Modular Verification of Op-Based CRDTs in Separation Logic

ABEL NIETO, Aarhus University, Denmark
LÉON GONDELMAN, Aarhus University, Denmark
ALBAN REYNAUD, ENS Lyon, France
AMIN TIMANY, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

Operation-based Conflict-free Replicated Data Types (op-based CRDTs) are a family of distributed data structures where all operations are designed to commute, so that replica states eventually converge. Additionally, op-based CRDTs require that operations be propagated between replicas in causal order. This paper presents a framework for verifying safety properties of OCaml op-based CRDT implementations using separation logic. The framework consists of two libraries. One implements a *Reliable Causal Broadcast* (RCB) protocol so that replicas can exchange messages in causal order. A second *OpLib* library then uses RCB to export an interface for building op-based CRDTs that simplifies their creation and correctness proofs. OpLib allows clients to implement new CRDTs as purely-functional data structures, without having to reason about network operations, concurrency control and mutable state, and without having to re-implement causal broadcast each time. Using OpLib, we have implemented 12 example CRDTs from the literature, including multiple versions of replicated registers and sets, two CRDT combinators for products and maps, and two example use cases of the map combinator. Our proofs are conducted in the Aneris distributed separation logic and are formalized in Coq. Our technique is the first work on verification of op-based CRDTs that satisfies both of the following properties: it is *modular* and targets (executable) *implementations*, as opposed to high-level protocols.

CCS Concepts: • **Theory of computation → Program verification**; **Distributed algorithms**; **Separation logic**.

1 INTRODUCTION

To an outside observer, a distributed system ideally appears to function as a single computer, and the fact that the system is composed of multiple collaborating processes is an implementation detail hidden inside the proverbial black box. This behaviour is formally captured by the notion of *linearizability* Herlihy and Wing [1990], which (informally) says that concurrent execution histories of a linearizable data structure can be re-ordered so that operations appear to take place (a) atomically and (b) in a manner that is consistent with the sequential order.

Alas, the CAP[1] theorem [Gilbert and Lynch 2002] shows that, in the presence of network partitions (which are ultimately unavoidable), a system can be either linearizable or available, but not both. *Available* in this context means that the nodes in different network partitions can (independently) continue to service client requests, without waiting for the partitions to heal.

Confronted with this consistency vs availability dilemma, practitioners have developed systems that trade off stronger forms of consistency (e.g. linearizability and sequential consistency) in favour of better availability (e.g. [Bailis et al. 2013; Chang et al. 2008; Chodorow and Dirolf 2010; Lloyd et al. 2011; Sivasubramanian 2012; Tyulenev et al. 2019]). This is possible by adopting weaker consistency models; among such models are *strong eventual consistency* (SEC) [Shapiro et al. 2011b]

[1]Consistency, Availability, Partition tolerance

Authors' addresses: Abel Nieto, Aarhus University, Denmark, abeln@cs.au.dk; Léon Gondelman, Aarhus University, Denmark, gondelman@cs.au.dk; Alban Reynaud, ENS Lyon, France, alban.reynaud@ens-lyon.fr; Amin Timany, Aarhus University, Denmark, timany@cs.au.dk; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

2022. 0004-5411/2022/8-ART111 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and *causal consistency* [Ahamad et al. 1995]. For example, in SEC two processes that read from a replicated register might observe different values even though no intervening writes have occurred (something not possible when reading from sequentially-consistent local memory from within a process). Eventually, however, the state of the replicated register at different replicas must converge. More precisely, SEC requires the following two properties (note the first is a liveness property while the latter is a safety property):

- *(Eventual Delivery)* An update delivered to a correct replica is eventually delivered to all replicas.
- *(Convergence)* Replicas that have delivered the same updates eventually reach equivalent states.

Conflict-free Replicated Datatypes (CRDTs) [Shapiro et al. 2011a] are a class of distributed systems where a data structure (e.g. register, set, or map) is replicated over multiple replicas that mutate its state via local operations. Because replicas are allowed to invoke operations without coordinating with others, different replicas might arrive at conflicting states. CRDTs resolve such conflicts automatically. There are two main ways of going about this. One option is to model the replica state as a (join) semilattice, so that merges are accomplished by taking least upper bounds (joins); these are *state-based* or *convergent* CRDTs. Changes are then propagated by sending the entire state to other replicas on the (possibly unreliable) network. Another option is to propagate, instead of the entire state, just the effect of each individual update. It becomes then necessary to enforce that each operation is executed exactly once (at most once for the convergence and at least one for the eventual delivery properties above), which typically requires broadcasting primitives that offer reliable delivery. Furthermore, it is also necessary to enforce that some or all operations commute so that concurrent operations can be applied in any order. This last class, known as operation-based (*op-based*) or *commutative* CRDTs, is the focus of this paper[2].

Consider the following example of a counter data structure replicated over two nodes A and B:

```
(* Node A *)              (* Node B *)
add 1; add 200            add 2; let v = read () in assert((v = 2) || (v = 3) || (v = 203))
```

The counter exports two operations: `add(z)`, which adds an integer `z` to the counter, and `read()`, which returns the counter's current value. This CRDT is known as a *positive-negative counter* (PN-Counter)[Shapiro et al. 2011a].

One question of interest for the example above is what are the possible values of `v`. Because the counter should remain available even if A and B are partitioned, A's `add(1)` should execute without trying to synchronize with B. This means that A's and B's `add` operations potentially happen concurrently. By contrast, when A's two operations are broadcast to B, they should be applied by B following A's program order. Finally, when B reads, we do not know whether A's updates have been received, but we do expect that the `add(2)` has been recorded locally. This means that the possible values for `v` are `2` (only the local `add` has been applied), `3` (only the A's first `add` has been applied), and `203` (all `add` have been applied). Results like `0`, `200` and `202` are not valid answers.

Causal Delivery. Our intuitions about valid execution traces in the example above can be captured by a *happens-before* or *causality* relation on events [Lamport 1978]. Let a and b be two events (possibly taking place at different processes). Then a *happens before* b (and b is *causally dependent* on a), written $a \longrightarrow b$, if one of the following holds:

- a and b take place in the same process, and $a < b$ according to *program order*.
- a is the event of sending a message m and b is the corresponding event where m is received.
- $a \longrightarrow c$ and $c \longrightarrow b$ for some other event c (the transitive closure of the above two rules).

[2]From now on whenever we use the term *CRDT* the reader can safely assume that we mean *op-based* CRDT, unless explicitly noted otherwise.

If neither $a \longrightarrow b$ nor $b \longrightarrow a$, then we say they are *concurrent*, written $a||b$. Informally, we say that events are *causally delivered* if the following property holds: if an event e is delivered[3] to a replica p, then all events on which e causally depends must have been previously delivered to p. We can then require that valid PN-counter execution traces satisfy causal delivery of operations. Indeed, this is a common requirement for many CRDTs in the literature [Baquero et al. 2014].

Reliable Causal Broadcast. One way to realize the guarantees of causal delivery is to implement a one-to-many communication protocol known as *Reliable Causal Broadcast* (RCB) [Cachin et al. 2011]. In RCB, a group of N replicas send each other messages. The protocol's interface consists of two functions: broadcast(msg), which sends message msg to all other $N - 1$ replicas, and deliver(), which returns a received message (if one exists) while respecting causal order.

1.1 Contributions

Because CRDTs are data structures replicated across multiple processes, each of which is allowed to reorder concurrent operations, they are challenging to specify and verify.

The main property of interest for verification is SEC [Shapiro et al. 2011b] which as we mentioned can be divided into convergence and eventual delivery [4]. However, convergence does not say how the CRDT's final state is computed from the set of received operations. Burckhardt et al. [2014a] addressed this question by showing how to give *functional correctness* specifications for CRDTs. Another consideration is whether the verified properties can be reused by components other than the CRDT: that is, whether the verification technique is *modular*. The recent work of Liang and Feng [2021] presents the first modular verification technique for op-based CRDTs.

An additional design decision is the level of detail at which to model the CRDT that is the target of verification. There are roughly two options: one can model the CRDT as a high-level protocol, perhaps assuming that the network is reliable or ignoring node-local concurrency. Alternatively, we can implement the CRDT in a general-purpose programming language where we have to deal with a plethora of low-level (but realistic) details such as an unreliable network, concurrency-control, and mutation.

Our work. This paper is about proving SEC and functional correctness of op-based CRDTs. To the best of our knowledge, all prior work on verification of op-based CRDTs consists of techniques that produce modular specifications but work at the protocol level, or techniques that work for implementations but are non-modular (see Section 7 for a classification of prior work). The main contribution of our work is to lift that restriction: we can produce *modular* specifications of CRDT *implementations*. Additionally, unlike prior work which assumes causal delivery by the network, our CRDTs include a general-purpose implementation of reliable causal broadcast. All our proofs are mechanized in Coq. More precisely, the contributions of this work are as follows:

(1) We implemented and verified an RcbLib library for reliable causal broadcast (RCB). To the best our knowledge, this is the first time a formalization of op-based CRDTs includes a general-purpose implementation of RCB, as opposed to assuming causal broadcast.
(2) On top of the RcbLib library, we implemented and verified an OpLib library for making op-based CRDTs. Using OpLib, one can create an op-based CRDTs as purely-functional data structures, without having to reason about low-level details like mutation, concurrency control, and network operations. Similarly, by proving only simple sequential specifications,

[3]Delivery occurs when the event processing layer makes its clients aware of the event; this can take different forms depending on the specific application.
[4]The terminology is not universal: Shapiro et al. [2011a] refers to both properties together as *eventual convergence.*

OpLib users obtain from the library rich specifications for their CRDTs, enabling modular reasoning about convergence, causality, and functional correctness.

(3) We evaluated OpLib by implementing a collection of 12 CRDTs, including multiple versions of registers and sets, as well as two combinators for products and maps. We further evaluated the modularity of our specifications by verifying a client program that uses a CRDT obtained via OpLib.

(4) We wrote our libraries in a subset of OCaml that is then automatically translated to AnerisLang, the programming language of the Aneris [Krogh-Jespersen et al. 2020] distributed separation logic. Our proofs were conducted in Aneris and are mechanized in Coq.

Structure of the paper. The rest of the paper is organized as follows: Section 2 gives a quick primer to Iris and Aneris program logics. Section 3 provides an overview of the keys ideas of our work and presents the concepts that CRDT implementer need to use our libraries. Section 4 describes in more detail RcbLib's implementation and correctness proof. Section 5 then does the same for OpLib. Section 6 discusses our case studies (the implemented CRDTs). We then take a look at prior work on Section 7, and conclude in Section 8.

