Modular Verification of State-Based CRDTs in Separation Logic

Abel Nieto
Aarhus University, Denmark

Arnaud Daby-Seesaram
ENS Paris-Saclay, France

Léon Gondelman
Aarhus University, Denmark

Amin Timany
Aarhus University, Denmark

Lars Birkedal
Aarhus University, Denmark

Abstract

Conflict-free Replicated Datatypes (CRDTs) are a class of distributed data structures that are highly-available and weakly consistent. The CRDT taxonomy is further divided into two subclasses: state-based and operation-based (op-based). Recent prior work showed how to use separation logic to verify convergence and functional correctness of op-based CRDTs while (a) verifying implementations (as opposed to high-level protocols), (b) giving high level specifications that abstract from low-level implementation details, and (c) providing specifications that are modular (i.e. allow client code to use the CRDT like an abstract data type). We extend this separation logic approach to verification of CRDTs to handle state-based CRDTs, while respecting the desiderata (a)–(c). The key idea is to track the state of a CRDT as a function of the set of operations that produced that state. Using the observation that state-based CRDTs are automatically causally-consistent, we obtain CRDT specifications that are agnostic to whether a CRDT is state- or op-based. When taken together with prior work, our technique thus provides a unified approach to specification and verification of op- and state-based CRDTs. We have tested our approach by verifying StateLib, a library for building state-based CRDTs. Using StateLib, we have further verified convergence and functional correctness of multiple example CRDTs from the literature. Our proofs are written in the Aneris distributed separation logic and are mechanized in Coq.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Distributed algorithms; Theory of computation → Separation logic

Keywords and phrases separation logic, distributed systems, CRDT, replicated data type, formal verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.12

Supplementary Material *Software (Proof Artifact)*: `https://doi.org/10.5281/zenodo.7718868`

Funding This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

1 Introduction

Conflict-free Replicated Data Types (CRDTs) are a class of distributed data structures that trade off strong consistency in favour of high availability. That is, local updates are not blocked by inter-replica synchronization; instead, they are immediately applied locally, and then propagated to other replicas. Because of the lack of synchronization, CRDTs need

a mechanism for resolving conflicting updates: a typical strategy is to make all updates commutative, so that they can be applied in any order.

There are two main implementation strategies for CRDTs, differing in how updates are propagated. *Operation-based* (op-based) CRDTs propagate local updates by reifying updates as operations, which are then transmitted to other nodes. Once these (now remote) operations are received by other replicas, they can be applied to their local states so they can "catch up". By contrast, in *state-based* CRDTs, an update is first applied to the local state, and then the *state* is propagated to other replicas. This is achieved by letting the state be an element of a join-semilattice, constraining updates to be monotonic, and combining local and remote states via the lattice's join operator. The choice of implementation style for a CRDT (op vs state-based) incurs several tradeoffs. Op-based based CRDTs are conceptually simpler, but make assumptions about the underlying delivery mechanism (e.g. at most once delivery). By contrast, the state-based approach can easily cope with duplicates and messages delivered out of order, because the merge operation (modelled with joins) is idempotent, associative, and commutative. On other hand, not only must the datatype semantics be encoded via joins, but also sending the entire state across the network might be inefficient.

Figure 1 shows a *grow-only counter* (g-counter) CRDT implemented in both styles. A g-counter is a datatype with two operations: it can be read and it can be incremented by a non-negative number. The op-based implementation defines the counter's initial state (0), an `effect` function that adds the value we are incrementing by to the counter's current state, and a `read` function that is just the identity. An *event* is a tuple containing an operation (the value to increment by) plus metadata, including the replica id where the operation originated. The state-based g-counter is more complex. The counter's state is kept as a list of integers tracking each replica's contribution to the counter's value. The initial state is the list of all zeroes with size `numRep`, the number of replicas. A `mutator` function takes the current state and the increment value, and returns the updated list where the right entry, as determined by the operation's origin, was incremented. The `merge` function takes two states and computes their join in the underlying lattice. For the g-counter, we take the pointwise maximum of the two lists. Finally, to read the value of the g-counter we just sum all entries in the state list. These purely functional implementations capture the g-counter's core logic, but do not show how events are propagated between replicas.

The standard consistency model for CRDTs is *Strong Eventual Consistency* (SEC). SEC can, in turn, be divided into two sub-properties: *convergence* (two replicas that have processed the same set of updates must be in the same state) and *eventual delivery* (any update sent by a replica will eventually be delivered to all other replicas).

Additionally, the guarantees of SEC are sometimes strengthened to imply *causal consistency* [2], meaning that the causal order of updates is respected. In other words, given updates u and w, if u *happened before* w at a replica [12], then u must be processed before w at all other replicas. Causal consistency captures programmer's intuitions on how the order of operations should be preserved; for example, it implies that reads are monotonic: a read always returns data that is "fresher" than what previous reads have returned.

1.1 Denotational Specifications

Specifying CRDTs is tricky because of their replicated nature and relaxed consistency model. Some works adopt SEC as the correctness criteria and do not provide functional correctness specifications of CRDTs [18, 7, 17]. Eventual consistency is a key correctness property, but this approach has at least two (related) shortcomings. First, given, e.g., a g-counter implementation, proving SEC shows that different replicas eventually converge, but does

```
(* op-based g-counter *)
let init = 0

let effect st e =
  let (op, _, _) = e in
  st + op

let read st = st
```

```
(* state-based g-counter *)
let init = List.init numRep (fun _ → 0)

let mutator st e =
  let (op, src) = e in
  List.mapi (fun i c →
    if i = src then c + op else c)

let merge st1 st2 =
  let max = fun p →
    Int.max (fst p) (snd p) in
  List.map max (List.combine st1 st2)

let read = List.fold_left (+) 0
```

Figure 1 Op-based and state-based OCaml implementations of a grow-only counter. `numRep` is the number of replicas.

not tell us what they converge to. This means we cannot rule out bugs such as subtracting instead of adding in the definition of `effect` in Figure 1. Another problem is that by focusing on SEC we cannot abstract away from implementation details. For example, Figure 1 shows two g-counter implementations that use different techniques, but we should be able to reason about a g-counter as an *abstract data type* [15], without worrying about whether it is op-based or state-based.

Burckhardt et al. [3] were the first to give functional correctness specifications of CRDTs. Their key observation is that just like a sequential data type (e.g., a queue) can be specified as a partial function from a list of operations to a (final) state, a replicated data type can be specified as a partial function from a *set of events* to a state. As in Figure 1, an *event* contains an operation plus additional metadata, including the id of the replica that created the operation and a timestamp. The timestamps induce a partial or total order on the set of events and, furthermore, that order is consistent with causality.[1]

We call this partial function from sets of events to the CRDT's state after processing the events a *denotation*.[2] For example, the denotation for a g-counter is $[\![s]\!] = \sum_{e \in s} e.o$, where s is a set of events and $e.o$ extracts e's operation (the value to increment by). In this particular case we do not use the event metadata to specify the g-counter, but we do so for other, more complex, CRDTs where all operations do not naturally commute. Note that any CRDT specification that uses denotations trivially satisfies the convergence part of SEC, because $[\![\cdot]\!]$ is insensitive to the order in which events arrive at different replicas: if they have received the same set of events, then their states will be the same. Also notice that denotations are by nature closer to the informal op-based specification in Figure 1 than to the state-based one. This is because op-based CRDTs are framed in terms of individual operations.

1.2 Verifying with Denotations

The papers by Burckhardt et al. and Leinjse et al. [3, 13] are concerned with specifying CRDTs, but there is still a need for a mechanism that ties the high-level specifications, given in terms of denotations, to executable code written using features of a modern programming language: e.g., mutation, node-local concurrency, and higher-order functions. The recent work of Nieto et al. [20] connects denotations to low-level CRDT implementations using the

[1] In Burckhardt et al. [3] a *visibility* relation is used to order events, instead of a timestamp.
[2] The term is due to Leijnse et al. [13], who recast Burckhardt et al.'s formalism in a style more amenable to specifying CRDT combinators.

Aneris distributed separation logic [11]. Specifically, they show how to build separation logic propositions that track the local state of a CRDT, where, e.g., the return value of a read is then given by a denotation of the local state. Nieto et al. demonstrate their approach by verifying a library for building op-based CRDTs: the library user (the CRDT implementer) instantiates the library with a purely-functional implementation of the CRDT (similar to the op-based example in Figure 1), and obtain in return a replicated data type. The library handles network operations, concurrency control, and mutation of local state. Nieto et al. exclusively reason about operation-based CRDTs. As future work, the authors include a high-level sketch of how their techniques might be adapted to the state-based setting.

There is then, to the best of our knowledge, an unexplored gap in the literature for verifying functional correctness of *state*-based CRDTs using modular specifications.

Related to the last point, existing approaches to verifying CRDTs target either op-based [7, 16, 14, 17, 20] or state-based [23, 18, 24] CRDTs, but never both kinds. This is important because it means that it is not possible to give the *same* specification to two implementations of the same replicated data type, where each uses a different implementation strategy (as in Figure 1). Having specifications that hide away implementation details is something we take for granted for sequential data types (e.g. a set abstract data type can be implemented both via a linked list and a hash table, but both implementations can be given the same specifcation). It would be useful to have the same hold for CRDTs.

1.3 Contributions

We fill the gaps identified above through the following contributions:

1. We give the first modular specification of a general class of state-based CRDTs. Our specifications are given in the Aneris distributed separation logic and our proofs are mechanized in Coq.
2. Furthermore, when taken together with Nieto et al. [20], our work provides a *unified* framework for the specification and verification of *both* kinds of CRDTs. This is because our specifications of state-based CRDTs are compatible with Nieto et al.'s specifications of op-based CRDTs. We emphasize this point by re-verifying the example client program in Nieto et al. that uses a positive-negative counter CRDT, except we swap their op-based counter by a state-based equivalent. Crucially, the client's safety proof remains virtually unchanged,[3] showing that it is possible to specify CRDTs while hiding their implementation strategy.
3. We give the first formal proof that state-based CRDTs are causally-consistent.
4. We evaluate our approach by verifying a set of example CRDTs from the literature. The evaluation shows that our techniques can handle a variety of CRDTs, including counters, sets, and higher-order combinators.

2 Aneris Primer

Aneris [11] is a distributed separation logic built on top of the Iris program logic framework [9]. Aneris is designed to reason about safety properties of distributed systems written in Aneris-Lang, which can be thought of as a subset of OCaml deeply embedded in Coq. This subset includes support for higher-order functions, mutable state, node-local concurrency (including

[3] Modulo some manual editing of Nieto et al.'s proof development, which could be further eliminated with additional Coq engineering work that refactors some typeclasses

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

Figure 2 Aneris fragment adapted from Nieto et al. [20].

the ability to fork new threads dynamically), as well as expressions for sending and receiving messages over UDP-style sockets. The operational semantics models an unreliable network: once sent, messages can be dropped, re-ordered, arbitrarily-delayed, and duplicated.

Figure 2 shows the fragment of Aneris most relevant to this paper. First, notice that the logic includes the usual connectives of a higher-order logic. The separation logic connectives include the separating conjunction $P * Q$, indicating ownership of a resource that can be split into two parts, one satisfying P and the other satisfying Q. The magic wand $P \mathrel{-\!\!*} Q$ denotes resources that, when combined with a resource satisfying P then together satisfy Q. The points-to proposition $\ell \mapsto_{ip} v$ grants exclusive ownership of memory location ℓ on the node with IP address ip alongside the knowledge that value v is stored in ℓ. In other words, only the owner of this resource is allowed to read or modify ℓ's contents. As usual, the Hoare triple $\{P\}\ \langle ip; e\rangle\ \{x.\ Q\}$ is a partial correctness assertion for expression e running on the node with IP address ip. Notice that in the postcondition we bind the return value of e to x, whose scope extends over Q.

Aneris inherits from Iris a notion of *invariant*. An invariant $\boxed{P}^{\mathcal{N}}$ ($\mathcal{N}$ being the name — see below), once established at a point in a proof, asserts that the proposition P hold throughout the execution of the program from that point on and is respected by all threads and nodes. This is enforced by the program logic and is reflected in the invariant *opening* rule. The invariant opening rule allows invariants to be accessed around atomic expressions. That is, it allows us to assume that the invariant holds before the atomic step and enforces that after the atomic step executes, the invariant needs to be *closed* again, meaning that we have to show that it holds again after the execution of the atomic step. The notation $\boxed{P}^{\mathcal{N}}$ says that P is an invariant with *namespace* $\mathcal{N}$. One can think of $\mathcal{N}$ as an identifier for the invariant that helps the logic keep track of which invariants are open at any given point in the proof: this is important because an invariant that is currently open cannot be re-opened.

