlyC(_) => skip
    | Cons(v) => return Value(v)
    end;
  }
  return Failed;
```

```
def write_nid(k: key, v: value) ≡
  repeat num_try {
    q ← Pro_nid.prepare_nid(k);
    match q with
    | Fail => skip
    | AnyC => Pro_nid.commit_nid(k, v); return
    | OnlyC(v') => Pro_nid.commit_nid(k, v'); return
    | Cons(_) => return
    end;
  }
```

Fig. 16. Distributed WOR implementation.

This specification is logically atomic in the sense used throughout CRIS. The only $\mathcal{Y}$ points in `read` and `write` surround the access to `kvmap`; there is no yield while the operation reads the shared location or performs the write-once update. Consequently, each abstract storage operation has a single linearization point, even though the concrete WOR implementation may yield across many Paxos steps and retries.

Finally, the shaded assertions in Fig. 15 are the logical conditions carried by $\mathcal{W}(\mathrm{Cond}^{\mathrm{WOR}}, -)$, not part of the executable model. They serve as the separation-logic pre- and postconditions of the WOR interface, and they match the conditions inserted around application calls in Fig. 11. For example, before calling `init`, the wrapped application guarantees the callback-pointer facts and WORUninit(*nid*); the abstract WOR module assumes the same assertion at the entry of `init`. Such adjacent guarantee/assumption pairs are proof-only artifacts. The cancellation theorem discussed in §3 removes all such matched pairs automatically, so after the final cancellation step the shaded assertions disappear and the remaining WOR model is executable.

7.2 Implementation of WOR

The concrete WOR layer is a thin wrapper around the node-local Proposer interface verified in §6. There is one WOR module at each Proposer node, and Fig. 16 shows all three exported functions. The initialization function $\mathtt{init}_{nid}$ simply forwards the callback pointers to $\mathtt{Pro}_{nid}.\mathtt{init}_{nid}$. Thus WOR performs no additional initialization logic of its own; it relies on Proposer initialization to

$$\mathcal{I}_{\mathrm{WOR}}\ (\mathsf{kvmap} : \mathsf{key} \to ((\mathsf{Set\ value}) + \mathsf{value}))\ ()\ \triangleq\ \exists\ \mathsf{psmap} : \mathsf{key} \to (\mathsf{Set\ value} \times \mathsf{option\ value}),$$
$$\forall k, \mathsf{PaxosState}(k, \mathsf{psmap}(k)) * \ulcorner (\exists S,\ \mathsf{kvmap}(k) = \mathsf{inl}(S) \wedge \mathsf{psmap}(k) = (S, \mathsf{None})) \vee$$
$$(\exists v\, S,\ \mathsf{kvmap}(k) = \mathsf{inr}(v) \wedge \mathsf{psmap}(k) = (S, \mathsf{Some}(v)))\urcorner$$

Fig. 17. Module-local invariant for the WOR abstraction proof.

install the callbacks and create the Proposer-side resources needed by later storage operations. The storage operations do not manipulate Paxos messages or Acceptor state directly. They interact with Paxos only through $\mathtt{Pro}_{nid}\mathtt{.prepare}_{nid}$ and $\mathtt{Pro}_{nid}\mathtt{.commit}_{nid}$.

The `read` and `write` operations both run bounded retry loops, with at most `num_try` attempts. The bound is an implementation parameter that makes the executable code total; it is not part of the abstract storage state. If `read` does not reach a Paxos observation that it can return, it returns `Failed`. If `write` sees only failed prepare attempts, it returns without issuing a commit, which corresponds to a failed abstract write.

The read operation uses `prepare` as a read-only observation of the Paxos state for key k. If `prepare` returns `Fail`, the implementation has not obtained a usable quorum result and retries. If it returns AnyC, the Proposer has received a quorum of responses saying that no value has been submitted, so WOR can return NoValue. If it returns $\mathsf{OnlyC}(v)$, the Proposer has learned that the consensus process has already seen the submitted value v but has not yet produced a stable WOR read result, so the implementation retries. Finally, $\mathsf{Cons}(v)$ means that consensus has been made on v, so the read returns $\mathsf{Value}(v)$.