2 ANERIS PRIMER

Iris [Jung et al. 2018] is a state-of-the-art program logic designed to reason about concurrent programs based on separation logic. Aneris [Krogh-Jespersen et al. 2020] is a program logic built on top of Iris for reasoning about distributed systems. Figure 1 shows the fragment of Iris and Aneris logic that we need in this paper:

$$\begin{array}{lll}
P, Q \in iProp ::= \mathsf{True} \mid \mathsf{False} \mid P \land Q \mid P \Rightarrow Q \mid P \lor Q \mid \forall x.\, P \mid \exists x.\, P \mid \cdots & & \text{higher-order logic} \\
\qquad\qquad \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto_{ip} v \mid \{P\}\ \langle ip; e\rangle\ \{x.\, Q\} \mid \Box P & & \text{separation logic} \\
\qquad\qquad \mid \boxed{P}^{\mathcal{N}} \mid {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} & & \text{Iris resources and invariants}
\end{array}$$

Fig. 1. The fragment of Iris and Aneris relevant to this paper

First and foremost Iris is a higher-order logic with the usual connectives. Note how we can quantify, both existentially and universally, over any domain, including *iProp* itself (we write *iProp* for the universe of Iris propositions). Iris is a separation logic. Iris propositions can assert ownership of resources and express their *disjointness*. The proposition $P * Q$ holds if the owned resources can be split into two disjoint parts where one satisfies P and the other Q. The magic wand, $P \mathrel{-\!\!*} Q$, also called separating implication, asserts ownership over resources that when combined with (disjoint) resources satisfying P would satisfy Q. The so-called points-to proposition, $\ell \mapsto_{ip} v$, asserts exclusive ownership over the memory location ℓ stating that the value stored in this location is v. This proposition differs from the standard separation logic points-to proposition only in that it is annotated with the *Ip* address of the node to which it belongs — this is necessary as we are working with a distributed system in Aneris. Similarly, in Aneris a Hoare-triple $\{P\}\ \langle ip; e\rangle\ \{x.\, Q\}$, in addition to the program, also takes the *Ip* address of the node the program is running on.

The persistently modality, $\Box$, captures duplicability of propositions. It allows us to distinguish between propositions that are duplicable and those that are not, *e.g.*, points-to propositions: $\ell \mapsto_{ip} v * \ell \mapsto_{ip} w \vdash \mathsf{False}$. Here, $\vdash$ is the logical entailment relation of Iris. Intuitively, $\Box P$ holds if P does and furthermore, P does not assert ownership of any non-duplicable resources. We say a proposition is persistent if $P \vdash \Box P$; note that for any proposition P we always have $\Box P \vdash P$. Persistent propositions are duplicable, *i.e.*, $\Box P \vdash \Box P * \Box P$, and hence they merely express knowledge as opposed to expressing (exclusive) ownership over resources. An example of a persistent proposition is Iris invariants. The invariant $\boxed{P}^{\mathcal{N}}$ asserts that P must hold at all times throughout program execution. Hence, throughout a proof, for the duration of an atomic step of computation, we can access invariants, *i.e.*, we get to know that the invariant holds before the step of computation and need to guarantee that it also holds afterwards. The name of the invariant $\mathcal{N}$ is used to track accesses to invariants and prevent them from being accessed in an unsound manner, *e.g.*, accessing the same invariant twice during the same atomic step of computation which could result in duplicates of non-duplicable propositions like the points-to proposition. The update modality[5], ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}$, allows manipulation of invariants and resources in Iris. The masks $\mathcal{E}_1$ and $\mathcal{E}_2$ are sets of invariant names and respectively indicate which invariants hold before and after the "update" takes place. We write ${\Rrightarrow}_{\mathcal{E}}$ for ${}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}$. The update modality is the primary way of working with invariants in Iris. They are used in the definition of Iris Hoare-triples in such a way as to enforce the aforementioned invariant policy of only allowing access to invariants during atomic steps of computation. Intuitively, the proposition ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ holds if we can manipulate resources (allocate new resources, or update the existing ones) and manipulate invariants (create new invariants, access invariants, or reestablish invariants) so as to make sure that P holds. Furthermore, during this update we can access all invariants in $\mathcal{E}_1$ but must ensure that all invariants in $\mathcal{E}_2$ hold after the update is done.

3 MAIN IDEAS

This section provides a birds-eye view of the paper, focusing on concepts users need to use our libraries. Figure 2 shows an overview of our work. We structured our development as a collection of libraries, each exporting a modular specification.

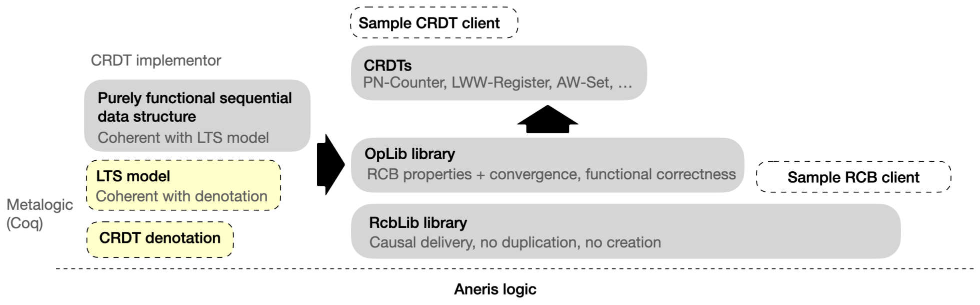

Fig. 2. Verified tower of components

Higher-level libraries can then be verified using solely the specifications of its dependencies, without knowledge of the dependency's implementation. Each box lists a library name and the safety properties guaranteed by its specification. Grey boxes correspond to OCaml libraries[6]; yellow boxes are written in Coq.

3.1 RcbLib

At the base of our verified tower of components we have a library implementing a reliable causal broadcast protocol [Cachin et al. 2011]. This library is built on top of UDP, so it makes minimal assumptions about network guarantees. In particular, messages can be dropped, re-ordered, and

[5]In Iris jargon this modality is called the fancy update modality; see Jung et al. [2018] for more details.
[6]Later automatically translated to AnerisLang, the programming language of the Aneris distributed separation logic.

duplicated by the network. The library deploys a suite of techniques, such as sequence ids, acknowledgments, retransmissions, and a delay queue, (see Section 4), to offer three main guarantees: broadcast messages are delivered in causal order, without duplicates, and ensuring that any message delivered was previously broadcast by another participant (the "no creation" property in Figure 2). These are the three safety properties of RCB [Cachin et al. 2011].

Verifying RcbLib. The main idea for verifying RcbLib is to generalize the treatment of causality in Gondelman et al. [2021] to the causal broadcast setting. We now briefly outline our approach and expand on it in Section 4.

The first step is to define separation logic resources tracking the set of broadcast messages between replicas in two ways: the $\mathsf{OwnGlobal}(h)$ resource provides a *global view* tracking the set h of all messages broadcast by any replica, while the $\mathsf{OwnLocal}(i, s)$ resource provides a *local view* tracking the set s of all messages that has been delivered by replica i. Here, messages are triples $(\mathrm{p}, \mathrm{vc}, \mathrm{o})$ consisting of the message's payload p, vector clock vc, and id of the originating replica p.

The next step is to craft separation logic specifications for RcbLib's broadcast and deliver functions. Below, we show a simplified specification for broadcast (see Section 4 for the full specs):

$$\begin{array}{l}\{\mathsf{OwnGlobal}(h) * \mathsf{OwnLocal}(i, s)\} \\ \quad \langle ip_i; \mathsf{broadcast}(p)\rangle \\ \{m.\, payload(m) = p * \mathsf{OwnGlobal}(h \uplus \{m\}) * \mathsf{OwnLocal}(i, s \uplus \{m\})\}\end{array}$$

This spec states that in order to broadcast a message with payload p, we need to provide both the global view and the local view of the broadcasting replica. broadcast can then execute without errors and return a message m with payload p. Logically, we know that the global set of broadcast messages now includes m, and also that node i has delivered (is aware of) the new message.

In addition to the broadcast and deliver specifications, following Gondelman et al. [2021] we provide to the user of RcbLib a set of laws governing the above resources. Notably, the causality law states that, given the ownership of $\mathsf{OwnGlobal}(h)$ and $\mathsf{OwnLocal}(i, s)$, we can conclude that

$$\forall m \in s, m' \in h.\ \mathrm{vc}(m') < \mathrm{vc}(m) \Rightarrow m' \in s$$

i.e., for any message m that has been delivered at node i, if we know of another message m' that has been broadcast by any other node such that m' happened before m[7], then it must be the case that node i has previously delivered m' as well. All laws are proven in Coq and provided as lemmas.

3.2 OpLib

Conceptually, an op-based CRDT implementation can be seen as an infinite loop that maintains the CRDT's state at a given replica. This loop has a number of responsibilities:

(1) accept *local* operations invoked by the user at the replica
(2) modify the CRDT's state as per the effects of local operations
(3) propagate local operations to other replicas
(4) listen for *remote* operations communicated via the network
(5) modify the CRDT's state as per the effects of remote operations

One can then observe that there are a number of derived responsibilities that flow from the ones above: for example, since steps (2) and (5) can happen concurrently, some form of concurrency control (e.g. locking) is needed. Additionally, because the network is unreliable, step (3) requires that the CRDT is be able to tolerate dropped messages. Another observation is that most of the

[7] $\mathrm{vc}(m)$ stands for m's *vector clock*, a mechanism for tracking causal dependencies. See Section 4.1 for details.

steps above are agnostic to the semantics of the specific CRDT: only when modifying the CRDT's state (steps (2) and (5)) do we need to know the inner workings of the data type's operations.

These observations suggest a design where the generic responsibilities are factored out as a library that is parametric on the CRDT's operations and their effects. Inspired by the approach in Baquero et al. [2014], we instantiate a CRDT via the OpLib library that we have implemented on top of the RcbLib. In our library, all that the user needs to provide is the data type's *initial state* and an *effect* function that can process new operations. This design allows a CRDT implementer to focus on the core logic of their data type as a purely-functional data structure, while delegating to OpLib all the gritty details of inter-replica communication, concurrency control, and mutation. Because OpLib uses RcbLib for propagating operations between replicas, clients can rely on the guarantees of causal broadcast. Once instantiated with the user's purely functional data type, OpLib turns it into a fully-fledged CRDT that exports two functions: get_state(), which returns (a copy of) the CRDT's current state, and update(op) which updates the state via a new operation *op*.