The points-to proposition $\ell \mapsto_{ip} v$ is but one example of the kind of *resources* that one can assert ownership over. In fact, the user of the logic can define new kinds of resources by creating *partial-commutative monoids* (PCMs): monoids where the product is commutative and partial. Given a PCM $\mathbb{M}$ and $a \in \mathbb{M}$ the proposition $\boxed{a}^{\gamma}$ asserts ownership of *ghost state* a. Here, γ is the *ghost name* under which a is *stored*. Crucially, $\boxed{a}^{\gamma} * \boxed{b}^{\gamma}$ is equivalent to the monoid product $\boxed{a \cdot b}^{\gamma}$. The logic guarantees that the product of all resources stored under the same ghost name is well-defined: hence by choosing appropriate monoids we can tweak the properties of ghost state.

The proposition $\Box P$, read *persistently* P, tells us that P holds and it does not assert ownership of any exclusive resources. We say that P is a persistent proposition if $P \vdash \Box P$. Persistent propositions are duplicable in the sense that $\Box P \vdash \Box P * \Box P$. Finally, the *update* modality ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$ says that P holds after updating (allocating or modifying) resources; furthermore, we can assume that all invariants in the set $\mathcal{E}_1$ hold when establishing P but must also (re)establish all invariants in the set $\mathcal{E}_2$. The notation $\Rightarrow_{\mathcal{E}}$ is shorthand for ${}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}}$.

3 Main Ideas

The main idea of this paper is that, from a user's perspective, whether a CRDT is op-based or state-based is an implementation detail, and one that ought not affect the data structure's specification. To capture this idea formally we reach for two tools: separation logic propositions for tracking the global and local states of the CRDT as a function of sets of events (user operations), and high-level specifications for the CRDT based on denotations.

To track a CRDT's state we construct propositions $\mathsf{GlobSt}(s_\mathsf{g})$ and $\mathsf{LocSt}\,(i, s_\mathsf{own}, s_\mathsf{for})$ for global and local states, respectively, as well as a global invariant $\mathsf{GlobInv}$. Ownership of the global state tells us that s_g is *exactly* the set of all events issued by any replica. An event is a triple (op, source, time) where op is the user-provided operation (e.g. `inc(10)` for a g-counter), source is the id of the replica that issued the operation (ids are just natural numbers), and time is a logical timestamp that allows us to order events according to the usual *happens-before* relation. Ownership of the local state $\mathsf{LocSt}\,(i, s_\mathsf{own}, s_\mathsf{for})$ tells us that replica i has issued *exactly* the events in s_own, and that it has received *at least* the events in s_for, which all originate outside of i (we only get an approximation of the events from outside of i because in between two user interactions with the CRDT at replica i, new merge operations might have taken place).

Once the above propositions have been defined, we prove a comprehensive suite of "resource lemmas" that allow a client to reason about the state of the CRDT. For example, we prove that if a user knows both $\mathsf{GlobSt}(s_\mathsf{g})$ and $\mathsf{LocSt}\,(i, s_\mathsf{own}, s_\mathsf{for})$, and they can find events e and e' where $e \in s_\mathsf{own} \cup s_\mathsf{for}$ (e is in the local state), $e' \in s_\mathsf{g}$ (e' is in the global state) and e' happens before e, then e' must have been received at replica i as well: $e' \in s_\mathsf{own} \cup s_\mathsf{for}$. This is an encoding of causality in separation logic [8, 20]. We do not claim this formulation as a contribution; instead, our contribution is defining the above predicates in such a way that they can track state-based CRDTs, and then proving existing lemma statements for our definitions. In effect, we have re-implemented an existing interface. This is crucial because it means that from a client's perspective it does not matter what kind of CRDT (state-based or op-based) they are working with: they all satisfy the same laws.

Technical Challenges

Resource lemmas like causality hold only in the presence of the global invariant $\mathsf{GlobInv}$. This invariant guarantees that at all times the state of the system is *valid*. The system state is a tuple $\vec{s_l}$ of the local state at every replica. Defining a notion of validity suitable for state-based CRDTs is one of the tricky technical parts of our proof. Two complications arise: how to represent local states, and how to represent time.

Defining local state as an element of the lattice over which the CRDT is implemented does not work, because we lose track of the individual events. For example, we might remember that the state of the counter is `[4, 5]`, but not that it resulted from three events: two increments of 2 each at replica 0, and one increment of 5 at replica 1. Remembering events is needed to show the resource lemmas. We therefore represent local states as sets of events, but now we need a way to link these sets of events to the lattice element that is computed at runtime. We do this with denotations, which are functions from sets of events to lattice elements. In our proof, a local invariant ensures that, for every replica i, if s is the set of tracked events for i and the runtime CRDT state at i is st, we must have $[\![s]\!] = st$. The definition of denotation for a state-based CRDT must then satisfy a number of *coherence* properties with respect to the underlying lattice: for example, we require that, under certain conditions, if $[\![s]\!] = st$ and $[\![s']\!] = st'$, then $[\![s \cup s']\!] = st \sqcup st'$. In other words, our proof tracks

the logical state with a free lattice of events, while the implementation computes using the CRDT-specific lattice. The denotation is the homomorphism between the two.

For lemmas like causality we also need to be able to compare events according to time. Prior work on op-based CRDTs implements a causal broadcast library that tags each event with a (physical) *vector clock* [4] that serves as a timestamp [20]. In our setting, since state-based CRDTs do not assume causal broadcast, we use a purely logical notion of time. Given an event e, its timestamp is taken to be the set of events that e causally depends on. This set is computed at the point e is created: if e was issued at replica i where the local state is an event set s, then e causally depends on every element of s. This works because we can show that local states are always *dependency-closed*: if $e' \in s$ and e_d is a dependency of e', then $e_d \in s$ as well. Our definition of logical time allows us to give the first formal proof that state-based CRDTs are causally-consistent.

Verified Examples

With the above in place we turn to the verification of different state-based CRDTs. We implemented and verified five CRDTs from the literature [21]: a grow-only counter, a positive-negative counter, an add-only set, a combinator for products of CRDTs, and a combinator for maps from strings to an underlying CRDT type. We implemented these examples in two steps: first, we implemented a StateLib library that factors out all the common elements in different examples: network calls, merging of remote states, and concurrency control. The library takes as input a purely-functional implementation of a state-based CRDT, in the form of a triple `(init_st, mutator, merge)`, where `init_st` is the CRDT's initial (lattice based) state, `mutator` is a function that takes a state and an operation and produces the next state, and merge implements the lattice's least upper bound operation. StateLib requires that the CRDT implementer proves the aforementioned coherence properties (e.g., that merging two states is the same as taking the denotation of the union of their corresponding event sets) about their purely-functional implementation. Given this core logic, the library returns a fully-fledged replicated data type and two functions, `get_state` and `update`, to query and update the state of the data-structure. In the second step, we wrote purely-functional implementations for each of the previously-mentioned examples and proved the relevant coherence properties so the library can be instantiated with them. The modular design of the library allows us to prove the library safe just once, and then re-use that proof to obtain safety proofs for each of the CRDTs.

Finally, we wanted to validate our claim that a client need not know whether the CRDT they are interacting with is state-based or not. We did this by using the example in Nieto et al. where they verify a client program together with an op-based positive-negative counter [20]. We were able to swap their op-based counter with our state-based positive-negative counter while leaving the proof virtually unmodified (small technical changes are required, but these could be eliminated with further Coq engineering). This shows that CRDTs can be true abstract data types, and can be specified while abstracting away implementation details.

The rest of the paper is structured as follows: Section 4 gives an overview of the CRDT resource lemmas in Nieto et al. [20], which we re-prove in the state-based context. Sections 5, 6, and 7 present StateLib's design, specification, and safety proof, respectively. Section 8 describes the verified example CRDTs, as well as the proof of the client program that is

[4] A vector of integers, with one entry per node in the system. The ith entry tells us how many updates have been done by replica i.

agnostic to the CRDT class. Section 9 surveys related work and Section 10 concludes.

4 Background: CRDTs in Separation Logic

We give an overview of the separation logic approach to verification of op-based CRDTs in Nieto et al. [20]. Specifically, they introduce abstract separation logic resources (abstract predicates) for tracking the local and global states of a CRDT. The abstract resources are later used in the specifications of CRDT operations. Nieto et al. reason only about op-based CRDTs, but in this paper we show how to instantiate the abstract resources so that we can verify state-based CRDT implementations as well. As we will later see, this allows clients to reason about a CRDT while remaining agnostic of the CRDT's implementation strategy.

4.1 Time, Events, and Denotations

We start by giving a few definitions that we will use throughout the paper.

Logical time allows us to order events in a distributed system using causal order. Time is axiomatized by a triple $(\mathsf{Time}, \leq_t, <_t)$, where Time is a set of *timestamps*, $\leq_t$ is a partial order on timestamps, and $<_t$ is the strict version of $\leq_t$. For example, for working with op-based CRDTs, Nieto et al. instantiate logical time by taking timestamps to be vector clocks and $\leq_t$ to be the associated pointwise ordering.

Logical events represent operations that are executed by the CRDT, together with metadata. A logical event is a triple $(\mathsf{op}, \mathsf{source}, \mathsf{time}) \in \mathsf{Event} \triangleq \mathsf{Op} \times \mathbb{N} \times \mathsf{Time}$. Here Op is the type of *operations* on the CRDT (e.g. $\mathsf{Op} \triangleq \{\mathsf{inc}(n) \mid n \in \mathbb{N}\}$ for a g-counter), source is the id of the replica that generated the event, and time is a timestamp. For an event e we write $e.o$, $e.s$, and $e.t$ for the operation, source, and timestamp of e, respectively.

Given two event sets s and s', we say that s is a *causally-closed subset* of s', written $s \subseteq_{\mathsf{cc}} s'$, if $s \subseteq s'$ and

$$\forall e\ e',\ e \in s' \Rightarrow e' \in s' \Rightarrow e.t \leq_t e'.t \Rightarrow e' \in s \Rightarrow e \in s$$

That is, if we start with two events from s' and the later one e' (according to timestamp ordering) is in s, then e (its causal dependency) must be in s as well.

Finally, we use *denotations* to specify CRDTs. A denotation is a tuple $(\mathsf{Op}, \mathsf{St}, s_{\mathsf{init}} : \mathsf{St}, [\![\cdot]\!] : \mathcal{P}(\mathsf{Event}) \rightharpoonup \mathsf{St})$. For example, as in Figure 1, for a g-counter we could have Op as previously defined, $\mathsf{St} \triangleq \mathbb{N}$, $s_{\mathsf{init}} \triangleq 0$, and $[\![s]\!] = \sum_{e \in s} \mathsf{unwrap}(e.o)$ with $\mathsf{unwrap}(\mathsf{inc}(n)) = n$. We could have also chosen St to be the set of lists of naturals of length N, where N is the number of replicas. This latter denotation would be closer to the state-based implementation.

It is useful for denotations to be partial because we can avoid giving a meaning to ill-formed sets of events. For example, suppose we have $s = \{a, b\}$ with $a.t = b.t$ but $a \neq b$. We might know that in practice events with equal timestamps must be equal, so s can never arise during an execution. We might then choose $[\![s]\!]$ to be undefined.

4.2 Separation Logic Resources

So far we have not shown any Aneris definitions. We do so in Figure 3, which shows the types of different abstract separation logic resources (predicates) that, together with associated lemmas, can be used to reason about the state of CRDTs. Specifically, these resources appear in the pre and post-conditions of functions that operate on a CRDT. The abstract resources are designed to be generic so they can be used with multiple CRDTs. Indeed, Nieto et al. verified multiple op-based example CRDTs using these resources [20], and we

Resources (abstract predicates)

$$
\begin{array}{rl}
\text{(Global invariant)} & \mathsf{GlobInv} : iProp \\
\text{(Global state)} & \mathsf{GlobSt} : \mathcal{P}(\mathsf{Event}) \to iProp \\
\text{(Global snapshot)} & \mathsf{GlobSnap} : \mathcal{P}(\mathsf{Event}) \to iProp \\
\text{(Local state)} & \mathsf{LocSt} : \mathbb{N} \to \mathcal{P}(\mathsf{Event}) \Rightarrow \mathcal{P}(\mathsf{Event}) \to iProp \\
\text{(Local snapshot)} & \mathsf{LocSnap} : \mathbb{N} \to \mathcal{P}(\mathsf{Event}) \Rightarrow \mathcal{P}(\mathsf{Event}) \to iProp
\end{array}
$$

Global state laws

$$
\begin{array}{rl}
\text{(GlobStTakeSnap)} & \forall E\ s, \mathcal{N}^{\uparrow} \subseteq E \Rightarrow \mathsf{GlobInv} \twoheadrightarrow \mathsf{GlobSt}(s) \twoheadrightarrow \Rrightarrow_E \mathsf{GlobSt}(s) * \mathsf{GlobSnap}(s) \\
\text{(GlobSnapIncl)} & \forall E\ s\ s', \mathcal{N}^{\uparrow} \subseteq E \Rightarrow \mathsf{GlobInv} \twoheadrightarrow \mathsf{GlobSnap}(s) \twoheadrightarrow \mathsf{GlobSt}(s') \twoheadrightarrow \Rrightarrow_E s \subseteq s' * \mathsf{GlobSt}(s')
\end{array}
$$