The write operation follows the same prepare-first discipline. If `prepare` returns `Fail`, it retries. If it returns AnyC, the writer is free to submit its requested value v, so it calls $\mathtt{Pro}_{nid}\mathtt{.commit}_{nid}(k, v)$ and returns. If it returns $\mathsf{OnlyC}(v')$, Paxos requires the next commit to preserve the value observed during prepare, so the WOR implementation commits v' rather than the caller's requested value v and returns. If `prepare` returns `Cons(_)`, the key is already fixed, so there is nothing left for this write to do.

7.3 Proofs for Atomic Abstraction

The result is the contextual refinement (7) in Fig. 6:

$$(\mathsf{Pro}_1^{\mathrm{spec}} \circ \mathsf{WOR}_1) \circ \cdots \circ (\mathsf{Pro}_n^{\mathrm{spec}} \circ \mathsf{WOR}_n) \sqsubseteq_{\mathrm{ctx}} \mathcal{W}(\mathsf{Cond}^{\mathrm{WOR}}, \mathsf{WOR}^{\mathrm{abs}}).$$

Here the left-hand side contains one WOR module at each Proposer node, while the right-hand side contains one shared atomic storage service.

ConCRIS lets a module refinement carry a simulation invariant relating the module-local states of the source and target programs. In this proof, the source state is the abstract WOR store `kvmap`, while the target side has no WOR-local state: neither the WOR implementation nor the Proposer specification carries such state. Thus the target local state is unit. Fig. 17 gives the invariant used for this simulation. It existentially chooses a Paxos state map `psmap` and owns the corresponding Proposer resource $\mathsf{PaxosState}(k, \mathsf{psmap}(k))$ for every key k. The pure side condition is the coupling between WOR and Paxos. If $\mathtt{kvmap}(k) = \mathtt{inl}\ S$, then Paxos has no consensus for k and its submitted-value set is exactly S. If $\mathtt{kvmap}(k) = \mathtt{inr}\ v$, then Paxos has consensus on v; the submitted-value set is no longer visible at the WOR level, so the invariant only records that it is some set S.

The useful feature of this setup is that the Proposer has already been replaced by its imaginary specification from Fig. 12. Thus, while proving the WOR refinement, calls to $\mathtt{Pro}_{nid}\mathtt{.prepare}_{nid}$ and $\mathtt{Pro}_{nid}\mathtt{.commit}_{nid}$ can be inlined to their imaginary specifications on the target side and processed as ordinary code.

The proof of `init` is simple. The target calls the Proposer's specification for `init` from Fig. 12, which is inlined. The source and target then expose the same assume and guarantee assertions, so the simulation between them is immediate. Neither side touches the shared storage state related by $\mathcal{I}_{\mathrm{WOR}}$.

For `write`, the proof synchronizes the atomic update to PaxosState in the inlined `commit` specification with the atomic update to `kvmap` in the `write` function of $\mathrm{WOR}^{\mathrm{Abs}}$. The loop first executes `prepare`. Failed prepares are matched by target-side retry steps, and if every attempt fails, the source write takes the nondeterministic failure branch of $\mathrm{WOR}^{\mathrm{Abs}}$. If `prepare` returns `Cons(_)`, the implementation returns without committing; by the invariant the key is already fixed in `kvmap`, so the source write can take the no-op fixed-key branch in sync.

The remaining cases are the branches that call $\mathtt{Pro}_{nid}\mathtt{.commit}_{nid}$. When the inlined Proposer specification atomically updates $\mathsf{PaxosState}(k, ps)$ to some $ns \in \mathtt{next\_states}(ps, w)$, where w is the value actually passed to `commit`, the proof opens $\mathcal{I}_{\mathrm{WOR}}$ and performs the corresponding atomic update to `kvmap` on the source side. If the Paxos update only changes the submitted-value set, the source takes the matching pending update allowed by $\mathrm{WOR}^{\mathrm{Abs}}$. If the Paxos update records consensus on a value, the source chooses the matching fixed value in $\mathrm{WOR}^{\mathrm{Abs}}$. Because the invariant equates pending WOR sets with Paxos submitted-value sets and equates fixed WOR values with Paxos consensus values, the same invariant is restored after both atomic updates.

