

Automated Parameterized Verification of CRDTs (Extended Version)

Kartik Nagar and Suresh Jagannathan

Purdue University, USA
{nagark,suresh}@cs.purdue.edu

Abstract. Maintaining multiple replicas of data is crucial to achieving scalability, availability and low latency in distributed applications. *Conflict-free Replicated Data Types* (CRDTs) are important building blocks in this domain because they are designed to operate correctly under the myriad behaviors possible in a weakly-consistent distributed setting. Because of the possibility of concurrent updates to the same object at different replicas, and the absence of any ordering guarantees on these updates, *convergence* is an important correctness criterion for CRDTs. This property asserts that two replicas which receive the same set of updates (in any order) must nonetheless converge to the same state. One way to prove that operations on a CRDT converge is to show that they commute since commutative actions by definition behave the same regardless of the order in which they execute. In this paper, we present a framework for automatically verifying convergence of CRDTs under different weak-consistency policies. Surprisingly, depending upon the consistency policy supported by the underlying system, we show that not all operations of a CRDT need to commute to achieve convergence. We develop a proof rule parameterized by a consistency specification based on the concepts of *commutativity modulo consistency policy* and *non-interference to commutativity*. We describe the design and implementation of a verification engine equipped with this rule and show how it can be used to provide the first automated convergence proofs for a number of challenging CRDTs, including sets, lists, and graphs.

1 Introduction

For distributed applications, keeping a single copy of data at one location or multiple fully-synchronized copies (i.e. state-machine replication) at different locations, makes the application susceptible to loss of availability due to network and machine failures. On the other hand, having multiple un-synchronized replicas of the data results in high availability, fault tolerance and uniform low latency, albeit at the expense of consistency. In the latter case, an update issued at one replica can be asynchronously transmitted to other replicas, allowing the system to operate continuously even in the presence of network or node failures [8]. However, mechanisms must now be provided to ensure replicas are kept consistent with each other in the face of concurrent updates and arbitrary re-ordering of such updates by the underlying network.

Over the last few years, *Conflict-free Replicated Datatypes* (CRDTs) [19,20,18] have emerged as a popular solution to this problem. In op-based CRDTs, when an operation on a CRDT instance is issued at a replica, an *effector* (basically an update function) is generated locally, which is then asynchronously transmitted (and applied) at all other replicas.¹ Over the years, a number of CRDTs have been developed for common datatypes such as maps, sets, lists, graphs, etc.

The primary correctness criterion for a CRDT implementation is *convergence* (sometimes called *strong eventual consistency* [9,19] (SEC)): two replicas which have received the same set of effectors must converge to the same CRDT state. Because of the weak default guarantees assumed to be provided by the underlying network, however, we must consider the possibility that effectors can be applied in arbitrary order on different replicas, complicating correctness arguments. This complexity is further exacerbated because CRDTs impose no limitations on how often they are invoked, and may assume additional properties on network behaviour [14] that must be taken into account when formulating correctness arguments.

Given these complexities, verifying convergence of operations in a replicated setting has proven to be challenging and error-prone [9]. In response, several recent efforts have used mechanized proof assistants to yield formal machine-checked proofs of correctness [9,23]. While mechanization clearly offers stronger assurance guarantees than handwritten proofs, it still demands substantial manual proof engineering effort to be successful. In particular, correctness arguments are typically given in terms of constraints on CRDT states that must be satisfied by the underlying network model responsible for delivering updates performed by other replicas. Relating the state of a CRDT at one replica with the visibility properties allowed by the underlying network has typically involved constructing an intricate simulation argument or crafting a suitably precise invariant to establish convergence. This level of sophisticated reasoning is required for every CRDT and consistency model under consideration. There is a notable lack of techniques capable of reasoning about CRDT correctness under different weak consistency policies, even though such techniques exist for other correctness criteria such as preservation of state invariants [10,11] or serializability [4,15] under weak consistency.

To overcome these challenges, we propose a novel *automated* verification strategy that does not require complex proof-engineering of handcrafted simulation arguments or invariants. Instead, our methodology allows us to directly connect constraints on events imposed by the consistency model with constraints on states required to prove convergence. Consistency model constraints are extracted from an axiomatization of network behavior, while state constraints are generated using reasoning principles that determine the *commutativity* and *non-interference* of sequences of effectors, subject to these consistency constraints. Both sets of constraints can be solved using off-the-shelf theorem provers. Be-

¹ In this work, we focus on the op-based CRDT model; however, our technique naturally extends to state-based CRDTs since they can be emulated by an op-based model[19]

cause an important advantage of our approach is that it is parametric on weak consistency schemes, we are able to analyze the problem of CRDT convergence under widely different consistency policies (e.g., eventual consistency, causal consistency, parallel snapshot isolation (PSI) [22], among others), and for the first time verify CRDT convergence under such stronger models (efficient implementations of which are supported by real-world data stores). A further pleasant by-product of our approach is a pathway to take advantage of such stronger models to simplify existing CRDT designs and allow composition of CRDTs to yield new instantiations for more complex datatypes.

The paper makes the following contributions:

1. We present a proof methodology for verifying the correctness of CRDTs amenable to automated reasoning.
2. We allow the proof strategy to be parameterized on a weak consistency specification that allows us to state correctness arguments for a CRDT based on constraints imposed by these specifications.
3. We experimentally demonstrate the effectiveness of our proposed verification strategy on a number of challenging CRDT implementations across multiple consistency schemes.

Collectively, these contributions yield (to the best of our knowledge) the first automated and parameterized proof methodology for CRDT verification.

The remainder of the paper is organized as follows. In the next section, we provide further motivation and intuition for our approach. Sec. 3 formalizes the problem definition, providing an operational semantics and axiomatizations of well-known consistency specifications. Sec. 4 describes our proof strategy for determining CRDT convergence that is amenable to automated verification. Sec. 5 provides details about our implementation and experimental results justifying the effectiveness of our framework. Secs. 6 presents related work and conclusions.

2 Illustrative Example

$$\begin{aligned} S &\in \mathbb{P}(E) \\ \text{Add}(a) : S &\quad \lambda S'. S' \cup \{a\} \\ \text{Remove}(a) : S &\quad \lambda S'. S' \setminus \{a\} \\ \text{Lookup}(a) : S &\quad a \in S \end{aligned}$$

Fig. 1: A simple Set CRDT definition.

We illustrate our approach using a Set CRDT specification as a running example. A CRDT $(\Sigma, O, \sigma_{\text{init}})$ is characterized by a set of states Σ , a set of operations O and an initial state $\sigma_{\text{init}} \in \Sigma$, where each operation $o \in O$ is a function with signature $\Sigma \rightarrow (\Sigma \rightarrow \Sigma)$. The state of a CRDT is replicated, and when operation o is issued

at a replica with state σ , the effector $o(\sigma)$ is generated, which is immediately applied at the local replica (which we also call the *source* replica) and transmitted to all other replicas, where it is subsequently applied upon receipt.