Local state laws

$$
\begin{array}{rl}
\text{(LocSnapIncl)} & \forall E\ i\ s_{\mathsf{own}}\ s_{\mathsf{for}}\ s'_{\mathsf{own}}\ s'_{\mathsf{for}}, \mathcal{N}^{\uparrow} \subseteq E \Rightarrow \mathsf{GlobInv} \twoheadrightarrow \mathsf{LocSnap}(i, s_{\mathsf{own}}, s_{\mathsf{for}}) \twoheadrightarrow \mathsf{LocSt}\left(i,\ s'_{\mathsf{own}},\ s'_{\mathsf{for}}\right) \twoheadrightarrow \\
& \quad \Rrightarrow_E s_{\mathsf{own}} \cup s_{\mathsf{for}} \subseteq_{\mathsf{cc}} s'_{\mathsf{own}} \cup s'_{\mathsf{for}} * \mathsf{LocSt}\left(i,\ s'_{\mathsf{own}},\ s'_{\mathsf{for}}\right) \\
\text{(LocSnapExt)} & \forall E\ i\ i'\ s_{\mathsf{own}}\ s_{\mathsf{for}}\ s'_{\mathsf{own}}\ s'_{\mathsf{for}}, \mathsf{LocSnap}(i, s_{\mathsf{own}}, s_{\mathsf{for}}) \twoheadrightarrow \mathsf{LocSnap}(i', s'_{\mathsf{own}}, s'_{\mathsf{for}}) \twoheadrightarrow \\
& \quad \Rrightarrow_E \forall e\ e', e \in s_{\mathsf{own}} \cup s_{\mathsf{for}} \Rightarrow e' \in s'_{\mathsf{own}} \cup s'_{\mathsf{for}} \Rightarrow e.t = e'.t \Rightarrow e = e' \\
\text{(LocSnapProv)} & \forall E\ i\ s_{\mathsf{own}}\ s_{\mathsf{for}}\ e, \mathcal{N}^{\uparrow} \subseteq E \Rightarrow e \in s_{\mathsf{own}} \cup s_{\mathsf{for}} \twoheadrightarrow \mathsf{GlobInv} \twoheadrightarrow \mathsf{LocSnap}(i, s_{\mathsf{own}}, s_{\mathsf{for}}) \twoheadrightarrow \\
& \quad \Rrightarrow_E \exists s_g, \mathsf{GlobSnap}(s_g) * e \in s_g \\
\text{(GlobSnapProv)} & \forall E\ i\ s_{\mathsf{own}}\ s_{\mathsf{for}}\ s_g, \mathcal{N}^{\uparrow} \subseteq E \Rightarrow \mathsf{GlobInv} \twoheadrightarrow \mathsf{LocSt}\left(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}}\right) \twoheadrightarrow \mathsf{GlobSnap}(s_g) \twoheadrightarrow \\
& \quad \Rrightarrow_E \mathsf{LocSt}\left(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}}\right) * \forall e, e \in s_g \Rightarrow \mathsf{EV\_Orig}(e) = i \Rightarrow e \in s_{\mathsf{own}} \\
\text{(Causality)} & \forall E\ i\ s_{\mathsf{own}}\ s_{\mathsf{for}}\ s_g, \mathcal{N}^{\uparrow} \subseteq E \Rightarrow \mathsf{GlobInv} \twoheadrightarrow \mathsf{LocSt}\left(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}}\right) \twoheadrightarrow \mathsf{GlobSnap}(s_g) \twoheadrightarrow \\
& \quad \Rrightarrow_E \mathsf{LocSt}\left(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}}\right) * \forall e\ e', e \in s_g \Rightarrow e' \in s_{\mathsf{own}} \cup s_{\mathsf{for}} \Rightarrow e <_t e' \Rightarrow e \in s_{\mathsf{own}} \cup s_{\mathsf{for}}
\end{array}
$$

Figure 3 CRDT resources and selected lemmas, from Nieto et al. [20].

have also verified multiple state-based examples. Notice that Figure 3 does not show the definition of the resources. The reader can think of Figure 3 as defining an *interface* in the software engineering sense. Nieto et al. implement this interface for op-based CRDTs, while we re-implement it for state-based CRDTs. For space reasons, we do not show the entire interface; the full interface can be found in the accompanying Coq code.

The defined resources are as follows: there is a *global invariant* $\mathsf{GlobInv}$ that holds throughout the CRDT's existence. Then we define resources for tracking *global* (GlobSt) and *local* (LocSt) states. The intuition is that if $\mathsf{GlobSt}(s_{\mathsf{g}})$ holds, then we know that s_{g} is *exactly* the set of events issued by any CRDT replica in the system. Similarly, ownership of $\mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})$ tells us that replica i has processed *exactly* the events in s_{own} and *at least* the events in s_{for}. The events in s_{own} (the "own" events) must have all originated at i, while the ones in s_{for} (the "foreign" events) must all originate outside of i. This is because a client of the CRDT knows exactly which events it has issued, but is potentially not aware of all events that have been received "in the background" since the last interaction with the CRDT.

Global and local states are *exclusive* (not shown in Figure 3), meaning that only one copy of the resource (or one per replica in the case of local state) can exist: e.g., $\mathsf{GlobSt}(s_{\mathsf{g}}) \twoheadrightarrow \mathsf{GlobSt}(s'_{\mathsf{g}}) \twoheadrightarrow \bot$.

By contrast, the interface also declares *global and local snapshots* ($\mathsf{GlobSnap}$ and $\mathsf{LocSnap}$), which are persistent (duplicable) and give us a *lower bound* on global and local states, respectively. Snapshots are useful as a "certificate" that an event was generated by a given replica: this is the case if one can prove, e.g., that $\mathsf{GlobSnap}(s_{\mathsf{g}})$ and $e \in s_{\mathsf{g}}$ for an event e. The use of global snapshots as certificates is validated by lemma $\mathsf{GlobSnapIncl}$ (Figure 3).

The lemma says that under the global invariant, if we own $\mathsf{GlobSnap}(s_g)$ and $\mathsf{GlobSt}(s'_g)$, then we must have $s_g \subseteq s'_g$. This conclusion holds under the update modality $\Rightarrow\!\!\!\!\Rightarrow_E$, which means that it holds possibly after opening (and closing) all invariants in the mask E. The premise $\mathcal{N}^\uparrow \subseteq E$ tells us that the global invariant's namespace $\mathcal{N}^\uparrow$ is part of E. This means that GlobInv must not be open when the lemma is called (because the proof of the lemma opens GlobInv). LocSnapIncl provides similar inclusion guarantees for local snapshots, but note that we actually get the stronger causally-closed inclusion $\subseteq_{\mathsf{cc}}$, as opposed to just $\subseteq$.

GlobStTakeSnap allows us to take snapshots of global states. LocSnapExt says that if two events in a local snapshot have equal timestamps, then the events must be equal.

Finally, we have three lemmas that tie together local and global states. LocSnapProv says that if e is tracked locally, then there must exist some global snapshot $\mathsf{GlobSnap}(s_g)$ such that $e \in s_g$. That is, all local events are also tracked globally. GlobSnapProv says that if an event $e \in s_g$ has origin i and we know $\mathsf{GlobSnap}(s_g)$ (e is tracked globally), then e must also be in the local state for i. Finally, Causality is our definition of causality in separation logic. This take on causality was originally developed for reasoning about a causally-consistent key value store by Gondelman et al. [8], and later generalized by Nieto et al. [20] so it can apply to events in an arbitrary CRDT. The lemma is as follows: suppose we have two events e and e' such that e' was recorded locally at node i (that is, $e' \in s_{\mathsf{own}} \cup s_{\mathsf{for}}$ and we know $\mathsf{LocSt}\,(i,\, s_{\mathsf{own}},\, s_{\mathsf{for}})$). Next suppose that e happened before e', and e is a logically tracked event (which we can show by presenting $\mathsf{GlobSnap}(s_g)$ with $e \in s_g$). Then causal delivery requires that e be observed locally at i as well: i.e., $e \in s_{\mathsf{own}} \cup s_{\mathsf{for}}$. Gondelman et al. [8] show how this definition of causality is strong enough to prove four *session guarantees* (monotonic reads, monotonic writes, read your writes, and writes follow reads) that programmers intuitively expect when interacting with a causally-consistent datatype.

Reasoning With Resources

To tie all the above together, Figure 4 shows how the previously-discussed resources can be used to specify the `inc` operation on a g-counter CRDT. We assume that `inc` both increments the counter and returns its current local value. The precondition for calling `inc` at replica i requires knowledge of both $\mathsf{LocSt}\,(i,\, s_{\mathsf{own}},\, s_{\mathsf{for}})$ and $\mathsf{GlobSt}(s_g)$. This is because every single increment must be tracked both locally at the replica where it is performed and globally. In the postcondition we get back $\mathsf{LocSt}\,(i,\, s_{\mathsf{own}} \cup \{e\}\,,\, s'_{\mathsf{for}})$, where e is the event generated by the increment and $s_{\mathsf{for}} \subseteq s'_{\mathsf{for}}$. Notice that the "own events" grow by exactly one event, e, but the "foreign events" grow to some superset s'_{for} of s_{for}. This is because since the last time `inc` was called any number of new events could have been propagated from other replicas to replica i. Notice how we connect the implementation to its functional correctness specification by saying that the return value v is the denotation of the locally-observed events $s_{\mathsf{own}} \cup \{e\} \cup s'_{\mathsf{for}}$. Finally, observe that by using denotations we automatically obtain convergence (the safety part of SEC), because the return value is a function of a *set* of events, so two replicas that have seen the same set of events must return the same result.

Resources for State-Based CRDTs

Two difficulties arise when instantiating the resource interface for state-based CRDTs.

1. How should we track local state? The replica state in an implementation of a state-based CRDT is a lattice element. By contrast, the resource interface logically tracks replica states as sets of events (operations). The solution is to link the two representations via a denotation: if the logical state is $\mathsf{LocSt}\,(i,\, s_{\mathsf{own}},\, s_{\mathsf{for}})$, then the physical state must be

IncSpec
$\{\mathsf{LocSt}\,(i,\, s_{\mathsf{own}},\, s_{\mathsf{for}}) * \mathsf{GlobSt}(s_{\mathsf{g}})\}$
$\quad\langle ip_i;\, \mathsf{inc}(n)\rangle$
$$\left\{\begin{array}{l} v.\ \exists e\, s'_{\mathsf{for}}.\ s'_{\mathsf{for}} \supseteq s_{\mathsf{for}} * e \notin s_{\mathsf{g}} * e.o = n * e.s = i \\ \quad [\![s_{\mathsf{own}} \cup \{e\} \cup s'_{\mathsf{for}}]\!] = v * \mathsf{LocSt}\left(i,\, s_{\mathsf{own}} \cup \{e\},\, s'_{\mathsf{for}}\right) * \mathsf{GlobSt}(s_{\mathsf{g}} \cup \{e\}) \end{array}\right\}$$

Figure 4 Simplified spec of an increment operation, which returns the counter's current value.

$[\![s_{\mathsf{own}} \cup s_{\mathsf{for}}]\!]$, which in turn must be drawn from the appropriate lattice. The link is also needed when propagating a replica's state to other replicas. In the implementation, a replica sends a message containing its entire state e, so others can merge it. Logically, we require that the sent state be paired with a local snapshot whose denotation is e.

2. How do we track time to prove causal consistency? For their treatment of op-based CRDTs, Nieto et al. [20] implemented a causal broadcast algorithm that ensures that every message sent by a CRDT replica is delivered in causal order. This is achieved via vector clocks in the standard way. But state-based CRDTs should not rely on causal broadcast; in fact, one of the main advantages of the state-based approach is that messages can be delivered out-of-order and re-delivered without causing issues (because of the properties of least upper bound). It is not immediately clear why the state-based design satisfies causal delivery. The first key observation is that if we start with a replica state st and look at the event set s that produced it, i.e., $[\![s]\!] = st$, then s "has no holes" with respect to causality. That is, if we take an event $e \in s$ and e' is another event that happened before e, then it must be the case that $e' \in s$ as well. This is formalized via the notion of *dependency-closure* (Section 7.1). The second observation is that when a new operation o is applied to a local state st with $[\![s]\!] = st$, it (logically) generates a new event e' with $e'.o = o$. e''s causal dependencies are *exactly* the events in s. This is important because it means that we can track an event's dependencies *purely logically*, without the need for vector clocks. Using these ideas we are able to prove that the (Causality) lemma holds for state-based CRDTs (Lemma 5). To our knowledge, this is the first formal proof that state-based CRDTs are causally-consistent.

5 StateLib : a Library for Implementing State-Based CRDTs

We have structured our CRDT implementations so that common functionality is factored out into a separate library, which can then be instantiated as needed by different CRDT examples. The library, called StateLib, is responsible for maintaining the CRDT's internal state and inter-replica propagation. The library's code is shown in Figure 5, together with an abridged example instantiating a *grow-only set* (g-set) CRDT, a set to which we can add elements, but from which we cannot remove them [21].