The proof of `read` needs prophecy reasoning. The atomic point should be the atomic read of PaxosState performed by the inlined `prepare` specification. However, `read` runs this prepare inside a bounded retry loop. The source-side atomic read of `kvmap` should be performed only at the prepare call that will make the WOR read return NoValue or $\mathsf{Value}(v)$. This is where we need prophecy: we need to predict at which iteration the `prepare` result will cause the read to return NoValue or $\mathsf{Value}(v)$.

Specifically, the prophecy predicts how many loop iterations will execute before the successful prepare, or predicts that all attempts will fail to produce a returnable result. With this prediction, the proof knows which prepare call, if any, should be linearized as the single abstract read of $\mathrm{WOR}^{\mathrm{Abs}}$. All earlier iterations are simulated as target-side retries: they may observe `Fail` or `OnlyC(_)`, but they do not perform the source read of `kvmap`. If the prophecy says that no iteration succeeds, the source takes the failing branch of the abstract read and the target loop eventually returns `Failed`.

At the predicted successful iteration, the proof opens $\mathcal{I}_{\mathrm{WOR}}$ exactly when the inlined `prepare` reads $\mathsf{PaxosState}(k, ps)$. The invariant exposes the matching source state $\mathtt{kvmap}(k)$. At this point, the Paxos state alone does not determine the final prepare result: for example, when $ps.2 = \mathtt{Some}(v)$, the Proposer may return either $\mathtt{OnlyC}(v)$, causing WOR to retry, or $\mathtt{Cons}(v)$, causing WOR to return $\mathsf{Value}(v)$. The prophecy resolves this uncertainty by selecting an iteration whose prepare result is returnable. At the predicted returnable iteration, the prepare result can only be `AnyC` or $\mathtt{Cons}(v)$. If the return value from `prepare` is `AnyC`, then $q \in \mathtt{valid\_quorums}(ps)$ implies $ps.2 = \mathtt{None}$, so the invariant gives $\mathtt{kvmap}(k) = \mathtt{inl}\ S$ and the source read can return NoValue. If the return value is $\mathtt{Cons}(v)$, then $q \in \mathtt{valid\_quorums}(ps)$ implies $ps.2 = \mathtt{Some}(v)$, and the invariant gives $\mathtt{kvmap}(k) = \mathtt{inr}\ v$. Therefore the source read can return $\mathsf{Value}(v)$.

Consequently, every behavior of the WOR implementation linked with the Proposer imaginary specification on all Proposer nodes is matched by the single global, logically atomic WOR model. This final abstraction is what lets applications interact with WOR as one shared atomic store, while the refinement accounts for the retry loops and Paxos-level interleavings hidden below.

8 Testing Applications

This section describes our bank application and the two executable testing targets derived from the refinement diagram: the top-level abstract model and the bottom-level implementation. We run the same randomized tests on both and compare how quickly they expose application-level bugs.

Bank Application. Our test application is a replicated bank service built using a general state-machine replication (SMR) library. The SMR library is implemented on top of WOR, using version numbers as WOR keys and deterministic state-transition operations as WOR values. Each SMR node maintains a local version number and the machine state at that version. To perform a requested operation, the node first catches up: it looks up later versions through WOR, one by one, and applies the stored operation at each version to its local state in order. It then tries to store its own transition operation at the next version. If the store fails because another operation has already been stored there, the node catches up again and repeats this process until its operation is stored at the next available version. In this way, all SMR nodes synchronize through the same globally sequenced operations. The bank application instantiates the SMR library with bank commands, `deposit`, `withdraw`, `transfer`, and `balance`, as the transition operations, and with a balance map recording each user's balance as the machine state.

At each proposer node, the bank application runs two threads. One thread receives requests from clients on different nodes through the network, parses the requests, and appends them to a bounded ring buffer in shared memory. The other thread removes requests from this list and processes them using the SMR library. Since this second thread is the only thread that calls into SMR, all underlying WOR operations are issued from one dedicated thread. This matches the simple case in §5: the ownership needed for WOR calls remains in that thread and can be framed across ordinary application code. In our experiment, these routine application-side proof obligations were almost fully automated with agentic AI, Codex 5.5; the auto-generated Rocq proof scripts are checked by the proof checker.