Additional constraints on the order in which effectors can be received and applied at different replicas are specified by a consistency policy, discussed below.

In the absence of any such additional constraints, however, we assume the underlying network only offers *eventually consistent* guarantees - all replicas eventually receive all effectors generated by all other replicas, with no constraints on the order in which these effectors are received.

Consider the simple **Set** CRDT definition shown in Fig. 1. Let E be an arbitrary set of elements. The state space Σ is $\mathbb{P}(E)$. $\text{Add}(\mathbf{a}) : \mathbf{S}$ denotes the operation $\text{Add}(\mathbf{a})$ applied on a replica with state \mathbf{S} , which generates an effector which simply adds \mathbf{a} to the state of all other replicas it is applied to. Similarly, $\text{Remove}(\mathbf{a}) : \mathbf{S}$ generates an effector that removes \mathbf{a} on all replicas to which it is applied. $\text{Lookup}(\mathbf{a}) : \mathbf{S}$ is a query operation which checks whether the queried element is present in the source replica \mathbf{S} .

A CRDT is *convergent* if during any execution, any two replicas which have received the same set of effectors have the same state. Our strategy to prove convergence is to show that any two effectors of the CRDT pairwise commute with each other modulo a consistency policy, i.e. for two effectors e_1 and e_2 , $e_1 \circ e_2 = e_2 \circ e_1$. Our simple **Set** CRDT clearly does not converge when executed on an eventually consistent data store since the effectors $e_1 = \text{Add}(\mathbf{a}) : \mathbf{S}_1$ and $e_2 = \text{Remove}(\mathbf{a}) : \mathbf{S}_2$ do not commute, and the semantics of eventual consistency imposes no additional constraints on the visibility or ordering of these operations that could be used to guarantee convergence. For example, if e_1 is applied to the state at some replica followed by the application of e_2 , the resulting state does not include the element \mathbf{a} ; conversely, applying e_2 to a state at some replica followed by e_1 leads to a state that does contain the element \mathbf{a} .

However, while commutativity is a sufficient property to show convergence, it is not always a necessary one. In particular, different consistency models impose different constraints on the visibility and ordering of effectors that can obviate the need to reason about their commutativity. For example, if the consistency model enforces $\text{Add}(\mathbf{a})$ and $\text{Remove}(\mathbf{a})$ effectors to be applied in the same order at all replicas, then the **Set** CRDT will converge. As we will demonstrate later, the PSI consistency model exactly matches this requirement. To further illustrate this, consider the definition of the **ORSet** CRDT shown in Fig. 2. Here, every element is tagged with a unique identifier (coming from the set I). $\text{Add}(\mathbf{a}, \mathbf{i}) : \mathbf{S}$ simply adds the element \mathbf{a} tagged with \mathbf{i} ², while $\text{Remove}(\mathbf{a}) : \mathbf{S}$ returns an effector that when applied to a replica state will remove all tagged versions of \mathbf{a} that were present in \mathbf{S} , the source replica.

$$\begin{aligned} & \mathbf{S} \in \mathbb{P}(E \times I) \\ \text{Add}(\mathbf{a}, \mathbf{i}) : \mathbf{S} & \\ & \lambda \mathbf{S}' . \mathbf{S}' \cup \{(\mathbf{a}, \mathbf{i})\} \\ \text{Remove}(\mathbf{a}) : \mathbf{S} & \\ & \lambda \mathbf{S}' . \mathbf{S}' \setminus \{(\mathbf{a}, \mathbf{i}) : (\mathbf{a}, \mathbf{i}) \in \mathbf{S}\} \\ \text{Lookup}(\mathbf{a}) : \mathbf{S} & \\ & \exists (\mathbf{a}, \mathbf{i}) \in \mathbf{A} \end{aligned}$$

Fig. 2: A definition of an **ORSet** CRDT.

² Assume that every call to Add uses a unique identifier, which can be easily arranged, for example by keeping a local counter at every replica which is incremented at every operation invocation, and using the id of the replica and the value of the counter as a unique identifier

Suppose $e_1 = \text{Add}(\mathbf{a}, \mathbf{i}) : S_1$ and $e_2 = \text{Remove}(\mathbf{a}) : S_2$. If it is the case that S_2 does not contain (\mathbf{a}, \mathbf{i}) , then these two effectors are guaranteed to commute because e_2 is unaware of (\mathbf{a}, \mathbf{i}) and thus behaves as a no-op with respect to effector e_1 when it is applied to any replica state. Suppose, however, that e_1 's effect was visible to e_2 ; in other words, e_1 is applied to S_2 before e_2 is generated. There are two possible scenarios that must be considered. (1) Another replica (call it S') has e_2 applied before e_1 . Its final state reflects the effect of the **Add** operation, while S_2 's final state reflects the effect of applying the **Remove**; clearly, convergence is violated in this case. (2) All replicas apply e_1 and e_2 in the same order; the interesting case here is when the effect of e_1 is always applied before e_2 on every replica. The constraint that induces an effector order between e_1 and e_2 on every replica as a consequence of e_1 's visibility to e_2 on S_2 is supported by a causally consistent distributed storage model. Under causal consistency, whenever e_2 is applied to a replica state, we are guaranteed that e_1 's effect, which adds (\mathbf{a}, \mathbf{i}) to the state, would have occurred. Thus, even though e_1 and

$$\begin{aligned} S &\in \mathbb{P}(E \times I) \times \mathbb{P}(E \times I) \\ \text{Add}(\mathbf{a}, \mathbf{i}) : (A, R) & \\ \lambda(A', R') . (A' \cup \{(\mathbf{a}, \mathbf{i})\}, R') & \\ \text{Remove}(\mathbf{a}) : (A, R) & \\ \lambda(A', R') . (A', R' \cup \{(\mathbf{a}, \mathbf{i}) : (\mathbf{a}, \mathbf{i}) \in A\}) & \\ \text{Lookup}(\mathbf{a}) : (A, R) & \\ \exists (\mathbf{a}, \mathbf{i}) \in A \wedge (\mathbf{a}, \mathbf{i}) \notin R & \end{aligned}$$

Fig. 3: A variant of the ORSet using tombstones.

visibility of effectors are instead used to guide and simplify our analysis. Applying this notion to pairs of effectors in arbitrarily long executions requires incorporating commutativity properties under a more general induction principle to allow us to generalize the commutativity of effectors in bounded executions to the unbounded case. This generalization forms the heart of our automated verification strategy.

Fig. 3 defines an ORSet with “tombstone” markers used to keep track of deleted elements in a separate set. Our proof methodology is sufficient to automatically show that this CRDT converges under EC.