Verifying OpLib. To verify OpLib we adapt the notion of CRDT *denotations* [Burckhardt et al. 2014a; Leijnse et al. 2019] to separation logic. A CRDT denotation $[\![\cdot]\!] : 2^{Msg} \rightharpoonup St$ is a (partial) function from sets of messages (a message contains an operation plus causality metadata) to the CRDT state that results from executing said operations. Both Msg and St vary depending on the specific CRDT. For example, the denotation for a PN-Counter is a function that maps a set of messages to the sum of its payloads: $[\![s]\!] = \sum_{m \in s} payload(m)$.

Denotations have been previously used to give high-level specifications for CRDTs as well as CRDT combinators (e.g. products of CRDTs and maps where the value type is an arbitrary CRDT) [Burckhardt et al. 2014a; Leijnse et al. 2019]. However, those works do not use denotations to verify implementations. We adapt denotations by constructing a separation logic resource $\mathsf{LocSt}(i, \bullet\, s, \circ\, r)$[8] which tracks the sets s and r of local and remote operations, respectively, processed by replica i. The key insight behind the resource $\mathsf{LocSt}(i, \bullet\, s, \circ\, n)$ is that it tracks *precisely* the set of processed local operations s, but provides only *a lower bound* on the set of processed remote operations. This captures the intuition that while a CRDT user can control which local operations they perform, they do not know which additional remote operations have been propagated from other replicas at a given moment in time. The simplified spec for get_state below shows how the resource is used:

$$\{\mathsf{LocSt}(i, \bullet\, s, \circ\, r)\}\ \mathsf{get\_state}()\ \{m. \exists r', r \subseteq r' * m = [\![s \cup r']\!] * \mathsf{LocSt}(i, \bullet\, s, \circ\, r')\}$$

The spec says that prior to calling get_state we must know that replica i has processed exactly the local messages in s, and at least the remote messages in r. The function then returns a state m that is the denotation of the set $s \cup r'$, where r' is a superset of r. This is because in between calls to get_state the CRDT might have processed additional remote operations.

3.3 CRDT Instances

The last element of Figure 2 we highlight is the recipe that CRDT implementer follow to use OpLib:

- First, the CRDT implementer must provide a denotation for their CRDT.
- In order to bridge the abstraction gap between the denotation, stated in terms of the sets of operations, and the effect function, which must process one operation at a time, the user provides a second specification in the form of a *labelled-transition system* (LTS). In this LTS, states are the CRDT's states and the transitions are labelled with operations. That is, a transition $s \xrightarrow{op} s'$ means that if the CRDT is in state s and an operation op is received, then it

[8]The notation is reminiscent of the so-called authoritative resource algebra [Jung et al. 2018].

will end up in state s'. Importantly, the denotation and LTS must agree in the following sense: if h is a set of operations such that $[\![h]\!] = s$, and $s \xrightarrow{op} s'$, then we must have $[\![h \cup \{op\}]\!] = s'$.
- Finally, the user shows that their effect function "implements" the LTS via a Hoare triple.

The first two steps are conducted outside separation logic in the meta-logic (Coq), while the last step requires proving a Hoare in within Aneris program logic.

We have followed the recipe above to implement 12 CRDTs, including multiple kinds of registers and sets, as well as two CRDT combinators for products and maps. Our combinators use Coq typeclasses as in Liu et al. [2020] to automatically generate and prove correctness of compound CRDTs from constituent CRDTs.

Our examples match those in literature Baquero et al. [2014]; Leijnse et al. [2019]; Shapiro et al. [2011a]). Importantly, they include CRDTs where all operations naturally commute (e.g. PN-Counter) as well as others that require causality information to make operations commutative (e.g. Last-Writer-Wins Register and Add-Wins Set).

4 RELIABLE CAUSAL BROADCAST

The network primitives (send and receive) provided by AnerisLang are for *point-to-point* communication: that is, through them a process communicates with one other process. They are also, as previously mentioned, unreliable in a number of ways: messages can get lost, duplicated, and re-ordered in transit.

A useful abstraction in distributed systems is that of *broadcast*. In broadcast, or *one-to-many* communication, a process transmits the same message to one or more other processes. There exist different broadcast algorithms providing different guarantees: one such kind is *reliable causal broacast* (RCB). In RCB, clients are provided with two operations, broadcast(msg) and deliver() that satisfy the following properties (taken from Cachin et al. [2011] and classified as either liveness or safety properties):

- (RCB1, liveness) *Validity*: If a correct process p broadcasts a message m, then p eventually delivers m.
- (RCB2, **safety**) *No duplication*: No message is delivered more than once.
- (RCB3, **safety**): *No creation*: If a process delivers a message m with sender s, then m was previously broadcast by process s.
- (RCB4, liveness): *Agreement*: If a message m is delivered by some correct process, then m is eventually delivered by every correct process.
- (RCB5, **safety**): *Causal delivery*: For any message m_1 that potentially caused a message m_2, i.e., $m_1 \longrightarrow m_2$, no process delivers m_2 unless it has already delivered m_1.

In this section, we sketch our implementation of a library for RCB, OpLib, based on Birman et al. [1991] and Baquero et al. [2014]. We proved specifications of our implementation that satisfy the three safety properties above. In fact, our RCB library implements a slightly stronger specification than regular RCB, because it exposes to its clients causality information associated to messages in the form of vector clocks. The additional information provided by this *tagged* form of RCB [Baquero et al. 2014] simplifies the task of building CRDTs using OpLib (see Section 6).

4.1 Implementation

Since AnerisLang's network primitives provide few guarantees, RcbLib deploys a few different techniques in order to achieve the safety properties just mentioned. Some of the challenges and their solutions are outlined in Table 1. Additionally, Figure 3 provides a high-level view of the design of the RCB algorithm. The main components are outlined below.

Challenge	**Technique**
Messages can be dropped, reordered and duplicated by the network.	Stop-and-wait protocol [Tanenbaum and van Steen 2007] using sequence ids, acknowledgments, and retransmissions to handle unreliable network.
The broadcasting process can be partitioned from the network before all processes receive a broadcast.	Eager reliable broadcast (retransmissions) [Cachin et al. 2011].
Messages need to be delivered in causal order.	Delay delivery of messages until causal dependencies are satisfied, using a *delay queue* and *vector clocks* [Birman et al. 1991].

Table 1. Challenges and techniques employed in RCB's implementation

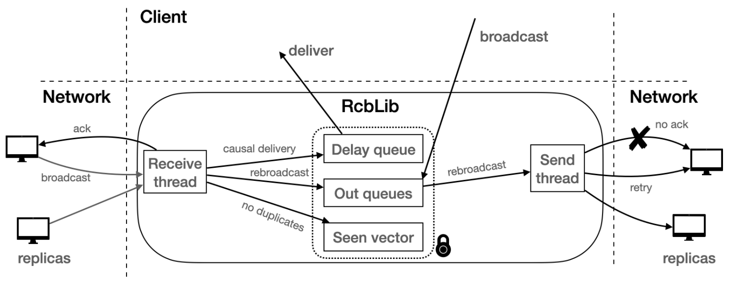

Fig. 3. Reliable causal broadcast overview

Receive and send threads. RcbLib consists of two concurrent threads that operate on a set of shared data structures (concurrent accesses are synchronized via a lock). The *receive* thread listens for messages on a network socket and places them in a *delay queue* and a collection of *out-queues*. It also acknowledges received messages so other replicas can move on to broadcasting new messages.

The *send thread* sends the messages in the out-queues to other replicas following a *stop-and-wait* protocol [Tanenbaum and van Steen 2007]. That is, a message is repeatedly sent to another replica until it is acknowledged by the foreign replica; at which point the send thread pops the relevant out-queue and moves on to a not-yet-acknowledged message.

Library API. The library has two client APIs: broadcast and deliver. The former removes a message m from the delay queue such that all of the message's causal dependencies have previously been delivered (i.e. a message that comes next according to causal order). If no such message exists, deliver returns None; otherwise it returns Some(m). The broadcast function broadcasts a message to all replicas (except to the current one). It does so by placing the message in all out-queues, so it can be later picked up by the send thread. broadcast(p) returns a new message m' containing the payload p together with the vector clock assigned to m' and the issuing replica's id. Because a replica doesn't broadcast to itself it must use the return value of broadcast if it wants to process the newly-broadcast message m'.

Vector clocks. We use vector clocks to keep track of logical time [Fidge 1987; Mattern et al. 1988]. A vector clock is an array of non-negative integers. There is one array entry per replica in the system, and each entry records the number of events that originate at the corresponding replica. It is possible to merge two vector clocks by by taking the maximum of the entries pointwise. We can define a partial order $\leq_{vc}$ on vector clocks by lifting $\leq$ (from $\mathbb{N}$) pointwise. The following result then holds: let a and b be events. then $a \longrightarrow b$ iff $vc(a) < vc(b)$.

Replicas maintain internal state with their current vector clock. Every sent message m is also tagged with a vector clock $vc(m)$. When broadcast is called, the replica increments its entry within the internal vector clock and tags the event with it. When the receive thread receives a new message, its vector clock is not immediately merged with the replica's vector clock; instead, the merge is delayed while the message waits in the delay queue.

Delay queue. In order to ensure causal delivery of messages, RCB stores messages received from other processes in a delay queue. That is, we do not deliver received messages immediately to the user. Given the internal vector clock v_i and a message m from the delay queue, we can determine whether (a) all causal dependencies of the message have been previously delivered and (b) the message has not been previously delivered. We do this using the following *delivery condition* [Birman et al. 1991];

$$\text{canDeliver}(m, v_i) \triangleq \forall k \in \{1 \ldots n\} \begin{cases} vc(m)[k] = v_i[k] + 1 & \text{if } k = origin(m) \\ vc(m)[k] \leq v_i[k] & \text{otherwise} \end{cases}$$

Once the delivery condition for m is met, it is safe (causally consistent) to deliver m to the user in the next invocation of deliver. At that point, the internal vector clock v_i can be updated by merging it with $vc(m)$.