Extraction to Executables. Both endpoints of the refinement diagram in Fig. 6 are executable. The bottom row gives the implementation program: after linking the applications, clients, scheduler, concrete Paxos/WOR stack, physical network model, and deterministic memory, we obtain a closed CRIS module whose behavior is represented by a single interaction tree. The top row gives the abstract model. It uses the same bank and client code, but replaces the verified storage stack with $\mathrm{WOR}^{\mathrm{Abs}}$, nondeterministic memory, and a masked network that leaves Paxos-owned communication abstracted away. This linked abstract CRIS module also gives an interaction tree.

We extract both interaction trees to OCaml with external handlers for the choose events and all I/O events. Specifically, for each **choose**(X) we choose a random value from the uniform distribution on X. For input events, the handler provides randomly generated inputs; for output events, it prints the output values.

Test Setup. We use five bank servers, one for each of the five proposer nodes; five acceptor nodes; and five clients. Each client sends 100 randomly generated requests to randomly selected bank sockets. Each request selects one of `deposit`, `withdraw`, `transfer`, and `balance` with equal probability. Users are selected uniformly from four account names, and amounts are chosen uniformly between 0 and 1000.

Checked Properties. We test two application properties. The first is a simple balance safety property: every user's balance must remain nonnegative. If a negative balance is reached, the check raises an exception. To exercise this property, we deliberately changed both `withdraw` and `transfer` so that they do not check whether the current balance is sufficient.

The second property is subtler: every request received by an application should be handled exactly once, in receive order. The test records, for each application, the unique IDs of received requests and processed requests, both in order. It then checks that the processed-ID sequence is a prefix of the received-ID sequence; if the check fails, it raises an exception. To exercise this property, we deliberately inserted a bug into the ring-buffer insertion code: an incorrect full-buffer check makes both the empty and full states have the same condition, namely `head = tail`. As a result, a full buffer is interpreted as empty, and requests already in the buffer can be lost.

Results. We measure wall-clock time to detect a bug, with a timeout when no bug is found. The results show a large gap between testing the abstract model and testing the full implementation.

For the negative-balance bug, we ran the abstract model 10 times, and every run found a negative balance within 0.2 seconds. The full implementation timed out after 10 minutes without finding the bug.

For the ring-buffer bug, we varied the buffer size and ran the abstract model 10 times for each size. With buffer sizes 2 and 5, every run found the bug within 0.1 seconds. With buffer size 16, every run found the bug within 0.2 seconds. By contrast, a full-implementation run with buffer size 2 timed out after 10 minutes without finding the bug.

The difference is expected from the refinement. The abstract target still runs the same application and client code, but it replaces the Paxos/WOR internals by a single atomic WOR object. Testing the full implementation must additionally explore network nondeterminism and races for Paxos consensus. In our network model, a receive can return any old packet still present at the socket, so old Paxos messages remain available and greatly enlarge the search space. This nondeterminism can prevent randomized testing from quickly reaching bug-exposing application states; in model checking, the same nondeterminism becomes explicit branching and can cause state-space explosion. The abstract model removes that internal Paxos nondeterminism while preserving application behavior. As a result, randomized testing exposes the injected application-level bugs much faster, and model checking starts from a much smaller application-level search space.

9 Related Work and Conclusion

End-to-end verified distributed systems. A large body of work verifies distributed systems with a variety of methodologies. However, these efforts share the methodological premise discussed in §1, namely that every component inside the verified system is brought into a single proof framework. We are not aware of prior work whose end result is an *executable* top-level abstraction that arbitrary, unverified application code can be linked with and tested against, which is the goal of hybrid verification.

IronFleet [7] models distributed systems as network-based state machines and uses Dafny [18] to prove functional correctness, linearizability, and liveness for the Paxos-based IronRSL. Its guarantees apply to the whole verified system, since applications built on IronRSL are part of the Dafny development rather than external concrete code. WormSpace [23] offers a unified WOR-style interface for constructing certified distributed applications in Rocq [3]. This design directly inspires our own WOR layer and top-level model. WormSpace links verified and unverified code at the implementation level and shows that the combination remains practical, but the unverified side receives no formal artifact. Its correctness assumptions about the verified interface are not captured by a theorem. Our refinement, in contrast, makes the interaction discipline of callback registration, WOR call serialization, and socket noninterference an explicit and checkable condition. Verdi [27] lets users write applications against idealized network semantics and transfers their proofs to unreliable networks via verified system transformers, demonstrated on Raft [28]. The application,

however, must itself be written and verified inside Verdi. The transformers relate proof-level models rather than yielding an executable abstraction for external code.