3 Problem Definition

In this section, we formalize the problem of determining convergence in CRDTs parametric to a weak consistency policy. First, we define a general operational semantics to describe all valid executions of a CRDT under any given weak consistency policy. As stated earlier, a CRDT program \mathcal{P} is specified by the tuple $(\Sigma, O, \sigma_{\text{init}})$. Here, we find it convenient to define an operation $o \in O$ as

a function $(\Sigma \times (\Sigma \rightarrow \Sigma)^*) \rightarrow (\Sigma \rightarrow \Sigma)$. Instead of directly taking as input a generating state, operations are now defined to take as input a start state and a sequence of effectors. The intended semantics is that the sequence of effectors would be applied to the start state to obtain the generating state. Using this syntax allows us simplify the presentation of the proof methodology in the next section, since we can abstract a history of effectors into an equivalent start state.

Formally, if $\hat{o} : \Sigma \rightarrow (\Sigma \rightarrow \Sigma)$ was the original op-based definition, then we define the operation $o : (\Sigma \times (\Sigma \rightarrow \Sigma)^*) \rightarrow (\Sigma \rightarrow \Sigma)$ as follows:

$$\begin{aligned} \forall \sigma. \quad o(\sigma, \epsilon) &= \hat{o}(\sigma) \\ \forall \sigma, \pi, f. \quad o(\sigma, \pi f) &= o(f(\sigma), \pi) \end{aligned}$$

Note that ϵ indicates the empty sequence. Hence, for all states σ and sequence of functions π , we have $o(\sigma, \pi) = \hat{o}(\pi(\sigma))$.

To define the operational semantics, we abstract away from the concept of replicas, and instead maintain a global pool of effectors. A new CRDT operation is executed against a CRDT state obtained by first selecting a subset of effectors from the global pool and then applying the elements in that set in some non-deterministically chosen permutation to the initial CRDT state. The choice of effectors and their permutation must obey the weak consistency policy specification. Given a CRDT $\mathcal{P} = (\Sigma, O, \sigma_{\text{init}})$ and a weak consistency policy Ψ , we define a **labeled transition system** $\mathcal{S}_{\mathcal{P}, \Psi} = (\mathcal{C}, \rightarrow)$, where \mathcal{C} is a set of configurations and \rightarrow is the transition relation. A **configuration** $c = (\Delta, \text{vis}, \text{eo})$ consists of three components : Δ is a set of events, $\text{vis} \subseteq \Delta \times \Delta$ is a *visibility* relation, and $\text{eo} \subseteq \Delta \times \Delta$ is a global *effector order* relation (constrained to be anti-symmetric). An **event** $\eta \in \Delta$ is a tuple $(\text{eid}, o, \sigma_s, \Delta_r, \text{eo})$ where eid is a unique event id, $o \in O$ is a CRDT operation, $\sigma_s \in \Sigma$ is the start CRDT state, Δ_r is the set of events visible to η (also called the history of η), and eo is a total order on the events in Δ_r (also called the local effector order relation). We assume projection functions for each component of an event (for example $\sigma_s(\eta)$ projects the start state of the event η).

Given an event $\eta = (\text{eid}, o, \sigma_s, \Delta_r, \text{eo})$, we define η^e to be the **effector** associated with the event. This effector is obtained by executing the CRDT operation o against the start CRDT state σ_s and the sequence of effectors obtained from the events in Δ_r arranged in the reverse order of eo . Formally,

$$\eta^e = \begin{cases} o(\sigma_s, \epsilon) & \text{if } \Delta_r = \phi \\ o(\sigma_s, \prod_{i=1}^k \eta_{P(i)}^e) & \text{if } \Delta_r = \{\eta_1, \dots, \eta_k\} \text{ where } P : \{1, \dots, k\} \rightarrow \{1, \dots, k\} \\ & \forall i, j. i < j \Rightarrow (\eta_{P(j)}, \eta_{P(i)}) \in \text{eo} \end{cases} \quad (1)$$

In the above definition, when Δ_r is non-empty, we define a permutation P of the events in Δ_r such that the permutation order is the inverse of the effector order eo . This ensures that if $(\eta_i, \eta_j) \in \text{eo}$, then η_j^e occurs before η_i^e in the sequence passed to the CRDT operation o , effectively applying η_i^e before η_j^e to obtain the generating state for o .

The following rule describes the transitions allowed in $\mathcal{S}_{\mathcal{P},\Psi}$:

$$\frac{\begin{array}{l} \Delta_r \subseteq \Delta \quad o \in O \quad \sigma_s \in \Sigma \quad \text{eo}_r \text{ is a total order on } \Delta_r \\ \text{eo} \subseteq \text{eo}_r \quad \text{fresh id} \quad \eta = (\text{id}, o, \sigma_s, \Delta_r, \text{eo}) \\ \Delta' = \Delta \cup \{\eta\} \quad \text{vis}' = \text{vis} \cup \{(\eta', \eta) \mid \eta' \in \Delta_r\} \quad \Psi(\Delta', \text{vis}', \text{eo}') \end{array}}{(\Delta, \text{vis}, \text{eo}) \xrightarrow{\eta} (\Delta', \text{vis}', \text{eo}')}$$

The rule describes the effect of executing a new operation o , which begins by first selecting a subset of already completed events (Δ_r) and a total order eo_r on these events which obeys the global effector order eo . This mimics applying the operation o on an arbitrary replica on which the events of Δ_r have been applied in the order eo_r . A new event (η) corresponding to the issued operation o is computed, which is used to label the transition and is also added to the current configuration. All the events in Δ_r are visible to the new event η , which is reflected in the new visibility relation vis' . The system moves to the new configuration $(\Delta', \text{vis}', \text{eo}')$ which must satisfy the consistency policy Ψ . Note that even though the general transition rule allows the event to pick any arbitrary start state σ_s , we restrict the start state of all events in a **well-formed execution** to be the initial CRDT state σ_{init} , i.e. the state in which all replicas begin their execution. A trace of $\mathcal{S}_{\mathcal{P},\Psi}$ is a sequence of transitions. Let $\llbracket \mathcal{S}_{\mathcal{P},\Psi} \rrbracket$ be the set of all finite traces. Given a trace τ , $L(\tau)$ denotes all events (i.e. labels) in τ .