We start by describing the `init` function, which is the library's entry point. A CRDT implementer calls `init` with the following arguments: serialization and de-serialization functions, a list of all replica addresses, the current replica id, and a `crdt` parameter that describes how the specific CRDT being instantiated should behave. The serialization functions have the expected types: `type 's serT = 's → string` and `type 's deserT = string → 's`. The `crdt` argument has type `('o,'s) crdT`, parametrized on operations and states:

```
type 's mergeT = 's → 's → 's
type ('o,'s) mutT = int → 's → 'o → 's
type ('o,'s) crdtT = (('s * ('o,'s) mutT) * 's mergeT)
```

That is, a value of type `('o, 's) crdtT` is a triple `(init_st, mutator, merge)` containing an initial

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

let rec loop_forever thunk =
  thunk ();
  loop_forever thunk

let apply deser lk sh st merge : =
  loop_forever (fun () →
    let msg =
      unSOME (receiveFrom sh) in
    let st' = deser (fst msg) in
    acquire lk;
    st := merge !st st';
    release lk)

let update lk mut i st op =
  acquire lk;
  st := mut i !st op; (* LP *)
  release lk

let sendToAll sh dstl i msg =
  let j = ref 0 in
  let rec aux () =
    if !j < list_length dstl then
    if i = j then (j := !j +1; aux ())
    else begin
      let dst =
        unSOME (list_nth dstl !j) in
      sendTo sh msg dst;
      j := !j + 1
      aux ()
    end
    else ()
  in aux ()
```

```
let broadcast ser lk sh st dstl i =
  loop_forever(fun () →
    Unix.sleepf 2.0;
    acquire lk;
    let s = !st in
    release lk;
    let msg = ser s in
    sendToAll sh dstl i msg)

let init ser deser addrs rid crdt =
  let ((init_st, mut), merge) = crdt in
  let st = ref (init_st ()) in
  let lk = newlock () in
  let sh = socket () in
  let addr = unSOME (list_nth addrs rid) in
  socketBind sh addr;
  fork (apply deser lk sh st merge);
  fork (broadcast ser lk sh st addrs rid);
  let get = get_state lk st in
  let upd = update lk mut rid st in
  (get, upd)


(* G-Set instantiation *)

let mutator i st op = set_add op st
let merge st1 st2 = set_union st1 st2
let init_st = set_empty
let gset_crdt = ((init_st, mutator), merge)