Out queues. Consider the following scenario. There are three processes A, B and C. A broadcasts a message m to B and C. After A has sent m to B, but before it has a chance to send it to C, the network becomes partitioned into two partitions $\{A\}$ and $\{B, C\}$. Now B receives m, but C will not receive m until the partition is healed. This violates property RCB4 of above because the partition might never heal, so C might never get m. Additionally, suppose that B creates a new message m', which now is causally dependent on m: $m \longrightarrow m'$. Even though B and C are in the same partition, C cannot deliver m' until it delivers m first (a causal dependency). The whole system is stuck because one process is partitioned.

For this reason, RCB implements a form of *eager reliable broadcast* [Cachin et al. 2011]. That is, every process re-broadcasts every single message received to every other process (taking care to not enter into loops). Eager rebroadcasting is inefficient, since for every message sent there are $O(n^2)$ re-broadcasts in a system with n replicas (as opposed to $O(n)$, which is the best case for broadcast). We have chosen this mechanism for the first iteration of the RCB library due to its simplicity.

Given the need to re-broadcast messages, and because the network is unreliable, each process maintains a set of *out queues*, one per other process in the system (so n queues per node). Each queue contains the outbound messages that need to be sent to a specific process, but have not yet been acknowledged by that process. Messages are copied from the delay queue to the out queues, and are removed from the out queues when acknowledged by the intended recipient.

Seen vector. A message could be received multiple times by the same process: because the network generated a duplicate or the message was re-broadcast multiple times by other processes. In either case, we need a mechanism to avoid re-delivery of the same message; in other words, we need to avoid putting the same message twice in the delay queue. To this effect, we use vector clocks as sequence identifiers. Given a message m, the pair $(origin(m), vc(m)[origin(m)])$ uniquely identifies a message in the system. We can then construct a *seen vector* where the ith entry gives us the highest sequence id of a message originating from process i that has been previously received. We only place a message originating at process i in the delay queue if its sequence id is higher (by one) than the current value of seen$[i]$.

$$
\begin{array}{l}
\textsc{DeliverSpec}\\
\langle \mathsf{OwnLocal}(i,s)\rangle\\
\quad\langle ip_i;\ \mathsf{deliver}()\rangle\\
\left\langle\begin{array}{l}
v.\ \exists s' \supseteq s.\ \mathsf{OwnLocal}(i,s') \ast\\
\quad\Big((v = \mathsf{None} \land s' = s) \lor\\
\quad(\exists w, a.\ v = \mathsf{Some}(w) \ast \mathsf{IsLocEv}(a,w) \ast\\
\quad\ s' = s \cup \{a\} \ast a \notin s \ast\\
\quad\ a \in \mathrm{Maximals}(s') \ast \mathit{origin}(a) \neq i \ast\\
\quad\ \mathsf{OwnGlobalSnapshot}(\{\lfloor a \rfloor\}))\Big)
\end{array}\right\rangle_{\mathcal{N}}
\end{array}
\qquad
\begin{array}{l}
\textsc{BroadcastSpec}\\
\langle \mathsf{OwnLocal}(i,s) \ast \mathsf{OwnGlobal}(h)\rangle\\
\quad\langle ip_i;\ \mathsf{broadcast}(v)\rangle\\
\left\langle\begin{array}{l}
w.\ \exists a.\ \mathsf{IsLocEv}(a,w) \ast a \notin s \ast \lfloor a \rfloor \notin h \ast\\
\quad \mathit{payload}(a) = v \ast \mathit{origin}(a) = i \ast\\
\quad a \in \mathrm{Maximals}(h \cup \{\lfloor a \rfloor\}) \ast\\
\quad a \in \mathrm{Maximum}(s \cup \{a\}) \ast\\
\quad \mathsf{OwnLocal}(i, s \cup \{a\}) \ast\\
\quad \mathsf{OwnGlobal}(h \cup \{\lfloor a \rfloor\})
\end{array}\right\rangle_{\mathcal{N}}
\end{array}
$$

Fig. 4. Logically-atomic specifications for deliver and broadcast . $\mathcal{N}$ is any namespace containing the global invariant's name.

4.2 Specification

As mentioned in Section 3, the specifications for deliver and broadcast (shown in Figure 4) use separation logic resources that keep track of the local and global states of the broadcast. The local resource $\mathsf{OwnLocal}(i,s)$ tells us that in process i RcbLib has previously delivered exactly the messages in s. Similarly, the global resource $\mathsf{OwnGlobal}(h)$ implies that h is exactly the set of messages that have been broadcast by any replica. We also maintain a *global invariant* RcbInv that ensures that global and local states are compatible. The invariant states that at all times if we combine all local states we obtain the global states, and furthermore that the local states satisfy causal delivery.

Deliver. Figure 4 shows the specification of RCB's deliver function. The intuition is that before calling deliver we should know which messages have been previously delivered at this process (via ownership of a resource $\mathsf{OwnLocal}(i,s)$). After deliver returns, there are two possibilities:

- No messages were available for delivery, so the function returns None, and we get back our unchanged $\mathsf{OwnLocal}(i,s)$.
- There was a message a available for delivery. In this case, the function returns Some(w), where w is the physical counterpart to a, reflected by the predicate $\mathsf{IsLocEv}(a,w)$. Additionally, we receive back a resource $\mathsf{OwnLocal}(i, s \cup \{a\})$. That is, we logically record the delivery of the new message. Crucially, we know that $a \notin s$, meaning that the returned message has not been previously delivered. Additionally, we get to know that a is *maximal* with respect to vector clock order in the set $s \cup \{a\}$. This means that no previously-received message could causally depend on a (but a can depend on previous messages). Finally, we obtain the resource $\mathsf{OwnGlobalSnapshot}(\{\lfloor a \rfloor\})$[9], which serves as proof that the returned message $\lfloor a \rfloor$ did not "come out of thin air": it was properly recorded in the global state. In general, owning a *global snapshot* $\mathsf{OwnGlobalSnapshot}(r)$ gives us a lower bound r on the set of all messages sent: if we own both $\mathsf{OwnGlobal}(h)$ and $\mathsf{OwnGlobalSnapshot}(r)$ we can conclude $r \subseteq h$. The $\mathsf{OwnGlobalSnapshot}(r)$ resource is persistent (Section 2), meaning that we can make copies of it freely; this makes snapshots useful as certificates that a certain message was broadcast by RCB.

[9]The notation $\lfloor a \rfloor$ stands for the *erasure* of a. This is a technical detail we inherited from the development in Gondelman et al. [2021], because we represent local and global events differently. The erasure of a local event a gives us the corresponding global event $\lfloor a \rfloor$.

Broadcast. Figure 4 also shows the specification of broadcast. Intuitively, the effect of broadcast is to generate a new message, which in our framework needs to be recorded both as part of the global state as well as of the local state of the process calling broadcast. This is why in the precondition of broadcast we need to provide both $\mathsf{OwnGlobal}(h)$ and $\mathsf{OwnLocal}(i, s)$. The function then returns a local event w and its logical representation a, as evidenced by the predicate $\mathsf{IsLocEv}(w, a)$. A few points worth pointing out:

- Unlike in traditional implementations of RCB, where broadcast returns unit, our broadcast returns the generated message (or local event) corresponding to the broadcast value. For example, if replica i broadcasts the value 2, then $\mathsf{broadcast}(2)$ returns a tuple $(2, \mathrm{vc}, i)$ for some vector clock vc that is globally maximal. In general, the return value is of the form $(\mathrm{payload}, \mathrm{vc}, \mathrm{origin})$. This is why we call our implementation *tagged* RCB, as per Baquero et al. [2014].
- As expected, the newly generated has not been previously recorded. This is given by $a \notin s$ and $\lfloor a \rfloor \notin h$.
- We obtain back resources $\mathsf{OwnGlobal}(h \cup \{\lfloor a \rfloor\})$ and $\mathsf{OwnLocal}(i, s \cup \{a\})$, showing that the event is been properly recorded both locally and globally.

Logical Atomicity. The observant reader might have noticed two peculiar points about the specs above.

First, the broadcast spec requires the user to provide the global state resource $\mathsf{OwnGlobal}(h)$. Separation logic is all about modular specification, so a global resource that tracks all broadcast events would seem to be antithetical to separation logic. However, we find that the global resource is useful when reasoning about closed programs, because it allows us to state invariants of the form "all messages ever sent satisfy a safety property P" (e.g. in a system with two replicas, all messages are sent by one of the replicas).

A more practical concern is how to get two processes to concurrently broadcast messages, since it would seem that the broadcast spec requires exclusive ownership of $\mathsf{OwnGlobal}(h)$; it in fact does not. The reason is that our specs do not use regular Hoare triples, but instead rely on *logically-atomic triples* [Jung et al. 2015]. Instead of the regular $\{P\}\ e\ \{Q\}$ we write $\langle P \rangle e \langle Q \rangle_{\mathcal{N}}$. The intuition is the following: if we can prove the atomic triple above, then e is evaluated until a certain step (its *linearization point* [Herlihy and Wing 1990]) at which point P holds, possibly after opening any invariant that is not in the $\mathcal{N}$ namespace. After the atomic step, Q then holds, and all opened invariants need to be closed. So Q does not necessarily hold when the function terminates, but it always holds after the linearization point. The advantage of atomic triples is that we are allowed to open invariants when proving the precondition P. This is useful in the broadcast spec, because the global resource $\mathsf{OwnGlobal}(h)$ is likely to be kept in an Iris invariant by most clients of RCB (otherwise clients will not be able to concurrently broadcast messages). Our definition of atomic triples is adapted from that in Perennial [Chajed et al. 2019].

Resource lemmas. As mentioned in Section 3.1, in addition to the specs above and the resources that track messages, we proved a number of lemmas (e.g. causality) that serve as reasoning principles for using the resources. Because our treatment of causality is an adaptation of Gondelman et al. [2021], the reader can consult that paper for the full list of resource lemmas.

Safety Properties. We now show how RcbLib satisfies the three the safety properties presented in Section 4.

(RCB2) No duplication. This property follows from the deliver spec (Figure 4); specifically, the postcondition guarantees that the delivered message a (if one exists), was not previously delivered to the same process ($\mathsf{OwnLocal}(i, s \cup \{a\}) * a \notin s$).