Definition 1 (Well-formed Execution). *A trace $\tau \in \llbracket \mathcal{S}_{\mathcal{P},\Psi} \rrbracket$ is a well-formed execution if it begins from the empty configuration $C_{\text{init}} = (\{\}, \{\}, \{\})$ and $\forall \eta \in L(\tau)$, $\sigma_s(\eta) = \sigma_{\text{init}}$.*

Let $\mathcal{WF}(\mathcal{S}_{\mathcal{P},\Psi})$ denote all well-formed executions of $\mathcal{S}_{\mathcal{P},\Psi}$. The **consistency policy** $\Psi(\Delta, \text{vis}, \text{eo})$ is a formula constraining the events in Δ and relations vis and eo defined over these events. Below, we illustrate how to express certain well-known consistency policies in our framework:

Consistency Scheme	$\Psi(\Delta, \text{vis}, \text{eo})$
Eventual Consistency [3]	$\forall \eta, \eta' \in \Delta. \neg \text{eo}(\eta, \eta')$
Causal Consistency [14]	$\forall \eta, \eta' \in \Delta. \text{vis}(\eta, \eta') \Leftrightarrow \text{eo}(\eta, \eta')$ $\wedge \forall \eta, \eta', \eta'' \in \Delta. \text{vis}(\eta, \eta') \wedge \text{vis}(\eta', \eta'') \Rightarrow \text{vis}(\eta, \eta'')$
RedBlue Consistency (O_r) [13]	$\forall \eta, \eta' \in \Delta. o(\eta) \in O_r \wedge o(\eta') \in O_r \wedge \text{vis}(\eta, \eta') \Leftrightarrow \text{eo}(\eta, \eta')$ $\wedge \forall \eta, \eta' \in \Delta. o(\eta) \in O_r \wedge o(\eta') \in O_r \Rightarrow \text{vis}(\eta, \eta') \vee \text{vis}(\eta', \eta)$
Parallel Snapshot Isolation [22]	$\forall \eta, \eta' \in \Delta. (\text{Wr}(\eta^e) \cap \text{Wr}(\eta'^e) \neq \phi \wedge \text{vis}(\eta, \eta')) \Leftrightarrow \text{eo}(\eta, \eta')$ $\wedge \forall \eta, \eta' \in \Delta. \text{Wr}(\eta^e) \cap \text{Wr}(\eta'^e) \neq \phi \Rightarrow \text{vis}(\eta, \eta') \vee \text{vis}(\eta', \eta)$
Strong Consistency	$\forall \eta, \eta' \in \Delta. \text{vis}(\eta, \eta') \Leftrightarrow \text{eo}(\eta, \eta')$ $\wedge \forall \eta, \eta' \in \Delta. \text{vis}(\eta, \eta') \vee \text{vis}(\eta', \eta)$

For Eventual Consistency (EC) [3], we do not place any constraints on the visibility order and require the global effector order to be empty. This reflects the fact that in EC, any number of events can occur concurrently at different replicas, and hence a replica can witness any arbitrary subset of events which may be applied in any order. In Causal Consistency (CC) [14], an event is applied

at a replica only if all causally dependent events have already been applied. An event η_1 is causally dependent on η_2 if η_1 was generated at a replica where either η_2 or any other event causally dependent on η_2 had already been applied. The visibility relation vis captures causal dependency, and by making vis transitive, we ensure that all causal dependencies of events in Δ_r are also present in Δ_r (this is because in the transition rule, Ψ is checked on the updated visibility relation which relates events in Δ_r with the newly generated event). Further, causally dependent events must be applied in the same order at all replicas, which we capture by asserting that vis implies eo . In **RedBlue Consistency (RB)** [13], a subset of CRDT operations ($O_r \subseteq O$) are synchronized, so that they must occur in the same order at all replicas. We express RB in our framework by requiring the visibility relation to be total among events whose operations are in O_r . In **Parallel Snapshot Isolation (PSI)** [22], two events which conflict with each other (because they write to a common variable) are not allowed to be executed concurrently, but are synchronized across all replicas to be executed in the same order. Similar to [10], we assume that when a CRDT is used under PSI, its state space Σ is a map from variables to values, and every operation generates an effector which simply writes to certain variables. We assume that $\text{Wr}(\eta^e)$ returns the set of variables written by the effector η^e , and express PSI in our framework by requiring that events which write a common variable are applied in the same order (determined by their visibility relation) across all replicas; furthermore, the policy requires that the visibility operation among such events is total. Finally, in **Strong Consistency**, the visibility relation is total and all effectors are applied in the same order at all replicas.

Given an execution $\tau \in \llbracket \mathcal{S}_{\mathcal{P}, \Psi} \rrbracket$ and a transition $C \xrightarrow{\eta} C'$ in τ , we associate a set of replica states Σ_η that the event can potentially witness, by considering all permutations of the effectors visible to η which obey the global effector order, when applied to the start state $\sigma_s(\eta)$. Formally, this is defined as follows, assuming $\eta = (\text{eid}, o, \sigma_s, \{\eta_1, \dots, \eta_k\}, \text{eo}_r)$ and $C = (\Delta, \text{vis}, \text{eo})$:

$$\Sigma_\eta = \{ \eta_{P(1)}^e \circ \eta_{P(2)}^e \circ \dots \circ \eta_{P(k)}^e(\sigma_s) \mid P : \{1, \dots, k\} \rightarrow \{1, \dots, k\}, \\ \text{eo}_P \text{ is a total order, } i < j \Rightarrow (\eta_{P(j)}, \eta_{P(i)}) \in \text{eo}_P, \text{eo} \subseteq \text{eo}_P \}$$

In the above definition, for all valid local effector orders eo_P , we compute the CRDT states obtained on applying those effectors on the start CRDT state, which constitute Σ_η . The original event η presumably would have witnessed one of these states.

Definition 2 (Convergent Event). *Given an execution $\tau \in \llbracket \mathcal{S}_{\mathcal{P}, \Psi} \rrbracket$ and an event $\eta \in L(\tau)$, η is convergent if Σ_η is singleton.*

Definition 3 (Strong Eventual Consistency). *A CRDT $(\Sigma, O, \sigma_{\text{init}})$ achieves strong eventual consistency (SEC) under a weak consistency specification Ψ if for all well-formed executions $\tau \in \mathcal{WF}(\mathcal{S}_{\mathcal{P}, \Psi})$ and for all events $\eta \in L(\tau)$, η is convergent.*

An event is convergent if all valid permutations of visible events according to the specification Ψ lead to the same state. This corresponds to the requirement

that if two replicas have witnessed the same set of operations, they must be in the same state. A CRDT achieves SEC if all events in all executions are convergent.

4 Automated Verification

In order to show that a CRDT achieves SEC under a consistency specification, we need to show that all events in any well-formed execution are convergent, which in turn requires us to show that any valid permutation of valid subsets of events in an execution leads to the same state. This is a hard problem because we have to reason about executions of unbounded length, involving unbounded sets of effectors and reconcile the declarative event-based specifications of weak consistency with states generated during execution. To make the problem tractable, we use a two-fold strategy. First, we show that if any pair of effectors generated during any execution either commute with each other or are forced to be applied in the same order by the consistency policy, then the CRDT achieves SEC. Second, we develop an inductive proof rule to show that *all* pairs of effectors generated during any (potentially unbounded) execution obey the above mentioned property. To ensure soundness of the proof rule, we place some reasonable assumptions on the consistency policy that (intuitively) requires behaviorally equivalent events to be treated the same by the policy, regardless of context (i.e., the length of the execution history at the time the event is applied). We then extract a simple sufficient condition which we call as *non-interference to commutativity* that captures the heart of the inductive argument. Notably, this condition can be automatically checked for different CRDTs under different consistency policies using off-the-shelf theorem provers, thus providing a pathway to performing automated parametrized verification of CRDTs.