(* Instantiate via *)
let (get, upd) = init ... gset_crdt
```

Figure 5 StateLib implementation and a G-Set example. Linearization points are marked with an `LP` comment.

state, a mutator function, and a merge function. The mutator takes a replica id, the current state, and a new operation, and returns the state that results after applying the operation locally. The merge function takes two states and returns their least upper bound.

Back to the body of `init`, we see that it unpacks the `crdt` argument. It then allocates a local reference to hold the current state of the CRDT, a lock to control updates to the state, and a socket over which it can communicate with other replicas. The function then spawns two concurrent threads, `apply` and `broadcast`, in charge of receiving updates from other replicas and propagating local updates, respectively. Finally, `init` returns a pair of functions `get_state` and `update` allowing the library user to query and update the CRDT state.

Both `get_state` and `update` have simple implementations. The former just dereferences the local state, while the latter uses the user-provided mutator to compute the CRDT's next state. Both operations are guarded by a lock.[5]

The `apply` function, which is spawned off as a separate thread from `init`, is responsible for receiving updates from other replicas and then merging them with the local state, using the user-provided `merge` function. The call to `receiveFrom` blocks until a message is available at the given socket handle. The function `unSOME : 'a option → 'a` unwraps a value of an option type, crashing if the argument is `None`. Notice that the received message needs to be

[5] In the case of `get_state`, loads in AnerisLang are atomic, so the lock is not strictly needed; however, dereferences are not atomic in OCaml [6], which we use to run our code, so we use a lock.

deserialized via the user-provided `deser` function. Also notice that `apply` does not terminate, but mutates the CRDT's state.

The dual of `apply` is `broadcast`, which is tasked with propagating the local state to other replicas. This function also runs on a separate thread, and loops forever, working solely via side effects. The `broadcast` function retrieves a copy of the local state, serializes it, and then calls a helper function `sendToAll`. This helper takes a list of IP addresses `dstl`, the current replica id `i` and the message (state) to be sent. It then loops over the elements of `dstl` and sends the message to each of them (taking care to not send a message to itself).

Finally, Figure 5 sketches how one might implement a g-set via the library. We represent the state with a sequential set. The mutator is just set insertion, `merge` is implemented via set union, and the initial state is the empty set. We can then package these three components into a tuple `gset_crdt`, and provide the latter as an argument to `init` (together with the serialization functions and replica IP addresses). From `init` we get back a pair functions for querying the current value of the set and updating it.

6 Specifying StateLib

The StateLib library has two interfaces: an internal interface used by the CRDT *implementer*, consisting of the `init` function, and an external interface used by the CRDT *client*, consisting of get_state and update.

6.1 Internal Interface

Recall that StateLib is initialized by the CRDT implementer through an `init` function, taking in (de-) serialization functions for the CRDT state, a list of replica IP addresses, and finally a `crdt` argument describing the lattice being implemented (Figure 5). This last parameter is a triple $\mathsf{crdt} = (\mathsf{initSt}, \mathsf{mut}, \mathsf{merge})$ consisting of the CRDT's initial state $\mathsf{initSt} \in \mathsf{LatSt}$, a *mutator* function $\mathsf{mut} : \mathsf{LatSt} \to \mathsf{Event} \to \mathsf{LatSt}$, and a *merge* function $\mathsf{merge} : \mathsf{LatSt} \to \mathsf{LatSt} \to \mathsf{LatSt}$.

The CRDT implementer first defines a poset $(\mathsf{LatSt}, \leq_L)$ and then proves that it is a lattice. Because our tracking of replica states is defined in terms of event sets (Figure 3) we need a way to connect said events to the physical CRDT state, which is a lattice element. Intuitively, we would like to guarantee that the CRDT's physical state is the denotation of the set of events received so far. For this to be true, we need certain coherence properties between event sets, their denotations, and lattice elements. These are shown in Figure 6 and consist of specifications for initSt, mut and merge.

InitStSpec says that the denotation of the empty set of events must be the initial state.

MutatorSpec says that if we start in a state $st = [\![s]\!]$ and through a mutation get to a state $st' = \mathsf{mut}(st, op)$, then we can also arrive at st' by taking the denotation $[\![s \cup \{e\}]\!]$, where e is the event containing op. We also need to show that the mutator is monotonic: we must have $st \leq_L st'$ in the lattice order. In proving these goals we get to make additional assumptions about s and e. Specifically, we can assume that s is a set of events that is *valid with respect to coherence* (we explain validity in Section 7.1); additionally, we know that e is the maximum element with respect to timestamp ordering in the set $s \cup \{e\}$. Intuitively, this is because e is a new event being added, so it has every event in s as a causal dependency.

MergeSpec shows coherence of merges. Here we start with two states st and st' that we want to merge to get a third state st''. We know that $[\![s]\!] = st$ and $[\![s']\!] = st'$, and we would like to conclude that $[\![s \cup s']\!] = st''$ and also that merge is in fact computing the least upper bound, so $st \sqcup st' = st''$. Once again we get to assume validity of the relevant event

InitStSpec: $[\![\emptyset]\!] = \mathsf{initSt}$

MutatorSpec
$$\left\{\begin{array}{l}[\![s]\!] = st \land e.o = op \land e.s = i \land e \notin s \\ \land\ \mathsf{CohVal}(s \cup \{e\}) \land \mathsf{maximum}(e, s \cup \{e\})\end{array}\right\}$$
$$\langle ip_i;\ \mathsf{mut}(st, op)\rangle$$
$$\left\{st'.\ [\![s \cup \{e\}]\!] = st' \land st \leq_L st'\right\}$$

MergeSpec
$$\left\{\begin{array}{l}[\![s_1]\!] = st_1 \land [\![s_2]\!] = st_2 \land \mathsf{SectIncl}(s_1, s_2) \\ \land\ \mathsf{CohVal}(s_1) \land \mathsf{CohVal}(s_2) \land \mathsf{CohVal}(s_1 \cup s_2)\end{array}\right\}$$
$$\langle ip_i;\ \mathsf{merge}(st_1, st_2)\rangle$$
$$\left\{st'.\ st_1 \sqcup st_2 = st' \land [\![s_1 \cup s_2]\!] = st'\right\}$$

$\mathsf{sect}(s, i) \triangleq \{e \in s \mid e.s = i\}$
$\mathsf{SectIncl}(s, s') \triangleq \forall i.\ \mathsf{sect}(s, i) \subseteq \mathsf{sect}(s', i) \lor \mathsf{sect}(s', i) \subseteq \mathsf{sect}(s, i)$
$\mathsf{CohVal}(s)$ is a version of "local state validity" (Section 7.1) that does not imply $\mathsf{depclosed}(s)$.
CrdtSpec$((\mathsf{initSt}, \mathsf{mut}, \mathsf{merge})) \triangleq$ InitStSpec$(\mathsf{initSt}) *$ MutatorSpec$(\mathsf{mut}) *$ MergeSpec(merge)

InitSpec
$$\{\ldots * \textsc{CrdtSpec}(\mathsf{crdt})\}$$
$$\langle ip_i;\ \mathsf{init}(\mathsf{addrs}, \mathsf{repId}, \mathsf{crdt})\rangle$$
$$\{(\mathsf{get\_state}, \mathsf{update}).\ \mathsf{LocSt}\,(i, \emptyset, \emptyset) * \textsc{GetStateSpec}(\mathsf{get\_state}) * \textsc{UpdateSpec}(\mathsf{update})\}$$

Figure 6 Internal specifications. GetStateSpec and UpdateSpec are defined in Figure 7.

sets, and now additionally we know an inclusion property of sections (a section is a subset of events that originates at a specific replica). The proposition $\mathsf{SectIncl}(s, s')$ tells us that if we look at any particular section, say section i, then either $\mathsf{sect}(s, i)$ is a subset of $\mathsf{sect}(s', i)$, or the other way around. Intuitively, this is because sections are always "complete": if a replica has received event $(6, 5)$, it must have also received all events in the range $(6, 1), \ldots, (6, 4)$.

We package the three specifications in the assertion CrdtSpec(crdt), which asserts that each of the components of the crdt tuple satisfies the corresponding spec above. Finally we have specification for the init function. In the precondition of InitSpec, we assert that the crdt argument satisfies CrdtSpec. In the postcondition, we learn that init returns a pair of functions get_state and update that satisfy the same-named specifications (described in the next section). We also gain ownership of the resource $\mathsf{LocSt}\,(i, \emptyset, \emptyset)$, indicating that the local replica has yet to receive any events (because it has just been initialized).

6.2 External Interface

StateLib's external interface consists of two functions: get_state, which takes no arguments and returns the local state of the CRDT, and update, which takes an operation, updates the local state, and returns a Unit. Figure 7 shows specifications for both functions; these specifications are identical to the ones for the same-named functions in Nieto et al.'s library for op-based CRDTs [20]. This is by design: by proving that our library meets the same specification as the equivalent library for op-based CRDTs we then make it possible for client programs to use (and reason about) a replicated data type without knowledge of whether the data type is op-based or state-based. That is, we hide implementation details to turn CRDTs into true abstract data types.

Looking at Figure 7, the reader will notice that the specifications use angle brackets instead of braces: i.e. we write $\langle P\rangle e\langle Q\rangle^{\mathcal{N}}$, instead of the usual Hoare triple $\{P\}e\{Q\}$. The former is a *logically-atomic* triple [10]. The motivation for these triples is that Aneris invariants can only be opened around *atomic* steps (otherwise concurrent threads might

GetStateSpec
$\langle \mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})\rangle$
$\langle ip_i;\ \mathsf{get\_state}()\rangle$
$\left\langle v.\ \exists s'_{\mathsf{for}}\ w.\ s'_{\mathsf{for}} \supseteq s_{\mathsf{for}} * \mathsf{StCoh}(w, v) * \mathsf{LocSt}\left(i,\ s_{\mathsf{own}},\ s'_{\mathsf{for}}\right) * [\![s_{\mathsf{own}} \cup s'_{\mathsf{for}}]\!] = w \right\rangle^{\mathcal{N}}$

UpdateSpec
$\langle \mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}}) * \mathsf{GlobSt}(s_{\mathsf{g}})\rangle$
$\langle ip_i;\ \mathsf{update}(v)\rangle$
$\left\langle ().\ \exists e\ s'_{\mathsf{for}}.\ s'_{\mathsf{for}} \supseteq s_{\mathsf{for}} * e \notin s_{\mathsf{own}} * e \notin s_{\mathsf{g}} * e.o = v * e.s = i * e \in \mathsf{maximals}(s_{\mathsf{g}} \cup \{e\}) * \mathsf{maximum}(e, s_{\mathsf{own}} \cup s'_{\mathsf{for}} \cup \{e\}) * \mathsf{LocSt}\left(i,\ s_{\mathsf{own}} \cup \{e\},\ s'_{\mathsf{for}}\right) * \mathsf{GlobSt}(s_{\mathsf{g}} \cup \{e\}) \right\rangle^{\mathcal{N}}$

Figure 7 External specifications. $\mathcal{N}$ is any invariant namespace that includes GlobInv's name. Adapted from Nieto et al. [20].

observe invariant violations); this means we cannot use a specification $\{P\}e\{Q\}$ if we need to open an invariant to prove P, provided e is not atomic, as is the case for both get_state and update. In particular, the precondition of update requires the global state $\mathsf{GlobSt}(s_{\mathsf{g}})$, which a client is likely to keep in an invariant because it is shared by all (concurrent) replicas. Logically-atomic triples solve this problem by allowing us to open invariants when proving the pre-condition. Their informal semantics are as follows: if we know $\langle P\rangle e\langle Q\rangle^{\mathcal{N}}$, then we know that e executes without crashing (although it might not terminate) provided that P holds. When proving P we are allowed to use any invariants that are not in namespace $\mathcal{N}$. [6] We also need to show that if Q holds we can close any invariants that were open when proving P. Another point of view is that P and Q hold around a *linearization point* in e; the linearization points for `get_state` and `update` are marked with an `LP` comment in Figure 5.

The specification of get_state can be read as follows. Before calling get_state we should know the local state at replica i: $\mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})$; afterwards, the function returns a physical state v which is *coherent* with a logical state w, [7] the local state is now $\mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s'_{\mathsf{for}})$ where $s'_{\mathsf{for}} \supseteq s_{\mathsf{for}}$ (reflecting the fact the set of local events is unchanged, but additional remote events might have been received), and the returned value w is the denotation of $s_{\mathsf{own}} \cup s'_{\mathsf{for}}$. Notice that the fact that the returned value is a function of the set of received events automatically gives us the safety part of eventual consistency (convergence): if two replicas have received the same set of events, then they are in the same state.

The specification of update says that we need to know both the local state $\mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})$ at replica i and the global state $\mathsf{GlobSt}(s_{\mathsf{g}})$. This is because every event needs to be tracked both locally and globally. The function does not return any meaningful value, but we do get (logical) knowledge that the set of events has expanded: locally we now know $\mathsf{LocSt}\,(i,\ s_{\mathsf{own}} \cup \{e\},\ s'_{\mathsf{for}})$ with $s_{\mathsf{for}} \subseteq s'_{\mathsf{for}}$, and globally we have $\mathsf{GlobSt}(s_{\mathsf{g}} \cup \{e\})$. The new event e ($e \notin s_{\mathsf{g}}$) has the value we are updating by as its payload ($e.o = v$) and it originates at replica i. Finally, we know that this new event is more recent than any other locally-received event ($\mathsf{maximum}(e, s_{\mathsf{own}} \cup s'_{\mathsf{for}} \cup \{e\})$), and we also know that no other event (even globally) has the new event as its dependency: $e \in \mathsf{maximals}(s_{\mathsf{g}} \cup \{e\})$.

[6] This is to prevent a user of StateLib from opening the global invariant, which is needed by StateLib. Invariants cannot be reopened, to preserve soundness of the logic.

[7] The *state coherence* predicate $\mathsf{StCoh}(w, v)$ links the physical and logical states. This is useful when the physical state has a more involved representation due to limitations of AnerisLang: for example, v might be a pair of pairs while w is a 3-tuple, because AnerisLang only supports pairs.

7 Verifying StateLib

To prove that StateLib meets its external specification we follow a proof methodology inspired by previous Aneris developments [8, 20]:

1. We first model the CRDT as a state-transition system (STS), where the STS states are tuples detailing the state of the CRDT both globally and at each replica (Section 7.1). Transitions correspond to mutations and merges. Crucially, we show that transitions preserve *state validity*, a safety invariant from which we can derive properties of interest (e.g., causality). This is done in the meta-logic (i.e., Coq) and outside of separation logic.

2. We then embed the STS model inside Aneris via a combination of invariants and ghost state (defined via PCMs, see Section 7.3). We use state validity and the properties of the relevant PCMs to show that the resource interface from Section 4 holds.

3. Finally, using the separation logic resources defined in the previous step, we prove that StateLib's implementation meets its external specification (Section 7.4).

The rest of Section 7 is technical in nature, so the reader interested in an overview of our work can skip to Section 8. We highlight Lemmas 4 and 5, which, to our knowledge, are the first formal proofs that state-based CRDTs are causally-consistent.

7.1 State-Transition System Model

We model the execution of a CRDT via an STS that keeps track of the per-replica state, as well as the global state. We then show a number of safety invariants that hold for any execution of the STS. In later sections we show how the AnerisLang implementation simulates the STS, therefore inheriting its safety properties. We use the simulation to prove the resource laws in Figure 3.