(RCB3) No creation. We prove this as a property of local state resources:

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathrm{GI}}} * \mathsf{OwnLocal}(i, s_i) * \mathsf{OwnLocal}(j, s_j) * e \in s_i * \mathit{origin}(e) = j \vdash \\ \Rrightarrow_{\mathcal{E}} \exists e'. e' \in s_j \land \lfloor e' \rfloor = \lfloor e \rfloor$$

Here, you can imagine i as the process that has just received message e. If i can assert that m originated in process j, and we also have knowledge of the local state of j in the form of $\mathsf{OwnLocal}(j, s_j)$, then the lemma guarantees that e is in fact also present in s_j (or, more precisely, that one can find messages in both local histories with equal erasures). The lemma above holds in the presence of a *global invariant* $\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathrm{GI}}}$ that RcbLib maintains to coordinate the local state resources of different replicas.

(RCB5) Causal delivery. This is the main resource lemma, which was already informally described in Section 3.1. The full form also holds under the global invariant, and uses global snapshots instead of the full global state:

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathrm{GI}}} * \mathsf{OwnLocal}(i, s) * \mathsf{OwnGlobalSnapshot}(h) \vdash \Rrightarrow_{\mathcal{E}} \forall a \in s, w \in h.\ vc(w) < vc(a) \Rightarrow \\ \exists a' \in s.\ \lfloor a' \rfloor = w$$

4.3 Correctness Proof and its Relationship to Gondelman et al. [2021]

As we mentioned in Section 3.1, our proof that RcbLib's implementation meets the specifications in Figure 4, as well as our proofs of the safety lemmas that follow under the global invariant, are based on the proof recipe outlined in Gondelman et al. [2021]. Gondelman et al. [2021] implement and specify a causally-consistent distributed key-value store, also within separation logic using Aneris. The proof recipe they outline (which we follow) can be summarized thus:

- First, model the distributed system as a state-transition system, where each state tracks the set of events at each replica[10]. Additionally, we track the global state of the system as the union of local events.
- Next, we embed the model in separation logic by using Aneris's ghost theory to create separation logic resources that represent knowledge of the local and global states. For example, Gondelman et al. [2021] construct a resource $\mathsf{Seen}(i, s)$ indicating that replica i has received *at least* the writes in s. Our analogous resource is $\mathsf{OwnLocal}(i, s)$, which captures the knowledge the replica i has delivered *exactly* the messages in s.
- Construct a global invariant (another proposition) that implies that the aforementioned resources describe reachable states in the state-transition system. For example, if we own $\mathsf{OwnLocal}(i, s)$, we can then conclude (provided the global invariant holds) that s is not an arbitrary set of messages, but instead satisfies certain safety properties (e.g. s is causally-closed, the origin field of messages is in the right range, etc.). This is also the step where we prove the resource laws (e.g. causality and no-creation).
- Finally, to verify the code running in each replica, establish a *lock invariant* [Birkedal and Bizjak 2017] that tracks the set of events that have been processed by the replica so far. In doing so, one has to carefully pick the right (combination of) resource algebras (RAs) from which to draw the separation logic resources, so that the right properties hold and invariants can be preserved.

[10]For them, an event is a write to the key-value store; for us, an event is a delivered message.

$$\Sigma = \mathbb{N} \qquad \sigma_i^0 = 0 \qquad \mathsf{prepare}_i(\mathsf{inc}, n) = \mathsf{inc} \qquad \mathsf{effect}_i(\mathsf{inc}, n) = n + 1 \qquad \mathsf{eval}_i(\mathsf{rd}, n) = n$$

Fig. 5. Specification of op-based counter CRDT from Baquero et al. [2014]

We were also able to reuse part of Gondelman et al. [2021] Coq's development in our proof of RcbLib. To be clear, we do not claim the proof recipe above as our contribution. Instead, our contribution is producing for the first time modular specifications for a general-purpose library for causal broadcast. By contrast, Gondelman et al. [2021] deal with causality specifically within the context of a key-value store. In addition, our implementation includes multiple techniques to improve reliability (e.g. sequence ids, acknowledgements, eager re-broadcasts) that are not present in Gondelman et al. [2021]. See Section 7 for additional details.

5 A LIBRARY FOR IMPLEMENTING CRDTS

Figure 5 shows a specification for a (op-based) counter CRDT[11] taken from Baquero et al. [2014]. This is not a separation-logic specification; instead, the counter is specified by instantiating several generic components: a set of states Σ (the set of naturals), an initial state (0), and a function effect that given a counter state and an operation (the counter only has one: add) returns the resulting state[12]. This style of specification is used throughout the CRDT literature [Baquero et al. 2014; Burckhardt et al. 2014a; Shapiro et al. 2011a] and it is a useful one because it allows us to focus on the parts of a CRDT that are truly unique to the CRDT in question. By contrast, the spec leaves many details unspecified: how are messages sent from one replica to others (some kind of broadcast), what happens when the current replica tries to update its state concurrently with a remote update being processed (we need locking), how is the replica state persisted across operations (mutable state). These details are common across different CRDTs, so it would be useful to factor their implementation into a separate library that can then be instantiated by CRDT implementer. This is what we have done with out OpLib library, which reuses our RCB implementation from Section 4 to provide the scaffolding for implementing op-based CRDTs.

5.1 Implementation

OpLib's code is shown in Figure 6. To use the library, the user calls oplib_init and provides serialization and deserialization functions (ser and deser) for the CRDT's operations, together with the addresses of replicas (addrs), the current replica id (rid) and most importantly the logic for the specific CRDT being implemented (crdt). The crdt value has the following polymorphic type:

```
type repIdTy = int  (* replica id *)
type 'opTy msgTy = ('opTy * vector_clock) * repIdTy
type ('opTy, 'stateTy) effectFnTy = 'opTy msgTy -> 'stateTy -> 'stateTy
type ('opTy, 'stateTy) crdtTy = 'stateTy * ('opTy, 'stateTy) effectFnTy (* init st, effect *)
```

That is, as in Figure 5 a CRDT is specified by its initial state and an effect function that knows how to transition from a state to the next. Unlike Figure 5, however, we now have executable OCaml code instead of a high-level specification.

Going back to oplib_init, the function uses the RcbLib library to obtain a pair of functions for delivering (receiving) and broadcasting messages to other replicas. It then allocates a reference to store the CRDT state (starting with the initial state provided by the user) and then forks an apply_thread that listens for messages sent by remote replicas, so we can apply their effects. Finally,

[11]Sometimes referred to as a *grow-only* or G-Counter, because the counter can only be incremented.
[12]The spec also shows two other functions: prepare which builds an "internal" operation from an "external", user-provided operation (this can often by taken to be just the identity); and eval which queries the CRDT's state.

```
let oplib_init ser dser addrs rid crdt =
  let res = rcb_init ser dser addrs rid in
  let (del, br) = res in
  let crdt_res = crdt () in
  let (init_st, eff) = crdt_res in
  let st = ref (init_st ()) in
  let lock = newlock () in
  fork (apply_thread lock del st) eff;
  (get_state lock st, update lock br st eff)

let get_state lock st () =
  acquire lock;
  let res = !st in
  release lock;
  res
```

```
let update lock br st effect op =
  acquire lock;
  let msg = br op in
  st := effect msg !st;
  release lock

let apply_thread lock del st eff =
  loop_forever (fun () ->
      acquire lock;
      begin
        match (del ()) with
          Some msg -> st := eff msg !st
        | None -> ()
      end;
      release lock;)
```

Fig. 6. Code of OpLib library

oplib_init returns a pair of functions (get_state, update) that the user can call to query the CRDT's state and update it, respectively.

The apply_thread function runs an infinite loop that first tries to deliver the next message in causal order (using RcbLib) and then, if one exists, updates the CRDT's state using the user-provided effect function.

Finally we have the user-facing functions get_state and update. The former returns a copy of CRDT's current state; the latter uses RCB to broadcast the new operation op to other replicas. RcbLib returns the user-provided operation wrapped with causality information (so an operation becomes a message); update then uses the newly-created message and the effect function to update the CRDT state.

Three points of note: first, effect is a pure function: given a state and a message it returns the resulting state. Second, concurrent accesses to the internal state (e.g. concurrent executions of apply_thread and update) are synchronized via a lock. Finally, notice that OpLib does not directly invoke any network operations (e.g. creating a network socket, sending a message, etc.); instead, all of the networking functionality is encapsulated in RcbLib.

5.2 Specification

We start by arguing why, for CRDTs, resources like $\mathrm{LocSt}(i, \bullet s, \circ h)$ that track the set of executed operations are preferable to those that track the CRDT's state. Another way to say this is that CRDTs benefit from having *intensional*[13] specifications.

From counters to replicated counters. Consider a simple counter module exposing two functions: incr() increases the counter's value by one, and read() returns the counter's current value. If used in a sequential setting, one can imagine being able to prove the following specifications: $\{\mathsf{c} \mapsto \mathsf{n}\}\, incr(c)\, \{c \mapsto n+1\}$ and $\{c \mapsto n\}\, \mathrm{query}()\, \{v.v \mapsto n\}$. Now we move to a concurrent or distributed setting, where the previous specs are still provable but no longer useful, because we need to be able to increment the counter concurrently. To solve this problem, we can track a lower bound of the counter's value, instead of the counter's exact value. Then every time we increment, we can increment the lower bound by one. This is precisely how Timany et al. [2021] structure their specification of a G-Counter CRDT: they have a resource $\mathrm{gcounter}(i, m)$, meaning that at replica i the counter's value is *at least* m (Figure 7).

[13]In the sense of Roscoe [1996], as opposed to the more common *extensional* specifications that focus on the observable effects of operations.

$$\{\mathsf{gcounter}(i,k)\}\,\langle ip_i;\ \mathtt{query}()\rangle\,\{m.\ k \le m * \mathsf{gcounter}(i,m)\} \qquad \textsc{QuerySpec}$$
$$\{\mathsf{gcounter}(i,k)\}\,\langle ip_i;\ \mathtt{incr}()\rangle\,\{().\ \exists m.\ k < m * \mathsf{gcounter}(i,m)\} \qquad \textsc{IncrSpec}$$

Fig. 7. G-Counter specification from Timany et al. [2021]

$$\{\mathsf{gcounter}(i,s,h)\}\,\langle ip_i;\ \mathtt{query}()\rangle\,\{m.\ \exists h' \supseteq h.\ m = |s \cup h'| * \mathsf{gcounter}(i,s,h')\} \qquad \textsc{QuerySpec}$$
$$\{\mathsf{gcounter}(i,s,h)\}\,\langle ip_i;\ \mathtt{incr}()\rangle\,\{().\ \exists h' \supseteq h.\ \mathsf{gcounter}(i, s \cup \{(\mathsf{inc}, i)\}, h')\} \qquad \textsc{IncrSpec}$$

Fig. 8. Intensional G-Counter specifications

This works but has at least two drawbacks. First, the incr spec is unable to distinguish between a properly-implemented counter and one that increments the state by two instead of one every time incr is called. Second, even if we are able to fix the previous issue, perhaps by tracking "contributions" as in Birkedal and Bizjak [2017], we face an even thornier problem if we consider not an increment-only counter, but one that additionally has a decrement operation. The problem there is what to write in incr's post-condition. Since the counter's state is no longer monotonic, if we start with a $\mathsf{gcounter}(i,m)$, we can end up with a $\mathsf{gcounter}(i,k)$ where k can be greater, equal, or less than m. We have lost all knowledge about the counter's state.