Given a transition $(\Delta, \text{vis}, \text{eo}) \xrightarrow{\eta} C$, we denote the global effector order in the starting configuration of η , i.e. eo as eo_η . We first show that a sufficient condition to prove that a CRDT is convergent is to show that any two events in its history either commute or are related by the global effector order.

Lemma 1. *Given an execution $\tau \in \llbracket \mathcal{S}_{\mathcal{P}, \Psi} \rrbracket$, and an event $\eta = (\text{id}, o, \sigma_s, \Delta_r, \text{eo}_\tau) \in L(\tau)$, if for all $\eta_1, \eta_2 \in \Delta_r$ such that $\eta_1 \neq \eta_2$, either $\eta_1^e \circ \eta_2^e = \eta_2^e \circ \eta_1^e$ or $\text{eo}_\eta(\eta_1, \eta_2)$ or $\text{eo}_\eta(\eta_2, \eta_1)$, then η is convergent³.*

We now present a property that consistency policies must obey for our verification methodology to be soundly applied. First, we define the notion of behavioral equivalence of events:

Definition 4 (Behavioral Equivalence).

Two events $\eta_1 = (\text{id}_1, o_1, \sigma_1, \Delta_1, \text{eo}_1)$ and $\eta_2 = (\text{id}_2, o_2, \sigma_2, \Delta_2, \text{eo}_2)$ are behaviorally equivalent if $\eta_1^e = \eta_2^e$ and $o_1 = o_2$.

That is, behaviorally equivalent events produce the same effectors. We use the notation $\eta_1 \equiv \eta_2$ to indicate that they are behaviorally equivalent.

³ All proofs can be found in the Appendix A

Definition 5 (Behaviorally Stable Consistency Policy). *A consistency policy Ψ is behaviorally stable if $\forall \Delta, \text{vis}, \text{eo}, \Delta', \text{vis}', \text{eo}', \eta_1, \eta_2 \in \Delta, \eta'_1, \eta'_2 \in \Delta'$ the following holds:*

$$\begin{aligned} (\Psi(\Delta, \text{vis}, \text{eo}) \wedge \Psi(\Delta', \text{vis}', \text{eo}') \wedge \eta_1 \equiv \eta'_1 \wedge \eta_2 \equiv \eta'_2 \wedge \text{vis}(\eta_1, \eta_2) \Leftrightarrow \text{vis}'(\eta'_1, \eta'_2)) \\ \Rightarrow \text{eo}(\eta_1, \eta_2) \Leftrightarrow \text{eo}'(\eta'_1, \eta'_2) \end{aligned} \quad (2)$$

Behaviorally stable consistency policies treat behaviorally equivalent events which have the same visibility relation among them in the same manner by enforcing the same effector order. All consistency policies that we discussed in the previous section (representing the most well-known in the literature) are behaviorally stable:

Lemma 2. *EC, CC, PSI, RB and SC are behaviorally stable.*

EC does not enforce any effector ordering and hence is trivially stable behaviorally. CC forces causally dependent events to be in the same order, and hence behaviorally equivalent events which have the same visibility order will be forced to be in the same effector order. RB forces events whose operations belong to a specific subset to be in the same order, but since behaviorally equivalent events perform the same operation, they would be enforced in the same effector ordering. Similarly, PSI forces events writing to a common variable to be in the same order, but since behaviorally equivalent events generate the same effector, they would also write to the same variables and hence would be forced in the same effector order. SC forces all events to be in the same order which is equal to the visibility order, and hence is trivially stable behaviorally. In general, behaviorally stable consistency policies do not consider the context in which events occur, but instead rely only on observable behavior of the events to constrain their ordering. A simple example of a consistency policy which is not behaviorally stable is a policy which maintains bounded concurrency [12] by limiting the number of concurrent operations across all replicas to a fixed bound. Such a policy would synchronize two events only if they occur in a context where keeping them concurrent would violate the bound, but behaviorally equivalent events in a different context may not be synchronized.

For executions under a behaviorally stable consistency policy, the global effector order between events only grows in an execution, so that if two events η_1 and η_2 are in the history of some event η are related by eo_η , then if they later occur in the history of any other event, they would be related in the same effector order. Hence, we can now define a common global effector order for an execution. Given an execution $\tau \in \llbracket \mathcal{S}_{\mathcal{P}, \Psi} \rrbracket$, the effector order $\text{eo}_\tau \subseteq L(\tau) \times L(\tau)$ is an anti-symmetric relation defined as follows:

$$\text{eo}_\tau = \{(\eta_1, \eta_2) \mid \exists \eta \in L(\tau). (\eta_1, \eta_2) \in \text{eo}_\eta\}$$

Similarly, we also define vis_τ to be the common visibility relation for an execution τ , which is nothing but the vis relation in the final configuration of τ .

Definition 6 (Commutative modulo Consistency Policy). *Given a CRDT \mathcal{P} , a behaviorally stable weak consistency specification Ψ and an execution $\tau \in \llbracket \mathcal{S}_{\mathcal{P}, \Psi} \rrbracket$, two events $\eta_1, \eta_2 \in L(\tau)$ such that $\eta_1 \neq \eta_2$ commute modulo the consistency policy Ψ if either $\eta_1^e \circ \eta_2^e = \eta_2^e \circ \eta_1^e$ or $\text{eo}_\tau(\eta_1, \eta_2)$ or $\text{eo}_\tau(\eta_2, \eta_1)$.*

The following lemma is a direct consequence of Lemma 1 :

Lemma 3. *Given a CRDT \mathcal{P} and a behaviorally stable consistency specification Ψ , if for all $\tau \in \mathcal{WF}(\mathcal{S}_{\mathcal{P}, \Psi})$, for all $\eta_1, \eta_2 \in L(\tau)$ such that $\eta_1 \neq \eta_2$, η_1 and η_2 commute modulo the consistency policy Ψ , then \mathcal{P} achieves SEC under Ψ .*

Our goal is to use Lemma 3 to show that all events in any execution commute modulo the consistency policy. However, executions can be arbitrarily long and have an unbounded number of events. Hence, for events occurring in such large executions, we will instead consider behaviorally equivalent events in a smaller execution and show that they commute modulo the consistency policy, which by stability of the consistency policy directly translates to their commutativity in the larger context. Recall that the effector generated by an operation depends on its start state and the sequence of other effectors applied to that state. To generate behaviorally equivalent events with arbitrarily long histories in short executions, we summarize these long histories into the start state of events, and use commutativity itself as an inductive property of these start states. That is, we ask if two events with arbitrary start states and empty histories commute modulo Ψ , whether the addition of another event to their histories would continue to allow them to commute modulo Ψ .