The reader might wonder why we develop this STS model when state-based CRDTs already have a well-understood model that is lattice-based. We do this because our goal is to prove that StateLib satisfies general functional correctness specifications that apply to both state-based CRTDs and op-based CRDTs. To this end we write our high-level specifications in terms of denotations, which talk about sets of events instead of lattice elements. This is why we need the STS model below. At an operational level, the STS model is needed to define the ghost state in Section 7.3 and as such is not directly exposed to the user.

We start by defining a purely logical notion of time that allows us to reason about causality in the absence of vector clocks. *Logical time* for state-based CRDTs is a triple $\mathsf{LogTime}_{\mathsf{st}} \triangleq (\mathcal{P}(\mathsf{EvId}), \subseteq, \subset)$, where $\mathsf{EvId} \triangleq \mathbb{N} \times \mathbb{N}$. Here EvId is the set of *event ids*, which are pairs (r, n) of a *replica id* and a *sequence number*, respectively. We show that $\mathsf{LogTime}_{\mathsf{st}}$ satisfies the requirements on logical time from Section 4.1.

Given a set $d \in \mathcal{P}(\mathsf{EvId})$ of event ids we can extract the subset that originates at a given replica id via a *section*: $\mathsf{sect}(d, i) \triangleq \{(i, n) \mid (i, n) \in d\}$. We can also compare event ids, but only if they are in the same section: $(s, n) \leq (s, n') \triangleq n \leq n'$.

We define logical events as triples $(\mathsf{op}, \mathsf{src}, \mathsf{time}) \in \mathsf{Op} \times \mathbb{N} \times \mathcal{P}(\mathsf{EvId})$. We tag each event e with the set of event ids of its *causal dependencies* $e.t$ and are then able to sort events according to causal order. For example, if $e.t = \{(1, 1)\}$ and $e'.t = \{(1, 1), (2, 1)\}$, then $e.t <_t e'.t$. Let $s \in \mathcal{P}(\mathsf{Event})$. We lift causal dependencies to sets of events: $\mathsf{deps}(s) \triangleq \bigcup_{e \in s} e.t$.

The *id* of an event e can be computed by counting the number of dependencies that originate at e's origin: $\mathsf{id}(e) = (e.s, |\mathsf{sect}(e.t, e.s)|)$. This way of computing event ids makes sense only if we assume that sequence numbers (a) start at 1 and (b) there are no "holes" in the ids stored in $e.t$. In our proof we maintain an invariant that implies these two properties.

To ensure causal consistency, we care about event sets that are closed with respect to causal dependencies. Let s be a set of events, then s is *dep-closed*, written $\mathsf{depclosed}(s)$, if $\forall e \in s, id \in e.t, \exists e' \in s, \mathsf{id}(e') = id$. For example, if $e \in s$ and $(1,2) \in e.t$, then we must have $e' \in s$ with $\mathsf{id}(e') = (1,2)$. Dep-closure is preserved by set union, which is key because it will allow us to link logical and physical states when we take least upper bounds of the latter.

We now define *local states*, which track the state of the CRDT at a given replica. Unlike in the implementation the replica state will not be a lattice element, but a set of events: $\mathsf{Lst} = \mathcal{P}(\mathsf{Event})$. We lift sections to local states: if $s \in \mathsf{Lst}$ then $\mathsf{sect}(s,i) \triangleq \{e \mid e.s = i\}$.

Recall that $\texttt{numRep} \in \mathbb{N}$ denotes the number of replicas. We define *global states* $\mathsf{Gst} \triangleq \mathcal{P}(\mathsf{Event}) \times \mathsf{Lst}^{\texttt{numRep}}$. The intuition for a global state $(s_g, \vec{s_l}) \in \mathsf{Gst}$ is that the first component s_g gives us a *global view* of the system (we will ensure that s_g equals the union of all $s_{l,i}$). The second component $\vec{s_l}$ is a vector of length $\texttt{numRep}$ containing the local state at each replica.

▶ **Definition 1** (State-transition system model). *The state-transition system model is* $\mathcal{S} = (\mathsf{Gst}, \mathsf{init}_{\mathcal{S}}, \rightarrow_{\mathcal{S}})$. *STS states are elements of* Gst *and* $\mathsf{init}_{\mathcal{S}} \triangleq (\emptyset, \vec{\emptyset})$ *is the initial state. The transition relation* $\rightarrow_{\mathcal{S}} \in \mathsf{Gst} \times \mathsf{Gst}$ *is defined by the following two inference rules:*[8]

$$\frac{s_{l,i} = s \quad d = \mathsf{deps}(s) \quad n = |\mathsf{sect}(d,i)| + 1 \quad t = \{(i,n)\} \cup d \quad e = (\mathsf{op}, i, t)}{(s_g, \vec{s_l}) \rightarrow_{\mathcal{S}} (s_g \cup \{e\}, \vec{s_l}[i \mapsto s \cup \{e\}])} \ \mathsf{TUpdate}$$

$$\frac{s \subseteq s_{l,j} \quad \mathsf{depclosed}(s)}{(s_g, \vec{l}) \rightarrow_{\mathcal{S}} (s_g, \vec{l}[i \mapsto s_{l,i} \cup s])} \ \mathsf{TMerge}$$

The TUpdate rule models the execution of a new operation at a particular replica. The premises say that the local state at replica i is s and d is the set of dependencies of all events in s. Then we compute the sequence number for the new event: since it originates in i this needs to be exactly one larger than the number of dependencies in d that come from i, hence $n = |\mathsf{sect}(d,i)| + 1$. Then we build a timestamp for the new event: every (old) event in s should be a causal dependency of the new event, plus the new event's id is also a dependency, so $t = \{(i,n)\} \cup d$. Finally we build the new event $e = (\mathsf{op}, i, t)$. Given all the above, we can take a step in the STS from a state $(s_g, \vec{s_l})$ to a state that includes the new event e. Because e is new, it should be added both to the global state and the local state for i. The notation $\vec{s_l}[i \mapsto s']$ stands for the vector that is like $\vec{s_l}$ except that the i'th entry is now s'.

The TMerge rule models merge operations where a replica updates its state by receiving and merging a (potentially old) state that was sent by another replica. In the rule, we start with some subset s of the local state at replica j. It is crucial that said subset be dep-closed so that we can preserve causality. In the rule's conclusion, we merge s with the state at replica i. That is, we can think of this rule as saying that replica j transmitted its state to replica i, which subsequently merged it. Finally, notice that the fact that s is a subset of $s_{l,j}$ and not exactly $s_{l,j}$ allows us to model the delay imposed by the network on message transmission — the fact that s is a *dep-closed* subset of $s_{l,j}$ means that s is a version of the state of the j^{th} replica from the past.

[8] As usual $\rightarrow_{\mathcal{S}}^*$ denotes the reflexive transitive closure of $\rightarrow_{\mathcal{S}}$.

7.2 Safety Invariants

The goal of the STS model is to allow us to show a number of safety invariants about the execution of the system. We do this through the notions of *local* and *global state validity*. Let $s \in \mathsf{Lst}$. Then s is a *valid local state*, written $\mathsf{LocStValid}(s)$, if all the following hold:

(DepClosed)	$\mathsf{depclosed}(s)$
(SameOrigComp)	$\forall i,\ \forall e\ e' \in \mathsf{sect}(s,i), e.t <_t e'.t \vee e.t = e'.t \vee e'.t <_t e.t$
(ExtId)	$\forall e\ e' \in s, \mathsf{id}(e) = \mathsf{id}(e') \Rightarrow e = e'$
(ExtTime)	$\forall e\ e' \in s, e.t = e'.t \Rightarrow e = e'$
(OrigRange)	$\forall e \in s, e.s < \mathtt{numRep}$
(SeqIdComplete)	$\forall e \in s, n \in \mathbb{N}, 0 < (e.s, n) \leq \mathsf{id}(e) \Rightarrow (e.s, n) \in e.t$
(SeqIdNon0)	$\forall e \in s, r\ n \in \mathbb{N}, \mathsf{id}(e) = (r, n) \Rightarrow 0 < n$
(EvIdMon)	$\forall e\ e' \in s, e.s = e'.s \Rightarrow e.t \leq_t e'.t \Rightarrow \mathsf{id}(e) \leq \mathsf{id}(e')$
(EvIdIncl)	$\forall e \in s, \mathsf{id}(e) \in e.t$
(EvIdTime)	$\forall e\ e' \in s, \mathsf{id}(e) \in e'.t \Rightarrow e.t \leq_t e'.t$

The different requirements on valid local states are as follows. (DepClosed) requires that a valid state s be also dep-closed. (SameOrigComp) says that events with the same origin can be totally ordered by timestamp ordering. (ExtId) and (ExtTime) say that events with equal id, resp. timestamp, must be equal. (OrigRange) ensures that replica ids are in the expected range. (SeqIdComplete) says that if $e \in s$ and e's id is e.g. $(4, 10)$, then all timestamps in the range $(4, 1) \ldots (4, 9)$ must also be in e's dependencies. (SeqIdNon0) says that all event ids have a sequence number that starts at 1. (EvIdMon) requires that timestamps ordering and event id ordering agree. (EvIdIncl) says that an event id must be included in the event's dependencies. Finally, (EvIdTime) requires that if e's id is in the dependencies of e' then e must have in fact happened before e' according to timestamp ordering. The definition of local state validity has been simplified with respect to prior developments [8, 20]. This is because these works use vector clocks as their notion of logical and physical time, whereas we only track time logically via sets of dependencies.

From local state validity we obtain a number of derived lemmas. For example, we can relate dep-closed and causally-closed subsets:

▶ **Lemma 2.** *Let $s \subseteq s_g$ where $\mathsf{depclosed}(s)$ and $\mathsf{LocStValid}(s_g)$. Then $s \subseteq_{\mathsf{cc}} s_g$.*

Now we define validity of global states. Let $q = (s_{\mathrm{g}}, \vec{s_l}) \in \mathsf{Gst}$. Then q is a *valid global state*, written $\mathsf{GlobStValid}(q)$, if all the following hold:

(InclLocal)	$s_{\mathrm{g}} = \bigcup_{1 \leq i \leq \mathtt{numRep}} s_{l,\,i}$
(InclOrig)	$\forall e \in s_{\mathrm{g}}, e \in s_{l,\,e.s}$
(GlobValid)	$\mathsf{LocStValid}(s_{\mathrm{g}})$
(LocValid)	$\forall 1 \leq i \leq \mathtt{numRep}, \mathsf{LocStValid}(s_{l,\,i})$

Given a global state $(s_{\mathrm{g}}, \vec{s_l})$, global state validity amounts to requiring that s_{g} be the union of the $s_{l,\,i}$ (InclLocal), that events be present in the local state from which they putatively originate (InclOrig), that the global state s_{g} itself be valid if treated as a local state (GlobValid) and that each local state be valid (LocValid). The definition of global state validity is essentially unchanged from prior work [8, 20].

The definitions of local and global state validity are motivated by two desiderata: they must be *invariants*, i.e. hold for all reachable states (including the initial state); and they must imply the CRDT resource lemmas from Figure 3 "at the model level".

▶ **Theorem 3** (Validity invariant). *Let* $q \in \mathsf{Gst}$ *such that* $\mathsf{init}_{\mathcal{S}} \to^*_{\mathcal{S}} q$. *Then* $\mathsf{GlobStValid}(q)$.

We use validity to show model-level counterparts of the lemmas in Figure 3. Intuitively, ownership of $\mathsf{GlobSt}(s_{\mathsf{g}})$ tells us that the global state is $(s_{\mathsf{g}}, \vec{s_l})$, whereas if we know $\mathsf{GlobSnap}(s'_{\mathsf{g}})$ all we can say is that $s'_{\mathsf{g}} \subseteq s_{\mathsf{g}}$. Similarly, ownership of $\mathsf{LocSt}\,(i,\, s_{\mathsf{own}},\, s_{\mathsf{for}})$ corresponds to knowing that the local state $s_{l,\,i}$ (which we can assume to be valid) equals $s_{\mathsf{own}} \dot{\cup} s'_{\mathsf{for}}$ (disjoint union), with $s_{\mathsf{for}} \subseteq s'_{\mathsf{for}}$. The local snapshot $\mathsf{LocSnap}(i, s_{\mathsf{own}}, s_{\mathsf{for}})$ give us $s_{\mathsf{own}} \dot{\cup} s_{\mathsf{for}} \subseteq s_{l,\,i}$. With this analogy in mind, here is the model-level version of causality.

▶ **Lemma 4** (Model-level causality). *Let* $q = (s_g, \vec{s_l})$ *and* $\mathsf{GlobStValid}(q)$. *Also let* $e \in s \subseteq_{\mathsf{cc}} s_{l,\,i}$ *and* $e' \in s_g$, *with* $e'.t <_t e.t$. *Then* $e' \in s$.

Proof. From $\mathsf{GlobStValid}(q)$ we can conclude $\mathsf{LocStValid}(s_{l,\,i})$, which in turn gives us $\mathsf{depclosed}(s_{l,\,i})$. Since $s_{l,\,i} \subseteq s_{\mathsf{g}}$, by Lemma 2 we get $s_{l,\,i} \subseteq_{\mathsf{cc}} s_{\mathsf{g}}$. Since we assumed $s \subseteq_{\mathsf{cc}} s_{l,\,i}$, we can use transitivity of $\subseteq_{\mathsf{cc}}$ to conclude $s \subseteq_{\mathsf{cc}} s_{\mathsf{g}}$. This implies the conclusion. ◀

7.3 Separation Logic Encoding

The next step in the proof is to encode the validity invariant using separation logic. We do this using a combination of Iris invariants and resources [9]. Recall that Iris invariants are propositions that hold throughout the execution of the operational semantics and resources are elements of partial commutative monoids (PCMs).

Here we use resources whose ownership reveals what state the system is partially in (e.g. the set of events received by a specific replica). In particular, we use three main PCM constructions, which we review below: [9]

1. The *authoritative* PCM $\textsc{Auth}(\mathbb{M})$, where $\mathbb{M}$ is itself a PCM. Given a PCM $\mathbb{X}$, we can define the *extension order* on elements of the carrier as follows: $x \leq_{\mathbb{X}} y \triangleq \exists z, x \cdot z = y$. The authoritative construction gives us two kinds of resources: a *full part* $\bullet_{\mathbb{M}}\, g$ and one or more *fragmental parts* $\circ_{\mathbb{M}}\, s$, where $g, s \in \mathbb{M}$. Ownership of the full part is exclusive, while ownership of a fragment $\circ_{\mathbb{M}}\, s$ is exclusive or persistent depending on whether ownership of s is exclusive or persistent in $\mathbb{M}$. The fragmental parts are guaranteed to be smaller than the full part according to extension order, so that if we own $\boxed{\bullet_{\mathbb{M}}\, g}^{\gamma} * \boxed{\circ_{\mathbb{M}}\, s}^{\gamma}$ we can conclude $s \leq_{\mathbb{M}} g$. We also have that $\circ_{\mathbb{M}}\, g \cdot \circ_{\mathbb{M}}\, s = \circ_{\mathbb{M}}\, (g \cdot s)$.

2. The *fractional* PCM $\textsc{Frac}(X)$, where X is a carrier set. Elements of this monoid are of the form s^p, where $s \in X$ and $p \in \mathbb{Q}_{(0,1]}$. This PCM allows us to split and re-combine fractions of a resource: $s^{p+q} = s^p \cdot s^q$. We also know that if we own multiple fractions then they must add to less than 1. This is useful to e.g. make a proposition exclusive (non-duplicable) by defining it as a fraction greater than $\frac{1}{2}$ as no two copies of such a resource can be owned separately. We also know that all fractions agree on the underlying element: $\boxed{s^p}^{\gamma} * \boxed{r^q}^{\gamma}$ implies $s = r$.

3. The *monotone* PCM $\textsc{Mono}(R)$, where $R \subseteq X \times X$ is a pre-order on a carrier set X [22]. This PCM allows us to lift R to the extension order of $\textsc{Mono}(R)$: any $x \in X$ can be injected into $\textsc{Mono}(R)$ via a $\mathsf{principal}_R$ function such that $xRy \iff \mathsf{principal}_R(x) \leq_{\textsc{Mono}(R)} \mathsf{principal}_R(y)$. Combining this with the authoritative PCM gives us a monoid $\textsc{Auth}(\textsc{Mono}(R))$ where if we know $\boxed{\bullet_{\mathbb{M}}\, \mathsf{principal}_R(g)}^{\gamma} * \boxed{\circ_{\mathbb{M}}\, \mathsf{principal}_R(s)}^{\gamma}$ we can conlude sRg. We instantiate this construction with $R = \subseteq_{\mathsf{cc}}$.

[9] We use a few additional PCMs in the Coq formalization but elide those additional structures here for the sake of brevity.

We use the defined invariants and resources to prove the interface described in Figure 3. In this section, we sketch out proofs for some of the interface lemmas.

The global invariant uses the predicate **GI** below. The predicate states that there exists a (model-level) global state h which is valid. Furthermore, it asserts ownership of global and local resources defined by the predicates $\mathbf{GR}(s_\mathsf{g})$ and $\mathbf{LR}(i, s_{l,i})$, respectively.

$$\mathbf{GI} \triangleq \exists h \in \mathsf{Gst}. h = (s_\mathsf{g}, \vec{s_l}) * \mathsf{GlobStValid}(h) * \mathbf{GR}(s_\mathsf{g}) * \overset{\texttt{numRep}}{\underset{i=1}{\text{\Large$*$}}} \mathbf{LR}(i, s_{l,i})$$

Given the above definition and an invariant name ι, we can instantiate the GlobInv predicate from Figure 3 by allocating an Aneris invariant stating that **GI** holds after every execution step: $\mathsf{GlobInv} \triangleq \boxed{\mathbf{GI}}^{\iota}$.

The *global resource* predicate $\mathbf{GR}(s_\mathsf{g})$ asserts ownership of two pieces of ghost state, both of which precisely track the value of s_g: $\mathbf{GR}(s_\mathsf{g}) \triangleq \boxed{s_\mathsf{g}^{\frac{1}{3}}}^{\gamma_\mathsf{gst}} * \boxed{\bullet_{\mathbb{S}}\, s_\mathsf{g}}^{\gamma_\mathsf{gsnap}}$.