Consider what happens if instead of trying to track the counter's state we track the operations that the (local) counter has processed. First, it makes sense to split said operations into those that are generated locally and the ones that come from other replicas. This is because a replica "knows" the operation *it* has performed, but it does not know what operations have been performed remotely until query or incr are called. Figure 8 show these new intensional specs. In this case an operation is a pair (inc, i) containing the operation type (we only have one kind of operation inc) and the replica id. This specification style solves our problems because it allows us to track what the current thread's contribution is to the counter's state. It also scales well to handling a dec operation: the incr spec would not change; we would just need to adjust query's spec so that the result m is not just the number of recorded operations $|s \cup h'|$ but instead takes into account whether each operation is an inc or a dec.

Scaling up to CRDTs via denotations. The idea of tracking operations as opposed to state (Figure 8) can be applied to specifying additional CRDTs in addition to the G-Counter. We just need two additional ingredients: first, abstract away the function that computes the CRDT's current state from the set of received operations (so instead of returning $|s \cup h'|$ in query we want $f(s \cup h')$ for some f). Second, when operations are not naturally commutative (for example, a replicated register that stores the "last" write) CRDTs use causality information to re-introduce commutativity. This is precisely what Burckhardt et al. [2014a] do with their notion of *operation contexts* which "include all we need to know about a store execution to determine the return value of a given operation" [Burckhardt et al. 2014a]; we will use the related notion of CRDT *denotations* from Leijnse et al. [2019]. The definitions below are implicitly parametrized by a given CRDT; specifically by its set of operations Op and states St.

Definition 5.1 (Events). The set of *events* is the product $\mathsf{Event} \triangleq \mathsf{Op} \times \mathsf{VC} \times \mathbb{N}$, where VC is the type of vector clocks and the third component denotes the originating replica id for the event. We lift the partial order of vector clocks to events.'

Definition 5.2 (Denotations). A denotation $[\![\cdot]\!] : 2^{\mathsf{Event}} \rightharpoonup \mathsf{St}$ is a partial function from sets of events to states.

GetStateSpec

$$\langle \mathsf{LocSt}(i, \bullet s, \circ h) \rangle$$
$$\langle ip_i;\ \mathsf{get\_state}() \rangle$$
$$\left\langle \begin{array}{l} v.\ \exists h'\, w.\, h' \supseteq h * \mathsf{StCoh}(w, v) * \\ \quad \mathsf{LocSt}(i, \bullet s, \circ h') * [\![s \cup h']\!] = w \end{array} \right\rangle^{\mathcal{N}}$$

UpdateSpec

$$\langle \mathsf{LocSt}(i, \bullet s, \circ r) * \mathsf{GlobSt}(h) \rangle$$
$$\langle ip_i;\ \mathsf{update}(v) \rangle$$
$$\left\langle \begin{array}{l} ().\ \exists a\, r'.\, r' \supseteq r * a \notin s * a \notin h * payload(a) = v * \\ \quad origin(a) = i * a \in \mathrm{Maximals}(h \cup \{a\}) * \\ \quad a \in \mathrm{Maximum}(s \cup r' \cup \{a\}) * \\ \quad \mathsf{LocSt}(i, \bullet s \cup \{a\}, \circ r') * \mathsf{GlobSt}(h \cup \{a\}) \end{array} \right\rangle^{\mathcal{N}}$$

Fig. 9. Logically-atomic specs for get_state and update where $\mathcal{N}$ must contain global invariant's name.

As an example, the following is the denotation for a *multi-value register* (mv − register) CRDT. An mv − register is one that stores concurrent writes; writes that come later in causal order replace earlier ones. The set Op of operations is just $\{\mathsf{write}(n) | n \in \mathbb{N}\}$.

$$[\![s]\!]_{\mathsf{mv-reg}} = \{(w, vc) | \exists o.(\mathsf{write}(w), vc, o) \in s \wedge vc \in \mathrm{Maximals}(s)\}$$

A nice feature of denotations is that they support specifying higher-order CRDT *combinators*. For example, given denotations A and B, we can form their product (another denotation) $A \times B$, defined in Section 6.

We can give specifications for OpLib's get_state and update functions that are parametric on the denotations of the CRDT being implemented. These are shown in Figure 9.

GetStateSpec. We use the get_state() function to query the CRDT's state. To verify get_state(), we need to provide the local state resource $\mathsf{LocSt}(i, \bullet s, \circ h)$. When the call completes, we get back $\mathsf{LocSt}(i, \bullet s, \circ h')$ for some $h' \supseteq h$. That is, we now logically now that the CRDT has received additional remote operations (namely $h' \setminus h$), and that the local operations have not changed (because we were holding the local resource, of which there is only one copy per replica). The return value v of get_state is *coherent* with a logical representation of the state w; this is given by the predicate $\mathsf{StCoh}(w, v)$. We do this because the logical version of the state w might offer a "cleaner" representation of the state that is not polluted by the idiosyncrasies of AnerisLang's design, of which v is a value. For example, w might be a triple while AnerisLang only supports pairs, so w's encoding (v) uses nested pairs. Finally, we know that the (logical version of the) return value is the denotation of the observed operations: $[\![s \cup h']\!] = w$.

UpdateSpec. To update the CRDT, we call update(v), where v is some operation[14]. As a precondition, we must provide the local and global state resources, $\mathsf{LocSt}(i, \bullet s, \circ r)$ and $\mathsf{GlobSt}(h)$, respectively. The update function returns unit. We get back updated resources $\mathsf{LocSt}(i, \bullet s \cup \{a\}, \circ r')$ and $\mathsf{GlobSt}(h \cup \{a\})$; the latter is because around the linearization point exactly one event has been added to the entire system, namely the new event a containing the operation v. This new event originates at node i, and is maximal with respect to all other events in h, and the maximum of the (local) events in $s \cup r' \cup \{a\}$: this is just like in the broadcast spec in Figure 4. The new local resource $\mathsf{LocSt}(i, \bullet s \cup \{a\}, \circ r')$ indicates that we are now aware of exactly one additional local event a, as well as zero or more remote events $r' \supseteq r$. Finally, $a \notin h$, indicating that every update generates a new event.

Denotations and labelled-transition systems. We have seen that denotations provide a high-level specification of a CRDT. The problem, however, is that denotations are too high-level. Specifically,

[14]Notice when the user calls update they do not know what vector clock will be assigned to the operation; that happens internally once RCB broadcasts the message.

Library	# lines of OCaml	# lines of Coq
RcbLib	196	5019
OpLib	86	3595
total	282	8614

Table 2. Library metrics

the denotation has access to the entire set of operations performed on the data type, whereas in reality operations arrive one at a time (either from remote updates or due to local function calls). The solution is to give a second, lower-level specification for CRDTs, one that is closer to the running program. We do so using labelled-transition systems (LTS). Our LTS is a tuple (St, Event, $\longrightarrow$, σ_0) containing the set St of (CRDT) states, the set Event of events which serve as labels (recall that events containing operations plus causality metadata), a (partial) transition function $\longrightarrow$: St $\times$ Op, and an initial state σ_0. Additionally, we place the following two restrictions on the LTS so that the two specifications, denotation and LTS, agree:

$$[\![\emptyset]\!] = \sigma_0 \qquad\qquad \forall s\, p\, e\, p'.\mathsf{Valid}(s,e) \land [\![s]\!] = p \land p \xrightarrow{e} p' \implies [\![s \cup e]\!] = p'$$

where $\mathsf{Valid}(s,e)$ is defined as follows:

$$\begin{aligned}
\mathsf{Valid}(s,e) &\triangleq e \notin s \land e \in \mathsf{Maximals}(s \cup \{e\}) \land \mathsf{EventsExt}(s \cup \{e\}) \land \mathsf{EventsTotal}(s \cup \{e\})\\
\mathsf{EventsExt}(s) &\triangleq \forall e\, e'. e \in s \land e' \in s \land \mathsf{vc}(e) = \mathsf{vc}(e') \implies e = e'\\
\mathsf{EventsTotal}(s) &\triangleq \forall e\, e'. e \in s \land e' \in s \land \mathit{origin}(e) = \mathit{origin}(e') \land e \neq e' \implies e < e' \lor e > e'
\end{aligned}$$

That is, as we move through the LTS from the initial state, we collect the labels (events) of the transitions we have taken. At every step, the current state must be the denotation of the collected labels. Additionally, we impose additional restrictions on s and e via the $\mathsf{Valid}(s,e)$ predicate: these amount to saying that new events arrive in s as if delivered by RCB.

EffectSpec. The final piece of OpLib's public interface is the effect function passed to oplib_init. More precisely, the initialization function expects a crdt argument that is a pair (init_st, effect) containing the CRDT's initial state and the effect function. The spec of effect says that it must implement the transition function $\longrightarrow$ described above.

5.3 Correctness Proof

The core of OpLib's correctness proof is a *lock invariant* [Birkedal and Bizjak 2017] asserting that the CRDT's internal state equals the denotation of the set of operations processed so far. The logical resources needed to enforce this invariant are divided across three areas of responsibility: first, a global invariant tracks the set of messages sent by all replicas, as well as the per-replica delivered messages. This global invariant also asserts that messages are sent via the RCB protocol, allowing us to inherit all resource-related lemmas from Section 4 (e.g. causal delivery). Next, the aforementioned lock invariant also tracks the messages delivered by a specific replica; the messages are divided in two groups: local and remote. Finally, we have the user-resources such as $\mathsf{LocSt}(i, \bullet\, s, \circ\, h)$ that are useful for verifying client programs. We use a number of resource algebras, including Timany and Birkedal [2021]'s monotone construction, to carefully coordinate these different logical resources: for example, to prove that ownership of $\mathsf{LocSt}(i, \bullet\, s, \circ\, h)$ really does grant precise knowledge of the set of delivered local messages s, but only partial knowledge of the remotely-delivered messages h.