Definition 7 (Non-interference to Commutativity). (Non-Interf) *A CRDT $\mathcal{P} = (\Sigma, O, \sigma_{init})$ satisfies non-interference to commutativity under a consistency policy Ψ if and only if the following conditions hold:*

1. *For all executions $C_{init} \xrightarrow{\eta_1} C_1 \xrightarrow{\eta_2} C_2$ in $\mathcal{WF}(\mathcal{S}_{\mathcal{P}, \Psi})$, η_1 and η_2 commute modulo Ψ .*
2. *For all $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$, if for execution $\tau \equiv C_{init} \xrightarrow{\eta_1} C_1 \xrightarrow{\eta_2} C_2$ in $\llbracket \mathcal{S}_{\mathcal{P}, \Psi} \rrbracket$ where $\sigma_s(\eta_1) = \sigma_1$, $\sigma_s(\eta_2) = \sigma_2$, η_1 and η_2 commute modulo Ψ , then for all executions $\tau' \equiv C_{init} \xrightarrow{\eta_3} C'_1 \xrightarrow{\eta'_1} C'_2 \xrightarrow{\eta'_2} C'_3$ such that $\sigma_s(\eta'_1) = \sigma_1$, $o(\eta'_1) = o(\eta_1)$, $\sigma_s(\eta'_2) = \sigma_2$, $o(\eta'_2) = o(\eta_2)$, $\sigma_s(\eta_3) = \sigma_3$, and $\text{vis}_\tau(\eta_1, \eta_2) \Leftrightarrow \text{vis}_{\tau'}(\eta'_1, \eta'_2)$, η_1 and η_2 commute modulo Ψ .*

Condition (1) corresponds to the base case of our inductive argument and requires that in well-formed executions with 2 events, both the events commute modulo Ψ . For condition (2), our intention is to consider two events η_a and η_b with any arbitrary histories which can occur in any well-formed execution and, assuming that they commute modulo Ψ , show that even after the addition of another event to their histories, they continue to commute. We use CRDT states σ_1, σ_2 to summarize the histories of the two events, and construct behaviorally equivalent events ($\eta_1 \equiv \eta_a$ and $\eta_2 \equiv \eta_b$) which would take σ_1, σ_2 as

their start states. That is, if η_a produced the effector $o(\sigma_{\text{init}}, \pi)^4$, where o is the CRDT operation corresponding to η_a and π is the sequence of effectors in its history, we leverage the observation that $o(\sigma_{\text{init}}, \pi) = o(\pi(\sigma_{\text{init}}), \epsilon)$, and assuming $\sigma_1 = \pi(\sigma_{\text{init}})$, we obtain the behaviorally equivalent event η_1 , i.e. $\eta_1^e \equiv \eta_a^e$. Similar analysis establishes that $\eta_2^e \equiv \eta_b^e$. However, since we have no way of characterizing states σ_1 and σ_2 which are obtained by applying arbitrary sequences of effectors, we use commutativity itself as an identifying characteristic, focusing on only those σ_1 and σ_2 for which the events η_1 and η_2 commute modulo Ψ .

The interfering event is also summarized by another CRDT state σ_3 , and we require that after suffering interference from this new event, the original two events would continue to commute modulo Ψ . This would essentially establish that any two events with any history would commute modulo Ψ in these small executions, which by the behavioral stability of Ψ would translate to their commutativity in any execution.

Theorem 1. *Given a CRDT \mathcal{P} and a behaviorally stable consistency policy Ψ , if \mathcal{P} satisfies non-interference to commutativity under Ψ , then \mathcal{P} achieves SEC under Ψ .*

Example: Let us apply the proposed verification strategy to the ORSet CRDT shown in Fig 2. Under EC, condition (1) of Non-Interf fails, because in the execution $C_{\text{init}} \xrightarrow{\eta_1} C_1 \xrightarrow{\eta_2} C_2$ where $o(\eta_1) = \text{Add}(\mathbf{a}, \mathbf{i})$ and $o(\eta_2) = \text{Remove}(\mathbf{a})$ and $\text{vis}(\eta_1, \eta_2)$, η_1 and η_2 don't commute modulo EC, since (\mathbf{a}, \mathbf{i}) would be present in the source replica of $\text{Remove}(\mathbf{a})$. However, η_1 and η_2 would commute modulo CC, since they would be related by the effector order. Now, moving to condition (2) of Non-interf, we limit ourselves to source replica states σ_1 and σ_2 where $\text{Add}(\mathbf{a}, \mathbf{i})$ and $\text{Remove}(\mathbf{a})$ do commute modulo CC. If $\text{vis}_\tau(\eta_1, \eta_2)$, then after interference, in execution τ' , $\text{vis}_{\tau'}(\eta'_1, \eta'_2)$, in which case η'_1 and η'_2 trivially commute modulo CC (because they would be related by the effector order). On the other hand, if $\neg \text{vis}_\tau(\eta_1, \eta_2)$, then for η_1 and η_2 to commute modulo CC, we must have that the effectors η_1^e and η_2^e themselves commute, which implies that $(\mathbf{a}, \mathbf{i}) \notin \sigma_2$. Now, consider any execution τ' with an interfering operation η_3 . If η_3 is another $\text{Add}(\mathbf{a}, \mathbf{i}')$ operation, then $\mathbf{i}' \neq \mathbf{i}$, so that even if it is visible to η_2 , η_2^e will not remove (\mathbf{a}, \mathbf{i}) , so that η'_1 and η'_2 would commute. Similarly, if η_3 is another $\text{Remove}(\mathbf{a})$ operation, it can only remove tagged versions of \mathbf{a} from the source replicas of η'_2 , so that the effector η_2^e would not remove (\mathbf{a}, \mathbf{i}) .

5 Experimental Results

In this section, we present the results of applying our verification methodology to a number of CRDTs under different consistency models. We collected CRDT implementations from a number of sources [19,1,18] and since all of the existing implementations assume a very weak consistency model (primarily CC), we additionally implemented a few CRDTs on our own intended to only work under

⁴ Note that in a well-formed execution, the start state is always σ_{init}

stronger consistency schemes but which are better in terms of time/space complexity and ease of development. Our implementations are not written in any specific language but instead are specified abstractly akin to the definitions given in Figs. 1 and 2. To specify CRDT states and operations, we fix an abstract language that contains uninterpreted datatypes (used for specifying elements of sets, lists, etc.), a set datatype with support for various set operations (add, delete, union, intersection, projection, lookup), a tuple datatype (along with operations to create tuples and project components) and a special uninterpreted datatype equipped with a total order for identifiers. Note that the set datatype used in our abstract language is different from the **Set** CRDT, as it is only intended to perform set operations locally at a replica. Our data types are uninterpreted since our verification technique generates verification conditions in first-order logic (FOL) with universal quantifiers. All existing CRDT definitions can be naturally expressed in this framework and can be found in Appendix B.