The ghost state $s_\mathsf{g}^{\frac{1}{3}}$ is drawn from the $\textsc{Frac}(\mathsf{Gst})$ PCM. Its purpose is to track the global set of events. The remaining $\frac{2}{3}$ fraction is kept *outside* of the invariant as the user-facing resource GlobSt from Figure 3: $\mathsf{GlobSt}(s_\mathsf{g}) \triangleq \boxed{s_\mathsf{g}^{\frac{2}{3}}}^{\gamma_\mathsf{gst}}$. Because the fraction in GlobSt is greater than a half, we can prove that $\mathsf{GlobSt}(s_\mathsf{g})$ is exclusive (GlobStExcl, Figure 3).

The second part of $\mathbf{GR}(s_\mathsf{g})$ asserts ownership of $\bullet_{\mathbb{S}}\, s_\mathsf{g}$. Here, $\mathbb{S}$ is the PCM of finite sets of events, with set union as composition. This means that $p \leq_{\mathbb{S}} q$ iff $p \subseteq q$. Consequently, $\boxed{\bullet_{\mathbb{S}}\, s_\mathsf{g}}^{\gamma_\mathsf{gsnap}} * \boxed{\circ_{\mathbb{S}}\, s'_\mathsf{g}}^{\gamma_\mathsf{gsnap}}$ implies $s'_\mathsf{g} \subseteq s_\mathsf{g}$. We keep the full part in the invariant and use the fragmental part to define global snapshots: $\mathsf{GlobSnap}(s) \triangleq \boxed{\circ_{\mathbb{S}}\, s'_\mathsf{g}}^{\gamma_\mathsf{gsnap}}$. Note that these fragmental parts are persistent (duplicable) as the set union operation is idempotent.

The next step is to define the local resources predicate $\mathbf{LR}(i, s)$ which tracks in the invariant the local resources for replica i:

$$\begin{aligned}\mathbf{LR}(i, s) \triangleq\ & \exists s_\mathsf{own}\ s_\mathsf{for}\ s_\mathsf{sub},\ s = s_\mathsf{own} \cup s_\mathsf{for} * s_\mathsf{own} \cup s_\mathsf{sub} \subseteq_\mathsf{cc} s_\mathsf{own} \cup s_\mathsf{for} \\ & * \mathsf{LocEv}(i, s_\mathsf{own}) * \mathsf{ForEv}(i, s_\mathsf{for}) * \mathsf{ForEv}(i, s_\mathsf{sub}) * \boxed{s_\mathsf{own}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{s_\mathsf{for}^{\frac{1}{2}}}^{\gamma_{\mathsf{for}_i}} * \boxed{s_\mathsf{sub}^{\frac{1}{3}}}^{\gamma_{\mathsf{sub}_i}} \\ & * \boxed{\bullet_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_\mathsf{cc}}(s_\mathsf{own} \cup s_\mathsf{for})}^{\gamma_{\mathsf{ccfor}_i}} * \boxed{\bullet_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_\mathsf{cc}}(s_\mathsf{own} \cup s_\mathsf{sub})}^{\gamma_{\mathsf{ccsub}_i}}\end{aligned}$$

The predicate $\mathbf{LR}(i, s)$ says that s can be broken up into two (disjoint) sets s_own and s_for. The sets are disjoint because every event in s_own originates at replica i, whereas all events in s_for originate outside of replica i. This is expressed by the predicates $\mathsf{LocEv}(i, p) = \forall e \in p.e.s = i$ and $\mathsf{ForEv}(i, p) = \forall e \in p.\ e.s \neq i$, respectively. Additionally, there is a third set s_sub which is a subset of s_for (this is implied by $s_\mathsf{own} \cup s_\mathsf{sub} \subseteq_\mathsf{cc} s_\mathsf{own} \cup s_\mathsf{for}$). The intuition for s_own and s_for is that they precisely track the set of events that have been delivered at replica i and originate at i or outside of i, respectively. However, the user at replica i is not aware of all those events: specifically, while the user is aware of (the effects of) all its local events, it might not have observed all events that originate outside of i. The set s_sub precisely tracks the set of remote events, $\mathsf{ForEv}(i, s_\mathsf{sub})$, that replica i has observed and is aware of. The tracking of all these event sets is precise because of the ownership of the fractions $\boxed{s_\mathsf{own}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{s_\mathsf{for}^{\frac{1}{2}}}^{\gamma_{\mathsf{for}_i}} * \boxed{s_\mathsf{sub}^{\frac{1}{3}}}^{\gamma_{\mathsf{sub}_i}}$. Additionally, the invariant has "read-only" access to the three pieces of ghost state because it does not possess the full fractions. Before pressing on with the definition of $\mathbf{LR}(i, s)$, let us look at the definition of local state and snapshot from Figure 3:

$$\begin{aligned}&\mathsf{LocSt}\,(i,\ s_\mathsf{own},\ s_\mathsf{sub}) \triangleq \boxed{s_\mathsf{own}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{s_\mathsf{sub}^{\frac{2}{3}}}^{\gamma_{\mathsf{sub}_i}} * \mathsf{LocSnap}(i, s_\mathsf{own}, s_\mathsf{sub}) \\ &\mathsf{LocSnap}(i, s_\mathsf{own}, s_\mathsf{sub}) \triangleq \mathsf{LocEv}(i, s_\mathsf{own}) * \mathsf{ForEv}(i, s_\mathsf{sub}) * \boxed{\circ_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_\mathsf{cc}}(s_\mathsf{own} \cup s_\mathsf{sub})}^{\gamma_{\mathsf{ccsub}_i}}\end{aligned}$$

Notice that ownership of the local state resource gives us knowledge of some but not all of the missing fractions for s_{own} and s_{sub}: $\boxed{s_{\mathsf{own}}}^{\gamma_{\mathsf{own}_i}}_{\frac{1}{3}} * \boxed{s_{\mathsf{sub}}}^{\gamma_{\mathsf{sub}_i}}_{\frac{2}{3}}$. This corresponds to our intuition that s_{sub} tracks the set of events the user is aware of. Notice that local state does not contain a fraction of s_{for}, because otherwise the library could not accept remote updates in a background thread.

We next explain the use of the PCM $\mathbb{M} \triangleq \textsc{Mono}(\subseteq_{\mathsf{cc}})$, where $\subseteq_{\mathsf{cc}}\colon \mathsf{Event} \times \mathsf{Event}$. We use two instances of this PCM, each under a different family of ghost names: $\gamma_{\mathsf{ccsub}_i}$ and $\gamma_{\mathsf{ccfor}_i}$. The local state contains just $\gamma_{\mathsf{ccsub}_i}$ and the invariant contains both $\gamma_{\mathsf{ccsub}_i}$ and $\gamma_{\mathsf{ccfor}_i}$. The intuition for holding the fragmental part $\circ_{\mathbb{M}} \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s_{\mathsf{own}} \cup s_{\mathsf{sub}})$ is to allow us to prove inclusion of local snapshots (LocSnapIncl, Figure 3).

We can now sketch the proof of the causality lemma. This proof is representative of our methodology: use the properties of the different PCMs to identify parts of a (valid) global state we are currently in, and then rely on a model-level lemma to get the result we want.

▶ **Lemma 5** (Causality, Figure 3). $\mathsf{GlobInv} \twoheadrightarrow \mathsf{LocSt}\,(i,\, o,\, s) \twoheadrightarrow \mathsf{GlobSnap}(h) \twoheadrightarrow \Rrightarrow_{\iota} \mathsf{LocSt}\,(i,\, o,\, s) *$
$\forall e\ e', e \in h \Rightarrow e' \in o \cup s \Rightarrow e <_t e' \Rightarrow e \in o \cup s.$

Proof. We open the global invariant and learn that the current global state $(s_{\mathsf{g}}, \vec{s_l})$ is valid. We also obtain global resources $\boxed{s_{\mathsf{g}}}^{\gamma_{\mathsf{gst}}}_{\frac{1}{3}} * \boxed{\bullet_{\mathbb{S}}\, s_{\mathsf{g}}}^{\gamma_{\mathsf{gsnap}}}$ and local resources $\boxed{s_{\mathsf{own}}}^{\gamma_{\mathsf{own}_i}}_{\frac{1}{3}} * \boxed{s_{\mathsf{sub}}}^{\gamma_{\mathsf{sub}_i}}_{\frac{1}{3}}$, where $s_{\mathsf{own}} \cup s_{\mathsf{sub}} \subseteq_{\mathsf{cc}} s_{\mathsf{own}} \cup s_{\mathsf{for}} = s_{l,\,i}$. From $\mathsf{LocSt}\,(i,\, o,\, s)$ we get $\boxed{o}^{\gamma_{\mathsf{own}_i}}_{\frac{1}{3}} * \boxed{s}^{\gamma_{\mathsf{sub}_i}}_{\frac{2}{3}}$, which tells us that $o = s_{\mathsf{own}}$ and $s = s_{\mathsf{sub}}$. From $\mathsf{GlobSnap}(h)$ we get $\boxed{\circ_{\mathbb{S}}\, h}^{\gamma_{\mathsf{gsnap}}}$, which tells us that $h \subseteq s_{\mathsf{g}}$. This means we have $e \in s_{\mathsf{g}}$ and e' is in a causally-closed subset of $s_{l,\,i}$, so we can finish by applying Lemma 4. ◀

There are additional two resources in the definition of $\mathbf{LR}(i, s)$: $\boxed{s_{\mathsf{for}}}^{\gamma_{\mathsf{for}_i}}_{\frac{1}{2}}$ and $\boxed{\bullet_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s_{\mathsf{own}} \cup s_{\mathsf{for}})}^{\gamma_{\mathsf{ccfor}_i}}$. These are used in the definition of the lock invariant and socket protocol, respectively. We explain them in the next section.

In addition to proving the resource lemmas from Figure 3, we also need to show that the PCMs we have chosen can make frame-preserving updates that are compatible with the two transitions (TUpdate and TMerge) from Definition 1. We refer to reader to our Coq formalization for details.

7.4 Safety Proof

StateLib's `init` function allocates a reference with the CRDT's initial state and spawns two concurrent threads: `apply` receives states from other replicas and merges them with the current state, and `broadcast` regularly sends the current state to all other replicas. Access to the (shared) local state is coordinated via a spinlock. The associated *lock invariant* [1], defined by the predicate $\mathbf{LI}(i, \ell)$ below, is the key ingredient of the library's safety proof (ℓ is the memory location holding the CRDT's state). When a thread acquires the lock, it gets to assume $\mathbf{LI}(i, \ell)$; unlike a regular invariant, which needs to be restored after a single atomic step, a lock invariant need not be restored until the thread releases the lock.

$$\mathbf{LI}(i, \ell) \triangleq \exists st\ s_{\mathsf{own}}\ s_{\mathsf{for}}.\ \mathsf{LocEv}(i, s_{\mathsf{own}}) * \mathsf{ForEv}(i, s_{\mathsf{for}}) * \ell \mapsto_{\mathsf{ip}} st$$
$$* [\![s_{\mathsf{own}} \cup s_{\mathsf{for}}]\!] = st * \boxed{s_{\mathsf{own}}}^{\gamma_{\mathsf{own}_i}}_{\frac{1}{3}} * \boxed{s_{\mathsf{for}}}^{\gamma_{\mathsf{for}_i}}_{\frac{1}{2}}$$

The lock invariant says that the CRDT's state is always the denotation of some set of events $s_{\mathsf{own}} \cup s_{\mathsf{for}}$ (ip is the IP address of replica i). Additionally, the invariant holds resources

$\boxed{s_{\mathsf{own}}}^{\frac{1}{3}\,\gamma_{\mathsf{own}_i}} * \boxed{s_{\mathsf{for}}}^{\frac{1}{2}\,\gamma_{\mathsf{for}_i}}$ which guarantee that the we are "in sync" with the logical state recorded for replica i in the global invariant GlobInv. Notice the lock invariant keeps γ_{for_i} and not γ_{sub_i} because the library knows exactly the set of foreign events that have been processed so far. The table below summarizes where the different fractions of γ_{own_i}, γ_{for_i} and γ_{sub_i} are kept:

	γ_{own_i}	γ_{for_i}	γ_{sub_i}
Global invariant	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{1}{3}$
Lock invariant	$\frac{1}{3}$	$\frac{1}{2}$	
Local state	$\frac{1}{3}$		$\frac{2}{3}$

As part of the proof we also need to define StateLib's socket protocol $\mathbf{SP}(i, st)$; i.e. a predicate that holds for all states st received by a replica (which dually creates a proof obligation whenever a replica messages others). The abridged version below assumes that st is already deserialized and that i is the replica id of the message's sender:

$$\begin{aligned}\mathbf{SP}(i, st) \triangleq \exists s'_{\mathsf{own}}\ s'_{\mathsf{for}}.\ \mathsf{LocEv}(i, s'_{\mathsf{own}}) * \mathsf{ForEv}(i, s'_{\mathsf{for}}) * [\![s'_{\mathsf{own}} \cup s'_{\mathsf{for}}]\!] = st \\ * \mathsf{LocStValid}(s'_{\mathsf{own}} \cup s'_{\mathsf{for}}) * \boxed{\circ_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s'_{\mathsf{own}} \cup s'_{\mathsf{for}})}^{\gamma_{\mathsf{ccfor}_i}}\end{aligned}$$

The socket protocol assumes that the received state st is the denotation of the union $s'_{\mathsf{own}} \cup s'_{\mathsf{for}}$, where s'_{own} and s'_{for} are event sets that are local and foreign, respectively, relative to the message's sender (not relative to the receiver). Additionally, we know that $s'_{\mathsf{own}} \cup s'_{\mathsf{for}}$ is a causally closed subset of the events recorded at replica i (in particular, we know the sender is not accidentally including events that have not been previously recorded).

Since both `apply` and `broadcast` recurse forever, their specifications are not very interesting: in particular, we do not care about their post-conditions. We do care about preserving the lock and global invariants as they execute. We briefly sketch the proof of `apply`. Before `apply` updates the CRDT state via `st := merge !st st'`, we know that all the following hold: the global invariant GlobInv, the lock invariant $\mathbf{LI}(i, \mathtt{st})$, and the socket protocol $\mathbf{SP}(j, st')$, where i and j are the ids of the local and sender replicas, respectively, and $i \neq j$. We open the lock invariant and get $\mathtt{!st} = [\![s_{\mathsf{own}} \cup s_{\mathsf{for}}]\!] * \boxed{s_{\mathsf{own}}}^{\frac{1}{3}\,\gamma_{\mathsf{own}_i}} * \boxed{s_{\mathsf{for}}}^{\frac{1}{2}\,\gamma_{\mathsf{for}_i}}$. Similarly, from the socket protocol we know that $\mathtt{st'} = [\![s'_{\mathsf{own}} \cup s'_{\mathsf{for}}]\!] * \mathsf{LocStValid}(s'_{\mathsf{own}} \cup s'_{\mathsf{for}}) * \boxed{\circ_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s'_{\mathsf{own}} \cup s'_{\mathsf{for}})}^{\gamma_{\mathsf{ccfor}_i}}$. We would like to apply the coherence lemma (MergeCoh), which tells us that merging two states is the same as (1) merging the corresponding events that generated those states, and (2) then taking the denotation of the union of the event sets. The premises of (MergeCoh) all follow from local state validity after opening the global invariant, and from the fact that $\boxed{\circ_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s'_{\mathsf{own}} \cup s'_{\mathsf{for}})}^{\gamma_{\mathsf{ccfor}_i}}$ proves that the events we are merging have been previously recorded. The resulting logical state is $\boxed{s_{\mathsf{own}}}^{\frac{1}{3}\,\gamma_{\mathsf{own}_i}} * \boxed{(s_{\mathsf{for}} \cup s'_{\mathsf{own}} \cup \{e \in s'_{\mathsf{for}} | e.s \neq i\})}^{\frac{1}{2}\,\gamma_{\mathsf{for}_i}}$.

The proof of `get_state` uses the lock invariant to conclude that the returned state is the denotation of the set of events received so far. It then uses the global invariant: specifically the relation $s_{\mathsf{own}} \cup s_{\mathsf{sub}} \subseteq_{\mathsf{cc}} s_{\mathsf{own}} \cup s_{\mathsf{for}}$ in the definition of $\mathbf{LR}(i, s)$ to update the local state to $\mathsf{LocSt}\,(i,\, o,\, f')$. The proof of `update` uses the preservation of global validity under a TUpdate transition (Theorem 3) to show that GlobInv is preserved. We refer the reader to our formalization for additional details.

8 Verified CRDTs

To test StateLib we verified five example CRDTs from the literature: grow-only counter (g-counter), grow-only set, product combinator, map combinator, and positive-negative

counter (pn-counter). We also verified a closed program consisting of client code that uses the pn-counter. This closed program appears in Nieto et al. [20], and we were able to swap out their op-based pn-counter for our state-based version without modifying their safety proof.[10] We used the closed example as a case study in giving our state-based CRDT the same specification as its op-based counterpart, hiding implementation details in the process.

In this section we focus on describing the pn-counter and the closed program. A pn-counter is a data structure that supports two operations: `add(z)` adds the integer `z`, which may be negative, to the counter, and `get_value` returns the counter's current value. The initial value is 0. The denotation for pn-counter used in Nieto et al. [20] is $[\![s]\!] = \sum_{e \in s} e.o$. We implemented the counter as a wrapper over the product `prod(g-counter, g-counter)`. We now explain how `g-counter` and `prod` work.

The g-counter is a simpler version of pn-counter where we can only add non-negative numbers. Operations are of the form `add(n)` with $n \in \mathbb{N}$. If N is the number of CRDT replicas, the lattice state is a vector $\vec{c}$ with N entries. The ith entry tracks the contribution of replica i to the counter's state. The initial value is the 0-vector of length N. The mutator is defined as $\mathsf{mut}(\vec{c}, e) = s[e.s \mapsto \vec{c}_{e.s} + e.o]$, and merges are done by taking the maximum of two vectors pointwise. The denotation $[\![s]\!]$ is the pointwise sum of all the vectors in s.

Given two CRDTs $C_A = (\mathsf{Op}_A, \mathsf{St}_A, \mathsf{init}_A, \mathsf{mut}_A, \mathsf{merge}_A)$ and $C_B = (\mathsf{Op}_B, \mathsf{St}_B, \mathsf{init}_B, \mathsf{mut}_B, \mathsf{merge}_B)$, their product is $C_{A\times B} = (\mathsf{Op}_A \times \mathsf{Op}_B, \mathsf{St}_A \times \mathsf{St}_B, \mathsf{init}_A \times \mathsf{init}_B, \mathsf{mut}_{A\times B}, \mathsf{merge}_{A\times B})$. Both the mutator and merge function operate in a component-wise fashion. The denotation $[\![s]\!]$ splits s into two sets s_A and s_B of A-events and B-events, respectively. It then computes the pair $([\![s_A]\!]_A, [\![s_B]\!]_B)$.

We then implement the pn-counter as a wrapper over the product `prod(g-counter, g-counter)`. Specifically, we wrap the product's `get_state` and `update` as follows (`sum_entries` is the function that sums all the entries of a vector):