We refer the reader to our Coq development for full details on the proof. Table 2 shows the number of lines of OCaml and Coq code needed to implement and verify both RcbLib and OpLib.

	CRDT	# lines of OCaml	# lines of Coq	
combined to make	Positive-Negative Counter	25	235	Simple CRDTs
	Grown-only Counter	26	243	Simple CRDTs
	Two-Part Set	25	182	Simple CRDTs
	Add-Wins Set	34	371	Simple CRDTs
	Remove-Wins Set	53	527	Simple CRDTs
	Grow-Only Set	22	159	Simple CRDTs
combined to make	Last-Writer-Wins Register	54	555	Simple CRDTs
	Multi-Value Register	35	334	Simple CRDTs
	Product Combinator	30	374	Combinators
	Map Combinator	34	531	Combinators
	Table of Positive-Negative Counters	22	74	Compound CRDTs
	Table of Last-Writer-Wins Registers	22	74	Compound CRDTs
	total	360	3585	

Table 3. CRDTs implemented on top of OpLib

6 IMPLEMENTING CRDTS

In order to put OpLib to test we have implemented twelve CRDTs using this library. These CRDTs consist of eight simple CRDTs, two CRDT combinators, and two compound CRDTs which apply the map combinator to one of the simple CRDTs. Below, we will briefly explain these examples and discuss and summarize what is depicted in Figure 3. The relatively low number of lines of code required to implement (in OCaml) and verify the CRDTs enumerated in Figure 3 shows the usefulness and success of our methodology of building CRDTs on top of the RcbLib and OpLib libraries. Moreover, as we will discuss below, the most intricate CRDTs in Figure 3, *i.e.*, the last two rows, are those with smallest implementation and verification codes thanks to our compositional approach using CRDT combinators.

Counters. We have implemented two counter variants: Global Counter which can only have non-negative values and can only be incremented, and Positive-Negative Counter which can be both incremented and decremented. These two CRDTs are the simplest examples we have implemented. Part of the size of the Coq code is caused by having to show that the operations are commutative and associative; basic arithmetic facts which nonetheless need to be established formally in Coq.

Sets. The only operation of the Grow-Only Set CRDT allows adding an element to the set. The Add-Wins Set and Remove-Wins Set CRDTs on the other hand support both adding elements and removing elements. The treat the removal operation differently though. The issue with the removal operation is that it causes ambiguity in case of concurrent operations which add and remove the same element. The Add-Wins Set and Remove-Wins Set, as their names indicate, resolve this ambiguity in favor of addition and removal respectively. Despite their apparent similarity these two CRDTs are conceptually different as can be seen in the difference in the number of lines of Coq code required to prove their correctness. This difference is the for the Add-Wins Set CRDT we only remember the additions in the local state. When we receive a removal operation we simply remove any element that was added *strictly* before that removal operation. This makes sense as an addition that is received after a removal can never be affected by it — in worst case, it is an addition concurrent with a removal which by definition wins. On the other hand, in the Remove-Wins Set CRDT we also need to track all remove operations in the local state of each replica as addition operations received after a removal operation can be invalidated by that removal operation. The Two-Part Set CRDT is conceptually simply obtained by gluing two Grow-Only Set CRDTs together.

It tracks two sets and operations can add elements to either set. In practice, this CRDT could be obtained by combining the Grow-Only Set CRDT with the Map Combinator as a map with the domain begin a fixed set of two elements (see below). However, we chose to implement this CRDT as a yet another simple example from scratch. All set CRDTs are parameterized by the collection of elements that can be stored in sets. In the OCaml code this means that the code is parameterized by a type variable for the type of elements of the set. It is only required that these elements can be serialized as we need to communicate them over the network.

Registers. We have implemented two simple registers: a Multi-Valued Register and a Last-Writer-Wins Register. Just like sets these CRDTs are also parameterized by the collection of values that can be stored in these registers. The difference between these two CRDTs is the way they handle the issue of concurrent write operations. The Multi-Valued Register simply collects all possible values (time-wise maximal write operations) and presents them to the user upon the register along with their corresponding time-stamp. The idea is that the user will have the authority to disambiguate the situation. The Last-Writer-Wins Register on the other hand considers the latest write in the set of maximal concurrent writes and considers that to be the valid value of the register. The concurrent nature of the events in our settings means that this method of disambiguation is not always viable. After all, concurrent events can be observed in different orders by different replicas. To obtain a complete disambiguation strategy the Last-Writer-Wins Register considers the latest write from the replica with the highest replica id to prevail.

Combinators. We have implemented two CRDT combinators: the Product Combinator and the Map Combinator. The Product Combinator takes two CRDTs and constructs a CRDT where the state is the product of two states. The operations of Product CRDT are pairs of operations which take effect component-wise. The Map Combinator is a versatile combinator which takes a CRDT and constructs the CRDT of finite maps, *i.e.*, tables, of that CRDT. The state of the Map CRDT is a map with keys ranging over strings and value ranging over the states of the given CRDT. An operation is a pair of a string, the key to which the operation applies, together with an operation of the underlying CRDT. The map is initially empty. Every time an operation is received for a key that does not exit in the map it is first initialized with the initial state of the given CRDT before the operation is applied to it.

Compound CRDTS. As illustrative examples we have implemented two compound CRDTs. Both these examples use the Map Combinator. One makes a table of Last-Writer-Wins Registers while the other makes a table of Positive-Negative Counters. The fact that these relatively complicated CRDTs can be constructed and proven correct with very little effort is an excellent evidence for he success of our methodology. We obtain full-functional correctness of these essentially databases, albeit with single-column tables, in under 50 lines of Coq code including the boilerplate for including necessary Coq libraries, *etc.*

A concrete closed example. As a minimal smoke test for our OpLib library we prove safety (*i.e.* the program does not crash) of a simple example program. More precisely, using the so-called adequacy theorem of Aneris, we obtain that when this program is executed, as per the operational semantics of AnerisLang, it does not get stuck. This example initializes two replicas of Positive-Negative Counters with initial state 0. The first replica adds 1 to the counter and the second replica adds 2. They both proceed to read the value of the counter after adding to it. Intuitively, we expect the first replica to either read 1 or 3 while the second replica could read 2 or 3; and this is what both replicas *assert* as their last operation. This makes sense as each replica will definitely observe its own performed operation but might or might not have observed the operation performed by the other replica when it reads the counter. The assert command in AnerisLang is designed to evaluate

Criterion	Question
Target	Does the technique target op-based or state-based CRDTs? Most verification efforts target one of the two kinds of CRDT, but not both.
Implementation	Does the paper claim to automatically produce executable code?
Convergence	Can the technique prove convergence [Shapiro et al. 2011b]? Convergence means that if two replicas have received the same set of events, then they are in equivalent states.
Eventual Delivery	Can the technique prove eventual delivery [Shapiro et al. 2011b]? Eventual delivery means that an update delivered to a correct replica eventually reaches all correct replicas. This is a liveness property.
Causality	If required, does the paper show that messages are delivered in causal order? The alternative is either that causal delivery is not required for the specific CRDT implemented, or it is required but is then assumed.
Functional Correctness	Can the technique prove functional correctness? That is, are there specifications that show how the outputs of CRDT operations depend on their inputs?
Modularity	Does the paper show an example of a client that uses the CRDT's specification to verify some property? For example, given a G-Counter CRDT can we show that if a replica calls inc twice the counter's value is at least two?
Mechanization	Are the proofs mechanized in a proof assistant? Particularly for works that operate at the implementation level and so need to consider many low level details, it is useful to mechanize correctness arguments to increase our trust in the proof's validity. For example, some paper proofs for Operational Transformations (OT) algorithms[15] were, after publication, found to be faulty via counterexamples [Gomes et al. 2017].

Table 4. Classification criteria for CRDT verification techniques

Paper	Target	Implementation	Convergence	Event. Del.	Causality	F.C.	Modularity	Mechanization
Burckhardt et al. [2014b]	both		✓			✓		
Zeller et al. [2014]	state		✓		free	✓		✓
Nair et al. [2020]	state		✓[16]		free			
Timany et al. [2021]	state	✓	✓	✓	free	✓	✓	✓
Gomes et al. [2017]	op	✓	✓					✓
Liu et al. [2020]	op	✓	✓		✓	✓		✓
Liang and Feng [2021]	op		✓			✓	✓	
this work	op	✓	✓		✓	✓	✓	✓

Table 5. Comparison of different CRDT verification techniques. "Event. Del." stands for eventual delivery, and "F.C." for functional correctness.

its boolean and ignore it if it evaluates to true (it returns unit) and crash otherwise; hence showing safety of the example does indeed establish that the result each replica obtains when reading the counter is as expected. Intuitively, to establish this property, we simply need to enforce, using an invariant, that the global state (tracked using the proposition $\mathrm{GlobSt}(h)$) has at most two operations in it: an addition of 1 to the counter originating in the first replica and an addition of 2 originating from the second replica. Therefore, each replica by knowing its own local state (tracked using the proposition $\mathrm{LocSt}(i, \bullet s, \circ r)$), and using the relation between the global and local states of CRDTs, can conclude that the value read is the expected one.

7 RELATED WORK

The literature on verification of CRDTs has grown over the years to produce many different approaches. In order to place our work within the mosaic of existing logics and tools, we identify several design criteria that help us build a taxonomy of the sub-field. For each criterion, we propose a concrete question that helps us classify each of the pieces of related work according to the criterion. Table 4 lists our proposed criteria and how to identify whether a paper meets each of them. Table 5 then looks at whether related work meets each criterion. Some works do not fit neatly in their assigned classes, nor do we argue that our choice of questions is canonical. We nevertheless think that posing concrete questions leads to a classification that is imperfect but useful.

[15] An alternative to CRDTs.

[16] In addition to convergence, Nair et al. [2020] prove many other safety properties for specific CRDTs.

Takeaways. We structure our discussion of related work around Table 5. Most techniques target either state-based CRDTs or op-based CRDTs, but not both. The exception is Burckhardt et al. [2014a], which focuses mostly on specifying both kinds of CRDTs via denotations, but not on verification.