Here, we revert back to the op-based specification of CRDTs. For a given CRDT $\mathcal{P} = (\Sigma, O, \sigma_{\text{init}})$, we convert all its operations into FOL formulas relating the source, input and output replica states. That is, for a CRDT operation $o : \Sigma \rightarrow \Sigma \rightarrow \Sigma$, we create a predicate $o : \Sigma \times \Sigma \times \Sigma \rightarrow \mathbb{B}$ such that $o(\sigma_s, \sigma_i, \sigma_o)$ is true if and only if $o(\sigma_s)(\sigma_i) = \sigma_o$. Since CRDT states are typically expressed as sets, we axiomatize set operations to express their semantics in FOL.

In order to specify a consistency model, we introduce a sort for events and binary predicates vis and eo over this sort. Here, we can take advantage of the declarative specification of consistency models and directly encode them in FOL. Given an encoding of CRDT operations and a consistency model, our verification strategy is to determine whether the **Non-Interf** property holds. Since both conditions of this property only involve executions of finite length (at most 3), we can directly encode them as UNSAT queries by asking for executions which break the conditions. For condition (1), we query for the existence of two events η_1 and η_2 along with vis and eo predicates which satisfy the consistency specification Ψ such that these events are not related by eo and their effectors do not commute. For condition (2), we query for the existence of events η_1, η_2, η_3 and their respective start states $\sigma_1, \sigma_2, \sigma_3$, such that η_1 and η_2 commute modulo Ψ but after interference from η_3 , they are not related by eo and do not commute. Both these queries are encoded in EPR [17], a decidable fragment of FOL, so if the CRDT operations and the consistency policy can also be encoded in a decidable fragment of FOL (which is the case in all our experiments), then our verification strategy is also decidable. We write **Non-Interf-1** and **Non-Interf-2** for the two conditions of **Non-Interf**.

Fig. 4 shows the results of applying the proposed methodology on different CRDTs. We used Z3 to discharge our satisfiability queries. For every combination of a CRDT and a consistency policy, we write \times to indicate that verification of **Non-Interf** failed, while \checkmark indicates that it was satisfied. We also report the verification time taken by Z3 for every CRDT across all consistency policies executing on a standard desktop machine. We have picked the three collection datatypes for which CRDTs have been proposed i.e. **Set**, **List** and **Graph**, and

CRDT	EC	CC	PSI+RB	PSI	Verif. Time (s)
Set					
Simple-Set	✗	✗	✓	✓	0.23
ORSet [19]	✗	✓	✓	✓	0.6
ORSet with Tombstones	✓	✓	✓	✓	0.04
USet[19]	✗	✗	✗	✓	0.1
List					
RGA[1]	✗	✓	✓	✓	5.3
RGA-No-Tomb	✗	✗	✓	✓	3
Graph					
2P2P-Graph[19]	✗	✓	✓	✓	3.5
Graph-with-ORSet	✗	✗	✓	✓	46.3

Fig. 4: Convergence of CRDTs under different consistency policies.

for each such datatype, we consider multiple variants that provide a tradeoff between consistency requirements and implementation complexity. Apart from EC, CC and PSI, we also use a combination of PSI and RB, which only enforce PSI between selected pairs of operations (in contrast to simple RB which would enforce SC between all selected pairs). Note that when verifying a CRDT under PSI, we assume that the set operations are implemented as Boolean assignments, and the write set Wr consists of elements added/removed. We are unaware of any prior effort that has been successful in automatically verifying *any* CRDT, let alone those that exhibit the complexity of the ones considered here.

Set: The Simple-Set CRDT in Fig 1 does not converge under EC or CC, but achieves convergence under PSI+RB which only synchronizes **Add** and **Remove** operations to the same elements, while all other operations continue to run under EC, since they do commute with each other. As explained earlier, ORSet does not converge under EC and violates Non-Interf-1. ORSet with tombstones converges under EC as well since it uses a different set (called a tombstone) to keep track of removed elements. USet is another implementation of the Set CRDT which converges under the assumptions that an element is only added once, and removes only work if the element is already present in the source replica. USet converges only under PSI, because under any weaker consistency model, NON-INTERF-2 breaks, since **Add(a)** interferes and breaks the commutativity of **Add(a)** and **Remove(a)**. Notice that as the consistency level weakens, implementations need to keep more and more information to maintain convergence—compute unique ids, tag elements with them or keep track of deleted elements. If the underlying replicated store supports stronger consistency levels such as PSI, simpler definitions are sufficient.

List: The List CRDT maintains a total ordering between its elements. It supports two operations : **AddRight(e, a)** adds new element **a** to the right of existing element **e**, while **Remove(e)** removes **e** from the list. We use the im-

plementation in [1] (called RGA) which uses time-stamped insertion trees. To maintain integrity of the tree structure, the immediate predecessor of every list element must be present in the list, due to which operations `AddRight(a,b)` and `AddRight(b,c)` do not commute. Hence RGA does not converge under EC because `Non-Interf-1` is violated, but converges under CC.

To make adds and removes involving the same list element commute, RGA maintains a tombstone set for all deleted list elements. This can be expensive as deleted elements may potentially need to be tracked forever, even with garbage collection. We consider a slight modification of RGA called RGA-No-Tomb which does not keep track of deleted elements. This CRDT now has a convergence violation under CC (because of `Non-Interf-1`), but achieves convergence under PSI+RB where we enforce PSI only for pairs of `AddRight` and `Remove` operations.

Graph: The Graph CRDT maintains sets of vertices and edges and supports operations to add and remove vertices and edges. The 2P2P-Graph specification uses separate 2P-Sets for both vertices and edges, where a 2P-Set itself maintains two sets for addition and removal of elements. While 2P sets themselves converge under EC, the 2P2P-Graph has convergence violations (to `Non-Interf-1`) involving `AddVertex(v)` and `RemoveVertex(v)` (similarly for edges) since it removes a vertex from a replica only if it is already present. We verify that it converges under CC. Graphs require an integrity constraint that edges in the edge-set must always be incident on vertices in the vertex-set. Since concurrent `RemoveVertex(v)` and `AddEdge(v,v')` can violate this constraint, the 2P2P-Graph uses the internal structure of the 2P-Set which keeps track of deleted elements and considers an edge to be in the edge set only if its vertices are not in the vertex tombstone set (leading to a remove-wins strategy).

Building a graph CRDT can be viewed as an exercise in composing CRDTs by using two ORSet CRDTs, keeping the internal implementation of the ORSet opaque, using only its interface. The Graph-with-ORSet implementation uses separate ORSets for vertices and edges and explicitly maintains the graph integrity constraint. We find convergence violations (to `Non-Interf-1`) between `RemoveVertex(v)` and `AddEdge(v,v')`, and `RemoveVertex(v)` and `RemoveEdge(v,v')` under both EC and CC. Under PSI+RB (enforcing RB on the above two pairs of operations), we were able to show convergence.