```
let pncounter_add z =
  if z >= 0 then prod_update((z, 0))
            else prod_update((0, -z))
```

```
let pncounter_get_value () =
  let (v1, v2) = prod_get_state () in
  (sum_entries v1) - (sum_entries v2)
```

Finally, we describe the closed program we verified. This example shows client code interacting with a pn-counter. The relevant snippet is shown below: we have two replicas, A and B, each of which increments the counter, reads it, and then asserts that the read returns one of two possible values. In A the possible values are 1 and 3, depending on whether the remote operation `add 2` has been propagated from B to A by the time the read happens. In either case, the `add 1` must be visible by the subsequent read because the latter happened later according to program order: this is the so-called *reads-follow-writes* session guarantee [8]. An analogous situation happens for replica B.

```
(* replica A *)
add 1;
let r = get_value () in
assert (r = 1 || r = 3)
```

```
(* replica B *)
add 2;
let r = get_value () in
assert (r = 2 || r = 3)
```

The reader can consult Nieto et al. [20] for more details on the proof that the assertions in the example above do not fail, but the important part is that we were able to swap out the op-based counter implementation by our state-based counter, while (almost) keeping the proof intact. The two implementations are quite different: the op-based implementation relies

[10] We have added one additional property to the common CRDT resource interface (an excerpt of which was shown in Figure 3), and thus we extended the proof of Nieto et al. [20] such that this additional property is also proved for their op-based library.

on a causal broadcast algorithm, while ours does not and instead relies on lattice properties, as well as on applications of the product combinator. The fact that both implementations meet the same specification is good evidence that the denotation-based separation logic specifications are general, flexible, and can hide implementation details.

9 Related Work

As previously mentioned, the idea of specifying CRDTs as a partial function from event histories (including causality metadata) to the data type's state comes from Burckhardt et al. [3]. We also learned from their paper that state-based CRDTs can be thought of as transitively delivering (the effects of) events when states are merged, which in turn makes it possible to prove, as we have done, that state-based CRDTs are causally-consistent. [11]

Leijnse et al. [13] reformulate the above specification style so it can be better applied to higher-order combinators, coining the term CRDT denotation. They work solely with specifications, and do not implement these combinators nor verify an implementation.

The idea of tracking the state of a concurrent data structure via a logical set of operations that is divided into contributions by the current thread and those originating from other threads, as in our $\mathsf{LocSt}(i, o, f)$ resource, has previously appeared in the context of the FCSL logic (where they are termed "self" and "other" contributions, respectively) [19, 5].

The Compass separation logic framework [4] (also Iris-based) can be used to specify and verify functional correctness of concurrent data structures in a relaxed memory model. There are a number of commonalities with our work: their specs are also given as logically-atomic triples that track the state of a data structure as a function of the set of writes that are visible by a given thread. They develop a notion of logical *views* that is similar to our local snapshots $\mathsf{LocSnap}(i, o, f)$ (without the distinction between own and foreign events): ownership of a view provides a lower bound on the set of observed events, and the views contain logical metadata that tracks the happens-before relation between writes. The main difference with our work is that we operate at the intersection of weak consistency and message passing, whereas their work is in the context of shared memory.

Zeller et al. [24] implement and formally verify multiple state-based CRDTs in Isabelle/HOL. To our knowledge, they are the first to explicitly link denotation-style specifications to their lattice-based implementations. Like us, they prove both convergence and functional correctness. There are two main differences with our work: their system model is an STS where the states map each replica id to the replica state (this is very similar to our STS model from Section 7.1). By contrast, in our work the system model is the operational semantics of $\mathsf{AnerisLang}$, with support for mutation, node-local concurrency, higher-order functions, etc. This means that while their technique can only be used to reason about a CRDT in isolation, ours can verify a system where the CRDT and a client (or a larger distributed system of which the CRDT is a small part) are executing together, and where both of these are implemented in a realistic programming language. The second difference is that in Zeller et al.'s work one needs to come up with a different invariant that implies coherence between the denotation and the lattice for each implemented CRDT, as well as proving for each example that the invariant is preserved by the different transitions. In our work, we prove just one invariant, global state validity, and the CRDT implementer then needs to prove coherence given that global validity holds.

[11] They do mention that state-based CRDTs are causally-consistent, but there is no formal proof, or even a precise lemma statement.

Nair et al. [18] verify several state-based CRDTs. These are not "pure" CRDTs in the sense that some of data structure's operations are at times disabled. For example, they show how to verify a distributed lock implemented as a CRDT (the release-the-lock operation can only be enabled if the local replica owns the lock). Proof wise, this means that sometimes it is useful to enforce example-specific invariants on the CRDT being verified: it would be interesting to modify StateLib so it can support these user-defined invariants and so can tackle the examples presented in their paper.

Gondelman et al. [8] verify functional correctness of a causally-consistent key-value store using Aneris. Their work introduces the encoding of causality in separation logic that we use. However, their key-value store is not a CRDT because it violates convergence. Additionally, their work is closer to op-based CRDTs because writes are propagated individually.

Timany et al. [23] develop an extension of Aneris called Trillium where one simultaneously proves both safety and also that the program being verified refines an STS model. Unlike in our work, where the STS is used just to prove safety, their refinement is *history-preserving*, which allows them to prove liveness properties as well. As one case study, they show that a state-based G-Counter CRDT is eventually consistent. They do not tackle any other (more complex) CRDTs, and their specification of the G-Counter relies on the fact that the G-Counter is monotonic. It would be interesting to recast our work using the Trillium methodology with the goal of showing that any CRDT implemented via StateLib is eventually-consistency (that is, additionally showing eventual delivery).

The closest related work is Nieto et al. [20], from which we inherit the CRDT resource interface from Figure 3, the modular structure of the implementation (a core library that can be instantiated for different CRDT examples), as well as the general structure of the proof: a state-transition system model that is embedded in the logic, as well as a lock invariant that ties the denotation-style specification to the lattice-based state. The main difference is that their paper deals exclusively with op-based CRDTs: as mentioned in Section 4.2 adapting their technique to the state-based setting leads to a number of technical challenges we had to solve in our approach.

10 Conclusions

We have shown how to give modular specifications to realistic state-based CRDT implementations using the Aneris separation logic. Our specifications show both convergence and functional correctness relative to an abstract denotational model of the CRDT.

We have explored our approach by implementing and verifying a library, StateLib, for building state-based CRDTs. Our library takes as input a purely-functional implementation of a state-based CRDT's core logic, together with coherence proofs between the CRDT's lattice-based and denotation-based models. The library then produces as output a fully-fledged CRDT that is replicated over multiple nodes. Using the library we implemented and verified multiple example CRDTs from the literature.

When taken together with Nieto et al. [20], our work presents a unified framework for the specification and verification of op- and state-based CRDTs. To show that we can abstract away from the fact that a CRDT is state-based, we re-prove Nieto et al.'s interface for resources tracking CRDT state using a new definition of resources that is compatible with state-based CRDTs. We test this approach by showing that one can start with a client program that uses an op-based counter CRDT, swap out the counter's implementation by our state-based implementation, and recover the safety proof for the entire closed program while making minimal changes to the original proof.

References

1 Lars Birkedal and Aleš Bizjak. Lecture notes on iris: Higher-order concurrent separation log. 2017. URL: http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf.

2 Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, 1991. doi:10.1145/128738.128742.

3 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2014, pages 271–284. ACM, January 2014. doi:10.1145/2535838.2535848.

4 Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. Compass: strong and compositional library specifications in relaxed memory separation logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 792–808, 2022.

5 Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Concurrent data structures linked in time. *arXiv preprint arXiv:1604.08080*, 2016.

6 Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. Bounding data races in space and time. In *PLDI*, pages 242–255. ACM, 2018.

7 Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA):109:1–109:28, 2017. doi:10.1145/3133933.

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

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

10 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980.

11 Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 336–365. Springer, 2020.

12 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

13 Adriaan Leijnse, Paulo Sérgio Almeida, and Carlos Baquero. Higher-order patterns in replicated data types. In *PaPoC@EuroSys*, pages 5:1–5:6. ACM, 2019. doi:10.1145/3301419.3323971.

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

15 Barbara H. Liskov and Stephen N. Zilles. Programming with abstract data types. In *SIGPLAN Symposium on Very High Level Languages*, pages 50–59. ACM, 1974.

16 Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. Verifying replicated data types with typeclass refinements in liquid haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA):216:1–216:30, 2020. doi:10.1145/3428284.

17 Kartik Nagar and Suresh Jagannathan. Automated parameterized verification of crdts. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 459–477. Springer, 2019.

18 Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. Proving the safety of highly-available distributed objects. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 544–571. Springer, 2020.

19 Aleksandar Nanevski. Separation logic and concurrency. oregon programming languages summer school, 2016.

20 Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. Modular verification of op-based crdts in separation logic. *Proc. ACM Program. Lang. OOPSLA (2022). Accepted for publication*, 2022.

21 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Research Report 7506, INRIA, January 2011. URL: `http://hal.inria.fr/inria-00555588/`.

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

23 Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal. Trillium: Unifying refinement and higher-order distributed separation logic. *CoRR*, abs/2109.07863, 2021.

24 Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of crdts. In *FORTE*, volume 8461 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2014. `doi:10.1007/978-3-662-43613-4_3`.