As explained in Burckhardt et al. [2014a] state-based approaches guarantee causal delivery "for free". This is because communicating an entire state is (logically) equivalent to sending an operation together with all its causal dependencies.

The only modular state-based approach that we are aware of is Timany et al. [2021]. This is also the only related work that proves eventual delivery. Like us, Timany et al. [2021] use the Aneris separation logic; however, unlike us they only verify one example CRDT (a G-Counter), and their specification style (which tracks states as opposed to sets of operations) is less expressive than ours (see Section 5). To prove liveness, Timany et al. [2021] develop an extension to the Iris program framework called Trillium, which allows them to show that the CRDT implementation refines a state-transition system. It would be interesting to restructure our development to use Trillium, since we already show that our CRDT implementations implement a labelled-transition system. Finally, Timany et al. [2021] focus on (one) state-based CRDT, whereas we verify multiple op-based CRDTs (see Section 8 for a discussion of how our approach could be extended to the state-based setting).

Looking at op-based CRDTs, the closest techniques are those in Liu et al. [2020] and Liang and Feng [2021].

Liu et al. [2020] extend Liquid Haskell [Vazou et al. 2014] by annotating typeclass declarations with refinement types. Their system can later typecheck instances against their (expressive) definitions. As a case study, they define a typeclass for op-based CRDTs and implement several instances, including a map combinator similar to ours. Instances of the CRDT class enjoy a *strong convergence* property that says that certain allowed permutations of a set of operations lead to the same final state. Additionally, they show functional correctness of their multi-set implementation by a simulation argument with respect to an abstract denotational specification (similarly to how we use denotations). They design their CRDTs so that they do not have to assume causal delivery, although in the process they do end up implementing parts of a causal broadcast algorithm (e.g. a delay queue).

The main difference between Liu et al. [2020] and our work is how modular the approaches are. In Liu et al. [2020] there does not seem to be a way to use strong convergence to verify a client program that uses a CRDT[17]. By contrast, as shown in Section 6 we can use our separation logic resources that track local and global states not only to show convergence and functional correctness, but also to verify clients. Additionally, we were able to verify causal broadcast as a general purpose library which is then re-use by our CRDT library. In their work, only the "business-logic" part of the CRDT is verified: that is their CRDTs are purely-functional data structures that are unaware of the existence of other replicas. By contrast, we verify not only the purely-functional part of a CRDT, but also all of its logic all the way through to network operations.

Liang and Feng [2021] introduce the first technique that produces modular specifications for op-based CRDTs. Specifically, they strengthen SEC to arrive at a trace property called *Abstract Converging Consistency* (ACC), which combines SEC with functional correctness. Functional correctness is obtained by relating a concrete CRDT model Π to its abstract counterpart Γ. In proving the relation, one is allowed to re-order certain abstract (commutative) operations that satisfy an *arbitration* relation $\bowtie$. Once we prove ACC, an *abstraction* theorem gives us contextual refinement:

[17]This is backed by the fact that they do not verify the client applications (a text editor and event calendar) that use their CRDTs.

meaning that in every program we can substitute the concrete CRDT by the abstract one (its spec) and still obtain the same results. The paper then introduces a rely-guarantee style logic to prove specification for clients using the CRDT.

Our work differs from Liang and Feng [2021]'s in several aspects. First, their CRDTs, including the concrete variants, are closer to what we would call specifications and not executable implementations. This is because they represent CRDTs as collections of functions that go from state to state via operations (this is very similar to our LTS-based models). In contrast, our CRDT implementations are written in OCaml and so must deal with many details associated with running code: e.g. message serialization, network sockets, node-local concurrency, and mutation. Second, when proving functional correctness of a client in their system one proves a judgment of the form $\vdash \{P\}$ with $(\Gamma, \bowtie)$ do $C_1 || \ldots || C_n \{Q\}$. That is, the existence of the CRDT is baked into the top-level term that one reasons about, and the CRDT Γ is distinguished from the clients C_i. By contrast, in our setting the CRDT and client code are both written in AnerisLang, and our reasoning principles come in the form of standard separation-logic resources (e.g. $\mathsf{LocSt}(i, \bullet\, s, \circ\, h)$). We expect that our approach makes it easier to re-use a verified CRDT as a component of a larger system; for example, we were able to create and use CRDT combinators.

Another related work (not shown on Table 5) is Gondelman et al. [2021]. They implement and verify a causally-consistent distributed key-value store, also using Aneris. Even though the term "CRDT" does not appear in their paper, they implement and model causal delivery of key-value store operations, and more generally their key-value store is very close to being a CRDT. It is not because it violates SEC: in certain execution traces, we can end up with replicas that have received the same set of writes, yet the same key is mapped to different values. This is because their tie-braking mechanism for concurrent writes is to take the write that arrives later, which is sensitive to network delays. Additionally, their db replicas do not re-transmit dropped messages, and they do not re-broadcast messages. As mentioned in Section 4 we adapt Gondelman et al. [2021]'s modelling of causality in separation logic so that it is applicable to a general-purpose RCB protocol. In fact, our table-of-registers example from Table 3 is also a key-value store where the above reliability issues are addressed. Our work can be then seen as generalizing the approach in their paper to apply to a wide range of CRDTs, as opposed to a bespoke key-value store.

8 CONCLUSIONS

We have verified implementations of multiple op-based CRDTs in separation logic. We structured our development as a collection of libraries. First, we verified an RcbLib library for Reliable Causal Broadcast. On top of RcbLib we then verified an OpLib library for building op-based CRDTs. CRDT implementers can use OpLib to implement their CRDTs as purely-functional data structures, without having to worry about low-level implementation details such as network operations and concurrency control. Finally, using OpLib we verified multiple CRDT instances: some are naturally commutative, while others use causality information and metadata to make operations commutative. That we were able to handle different kinds of CRDTs, including higher-order combinators, shows the applicability of our technique to a variety of scenarios. Our approach both can verify realistic implementations (as opposed to high-level protocols) and is modular (we can verify components in isolation and put their proofs back together to obtain verified stacks of components).

Future work: state-based CRDTs. A natural question is whether our techniques could be adapted to verify state-based CRDTs. More precisely, whether local and global resources can track operations in that setting even though the entire CRDT state is sent between replicas, as opposed to sending one operation at a time. We think the answer is yes; the insight is that when two CRDT states, which are lattice elements, are merged by taking their least upper bound, the operations that generated those states can also be merged[18]. That is, logically we can also take the least upper bound in the powerset lattice of operations, so that if we are in a state $\mathrm{LocSt}(i, \bullet s, \circ h)$ and a foreign replica sends their state that resulted from operations coming from a set q, then we can update our state to $\mathrm{LocSt}(i, \bullet s, \circ h \cup q')$, where $q' = \{e \in q | origin(e) \neq i\}$. The goal of this line of work would be to produce a common specification style for both kinds of CRDTs, so that clients can use a CRDT without worrying about which implementation strategy was used.

REFERENCES

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Comput.* 9, 1 (1995), 37–49. https://doi.org/10.1007/BF01784241

Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* 761–772. https://doi.org/10.1145/2463676.2465279

Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2014. Making operation-based CRDTs operation-based. In *Proceedings of the First Workshop on the Principles and Practice of Eventual Consistency, PaPEC@EuroSys 2014, April 13, 2014, Amsterdam, The Netherlands*, Marc Shapiro (Ed.). ACM, 7:1–7:2. https://doi.org/10.1145/2596631.2596632

Lars Birkedal and Aleš Bizjak. 2017. Lecture Notes on Iris: Higher-Order Concurrent Separation Log. (2017). http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf

Kenneth Birman, Andre Schiper, and Pat Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)* 9, 3 (1991), 272–314.

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014a. Replicated Data Types: Specification, Verification, Optimality. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014).* ACM, 271–284. https://doi.org/10.1145/2535838.2535848

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014b. Replicated data types: specification, verification, optimality. In *POPL*. ACM, 271–284.

Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, Chapter 3.

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 243–258. https://doi.org/10.1145/3341301.3359632

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4:1–4:26. https://doi.org/10.1145/1365815.1365816

Kristina Chodorow and Michael Dirolf. 2010. *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage.* O'Reilly.

Colin J Fidge. 1987. Timestamps in message-passing systems that preserve the partial ordering. (1987).

Seth Gilbert and Nancy A. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59. https://doi.org/10.1145/564585.564601

Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 109:1–109:28.

Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed causal memory: modular specification and verification in higher-order distributed separation logic. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434323

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* 637–650. https://doi.org/10.1145/2676726.2676980

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 -*

[18]This idea appeared in Burckhardt et al. [2014a]; the challenge would be to adapt it to separation logic so we can verify implementations modularly.

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. 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

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

Adriaan Leijnse, Paulo Sérgio Almeida, and Carlos Baquero. 2019. Higher-Order Patterns in Replicated Data Types. In *PaPoC@EuroSys*. ACM, 5:1–5:6.

Hongjin Liang and Xinyu Feng. 2021. Abstraction for conflict-free replicated data types. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 636–650. https://doi.org/10.1145/3453483.3454067

Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. 2020. Verifying replicated data types with typeclass refinements in Liquid Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 216:1–216:30.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. 401–416. https://doi.org/10.1145/2043556.2043593

Friedemann Mattern et al. 1988. *Virtual time and global states of distributed systems.* Univ., Department of Computer Science.

Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the Safety of Highly-Available Distributed Objects. 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 (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 544–571. https://doi.org/10.1007/978-3-030-44914-8_20

A. W. Roscoe. 1996. Intensional specifications of security protocols. In *CSFW*. IEEE Computer Society, 28–38.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report 7506. INRIA. http://hal.inria.fr/inria-00555588/

Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6976)*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. 729–730. https://doi.org/10.1145/2213836.2213945

Andrew S. Tanenbaum and Maarten van Steen. 2007. *Distributed systems - principles and paradigms, 2nd Edition.* Pearson Education.

Amin Timany and Lars Birkedal. 2021. Reasoning about monotonicity in separation logic. In *CPP*. ACM, 91–104.

Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2021. Trillium: Unifying refinement and higher-order distributed separation logic. *arXiv preprint arXiv:2109.07863* (2021).

Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, and Jack Mulrow. 2019. Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. 636–650. https://doi.org/10.1145/3299869.3314049

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *ICFP*. ACM, 269–282.

Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal Specification and Verification of CRDTs. In *FORTE (Lecture Notes in Computer Science, Vol. 8461)*. Springer, 33–48.