Compositional protocol verification. Disel [21] provides non-atomic, network-based separation-logic semantics for distributed protocols, supporting horizontal composition of verified modules within one logic. Bythos [30] extends this compositional line to Byzantine settings. It derives both safety and liveness of composite BFT protocols from the properties of their sub-protocols in Coq and extracts executable reference implementations. The extracted code is linked against an unverified network shim, but the shim is trusted rather than constrained. The assumptions that the verified protocol places on its unverified environment remain informal. Our refinement instead turns the analogous assumptions about callback behavior, registration discipline, and socket noninterference into formally discharged conditions.

Protocol logics and distributed separation logics. Among automated approaches, Ivy [19] combines state-based modeling with SMT-based reasoning and has verified advanced protocol features such as reconfiguration. Ivy targets protocol models rather than executable service stacks, so the question of surrounding unverified application code does not arise. Aneris [14] is a higher-order distributed separation logic built on Iris supporting modular node-local and thread-local reasoning. Gondelman et al. [4] use it to verify an eventually consistent key-value store. Grove [22] extends Iris and Perennial [2] to verify realistic Go-based distributed systems, including the GroveKV store and its clients. Trillium [25] extends Aneris to intensional refinement between programs and abstract transition-system models and proves that an implementation of single-decree Paxos refines its TLA$^+$ specification. Its refinement, however, runs in the opposite direction from ours. Trillium replaces a program by a mathematical model in order to transport trace properties from the model to the program, and the surrounding code must still be verified inside the logic, whereas we replace a verified stack by an executable abstraction so that surrounding unverified code can be run and tested against it. More broadly, the specifications of these logics are proof-level objects, in that a client benefits from a verified component only by carrying out its own proof in the same logic. Our hybrid imaginary specifications instead provide ownership reasoning where the proof needs it and an executable operational model everywhere else, so unverified applications remain testable concrete code.

Atomic-object abstractions of consensus. ADO [8] and its successors [9, 10, 20] give a unified atomic-object model expressing a family of consensus protocols, with strong support for optimization and reuse of verified building blocks. The ADO model is close in spirit to our atomic WOR$^{\text{abs}}$ abstraction. Both replace distributed consensus by a centralized atomic object. The difference is the intended consumer. ADO's model is a proof component for clients verified in the same framework, while our abstraction is an executable program proved, by conditional contextual refinement, to be a faithful replacement for the implementation under arbitrary concrete applications.

Verification frameworks. Several modular verification frameworks built on interactive theorem provers have been proposed [1, 6, 11, 13, 24]. Among these, CRIS serves as our foundation because its imaginary specifications can mix executable code with ownership assertions, which is exactly what hybrid verification requires. The same specification provides separation-logic reasoning principles during the proof and remains executable in the final model. Our methodology is not strictly tied to CRIS and should apply to any framework with comparable expressiveness.

CCAL [6] is well known for its refinement-based methodology and has demonstrated strong practical utility across major verification efforts [5, 12, 16, 17, 23]. Its contextual refinements are unconditional, so a verified layer cannot state assumptions on its context. The shared-resource dependencies of our setting, such as application-supplied callbacks, shared memory buffers, and

Paxos sockets, fall outside what its layer interfaces express. VST [1] and Iris [11] provide powerful separation-logic reasoning, but their specifications are not executable, so they do not by themselves produce the testing target that hybrid verification aims for. CCR [24] and CRIS [13] add the conditional refinement and imaginary specifications that close this gap, and our work scales them to a distributed consensus stack.

Conclusion. This paper presents the first hybrid verification of a realistic consensus-based distributed service. A Multi-Paxos-based write-once key-value store, verified together with its application-supplied callbacks and registration discipline, is replaced by an executable centralized atomic abstraction while arbitrary application and client code remains concrete. The refinement adapts prophecy and helping reasoning to the hybrid setting and yields an abstraction that is demonstrably a better testing target than the implementation: in our bank application experiments, the abstraction exposes the injected bugs within 0.2 seconds, while the corresponding full-implementation tests time out. This work shows that formal verification can scale to realistic distributed systems in which verified and unverified components coexist and share resources.

References

[1] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press, USA.

[2] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

[3] The Rocq Prover development team. 2025. *The Rocq Prover Reference Manual.* Inria and other contributors, Rocquencourt, France. https://rocq-prover.org/doc/master/ Accessed: 2025-11-14.

[4] Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. *Proc. ACM Program. Lang.* 7, ICFP, Article 217 (Aug. 2023), 31 pages. https://doi.org/10.1145/3607859

[5] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 653–669.

[6] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. *SIGPLAN Not.* 53, 4 (June 2018), 646–661. https://doi.org/10.1145/3296979.3192381

[7] 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)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

[8] 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

[9] Wolf Honoré, Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2024. AdoB: Bridging Benign and Byzantine Consensus with Atomic Distributed Objects. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 109 (April 2024), 30 pages. https://doi.org/10.1145/3649826

[10] Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2022. Adore: atomic distributed objects with certified reconfiguration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 379–394. https://doi.org/10.1145/3519939.3523444

[11] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[12] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. 2017. Safety and Liveness of MCS Lock—Layer by Layer. In *Programming Languages and Systems*, Bor-Yuh Evan Chang (Ed.). Springer International Publishing, Cham, 273–297.

[13] Yonghee Kim, Taeyoung Yoon, Sanghyun Yi, Jaehyung Lee, Soonwon Moon, Yeji Han, Seonho Lee, Taeyoung Rhee, Yujin Im, Donghyun Nam, Jieung Kim, and Chung-Kil Hur. 2026. CRIS: The Power of Imagination in Hybrid Verification. *Proc. ACM Program. Lang.* 10, PLDI, Article 239 (June 2026), 24 pages. https://doi.org/10.1145/3808317

[14] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems: 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings* (Dublin, Ireland). Springer-Verlag, Berlin, Heidelberg, 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

[15] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[16] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3953–3970. https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei

[17] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. 2019. Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. *Proc. ACM Program. Lang.* 4, POPL, Article 20 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371088

[18] Microsoft. 2023. *The Dafny Programming and Verification Language.* https://dafny.org Accessed on 2025-11-14.

[19] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA,

614–630. https://doi.org/10.1145/2908080.2908118

[20] 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. *Proc. ACM Program. Lang.* 8, PLDI, Article 193 (June 2024), 25 pages. https://doi.org/10.1145/3656423

[21] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158116

[22] 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

[23] 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)*. Association for Computing Machinery, New York, NY, USA, 299–311. https://doi.org/10.1145/3357223.3362739

[24] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (Jan. 2023), 31 pages. https://doi.org/10.1145/3571232

[25] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2024. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *Proc. ACM Program. Lang.* 8, POPL, Article 9 (Jan. 2024), 32 pages. https://doi.org/10.1145/3632851

[26] Robbert Van Renesse and Deniz Altınbüken. 2015. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 42:1–42:29.

[27] 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. *SIGPLAN Not.* 50, 6 (June 2015), 357–368. https://doi.org/10.1145/2813885.2737958

[28] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

[29] Taeyoung Yoon, Sanghyun Yi, Yonghee Kim, Jaehyung Lee, Yujin Im, Taeyoung Rhee, Seonho Lee, Yeji Han, and Chung-Kil Hur. 2025. *ConCRIS: Imaginary Specifications for Fine-grained Concurrency*. Technical Report SFLab-2025-002. SFLab. https://sf.snu.ac.kr/technical-reports/files/SFLab-2025-002.pdf

[30] Qiyuan Zhao, George Pîrlea, Karolina Grzeszkiewicz, Seth Gilbert, and Ilya Sergey. 2024. Compositional Verification of Composite Byzantine Protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 34–48. https://doi.org/10.1145/3658644.3690355