When a CRDT passes `Non-Interf` under a consistency policy, we can guarantee that it achieves SEC under that policy. However, if it fails `Non-Interf`, it may or may not converge. In particular, if it fails `Non-Interf-1` it will definitely not converge (because `Non-Interf-1` constructs a well-formed execution), but if it passes `Non-Interf-1` and fails `Non-Interf-2`, it may still converge because of the imprecision of `Non-Interf-2`. There are two sources of imprecision, both concerning the start states of the events picked in the condition: (1) we only use commutativity as a distinguishing property of the start states, but this may not be a sufficiently strong inductive invariant, (2) we place no constraints on the start state of the interfering operation. In practice, we have found that for all cases except U-Set, convergence violations manifest via failure of `Non-Interf-1`. If `Non-Interf-2` breaks, we can search for well-formed executions of higher length upto a

bound. For U-Set, we were successful in adopting this approach, and were able to find a non-convergent well-formed execution of length 3.

6 Related Work and Conclusions

Reconciling concurrent updates in a replicated system is an important well-studied problem in distributed applications, having been first studied in the context of collaborative editing systems [16]. Incorrect implementation of replicated sets in Amazon’s Dynamo system [7] motivated the design of CRDTs as a principled approach to implementing replicated data types. Devising correct implementations has proven to be challenging, however, as evidenced by the myriad pre-conditions specified in the various CRDT implementations [19].

Burckhardt *et al.* [6] present an abstract event-based framework to describe executions of CRDTs under different network conditions; they also propose a rigorous correctness criterion in the form of abstract specifications. Their proof strategy, which is neither automated nor parametric on consistency policies, verifies CRDT implementations against these specifications by providing a simulation invariant between CRDT states and event structures. Zeller *et al.* [23] also require simulation invariants to verify convergence, although they only target state-based CRDTs. Gomes *et al.* [9] provide mechanized proofs of convergence for ORSet and RGA CRDTs under causal consistency, but their approach is neither automated nor parametric.

A number of earlier efforts [2,21,10,12,11] have looked at the problem of verifying state-based invariants in distributed applications. These techniques typically target applications built using CRDTs, and assume their underlying correctness. Because they target correctness specifications in the form of state-based invariants, it is unclear if their approaches can be applied directly to the convergence problem we consider here. Other approaches [4,5,15] have also looked at the verification problem of transactional programs running on replicated systems under weak consistency, but these proposals typically use serializability as the correctness criterion, adopting a “last-writer wins” semantics, rather than convergence, to deal with concurrent updates.

This paper demonstrates the automated verification of CRDTs under different weak consistency policies. We rigorously define the relationship between commutativity and convergence, formulating the notion of commutativity modulo consistency policy as a sufficient condition for convergence. While we require a non-trivial inductive argument to show that non-interference to commutativity is sufficient for convergence, the condition itself is designed to be simple and amenable to automated verification using off-the-shelf theorem-provers. We have successfully applied the proposed verification strategy for all major CRDTs, additionally motivating the need for parametrization in consistency policies by showing variants of existing CRDTs which are simpler in terms of implementation complexity but converge under different weak consistency models.

Acknowledgments. We thank the anonymous reviewers for their insightful comments. This material is based upon work supported by the National Science

Foundation under Grant No. CCF-SHF 1717741 and the Air Force Research Lab under Grant No. FA8750-17-1-0006.

References

1. Attiya, H., Burckhardt, S., Gotsman, A., Morrison, A., Yang, H., Zawirski, M.: Specification and complexity of collaborative text editing. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25–28, 2016. pp. 259–268 (2016). <https://doi.org/10.1145/2933057.2933090>, <https://doi.org/10.1145/2933057.2933090>
2. Bailis, P., Fekete, A., Franklin, M.J., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Coordination avoidance in database systems. *PVLDB* **8**(3), 185–196 (2014). <https://doi.org/10.14778/2735508.2735509>, <http://www.vldb.org/pvldb/vol8/p185-bailis.pdf>
3. Bailis, P., Ghodsi, A.: Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM* **56**(5), 55–63 (2013). <https://doi.org/10.1145/2447976.2447992>, <https://doi.org/10.1145/2447976.2447992>
4. Bernardi, G., Gotsman, A.: Robustness against consistency models with atomic visibility. In: 27th International Conference on Concurrency Theory, CONCUR 2016, August 23–26, 2016, Québec City, Canada. pp. 7:1–7:15 (2016). <https://doi.org/10.4230/LIPIcs.CONCUR.2016.7>, <https://doi.org/10.4230/LIPIcs.CONCUR.2016.7>
5. Brutschy, L., Dimitrov, D., Müller, P., Vechev, M.T.: Static serializability analysis for causal consistency. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18–22, 2018. pp. 90–104 (2018). <https://doi.org/10.1145/3192366.3192415>, <https://doi.org/10.1145/3192366.3192415>
6. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20–21, 2014. pp. 271–284 (2014). <https://doi.org/10.1145/2535838.2535848>, <https://doi.org/10.1145/2535838.2535848>
7. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Voshall, P., Vogels, W.: Dynamo: amazon’s highly available key-value store. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14–17, 2007. pp. 205–220 (2007). <https://doi.org/10.1145/1294261.1294281>, <https://doi.org/10.1145/1294261.1294281>
8. Gilbert, S., Lynch, N.A.: Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* **33**(2), 51–59 (2002). <https://doi.org/10.1145/564585.564601>, <http://doi.acm.org/10.1145/564585.564601>
9. Gomes, V.B.F., Kleppmann, M., Mulligan, D.P., Beresford, A.R.: Verifying strong eventual consistency in distributed systems. *PACMPL* **1**(OOPSLA), 109:1–109:28 (2017). <https://doi.org/10.1145/3133933>, <https://doi.org/10.1145/3133933>

10. Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 371–384 (2016). <https://doi.org/10.1145/2837614.2837625>, <http://doi.acm.org/10.1145/2837614.2837625>
11. Houshmand, F., Lesani, M.: Hamsaz: replication coordination analysis and synthesis. PACMPL **3**(POPL), 74:1–74:32 (2019), <https://dl.acm.org/citation.cfm?id=3290387>
12. Kaki, G., Earanky, K., Sivaramakrishnan, K.C., Jagannathan, S.: Safe replication through bounded concurrency verification. PACMPL **2**(OOPSLA), 164:1–164:27 (2018). <https://doi.org/10.1145/3276534>, <https://doi.org/10.1145/3276534>
13. Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N.M., Rodrigues, R.: Making geo-replicated systems fast as possible, consistent when necessary. In: 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012. pp. 265–278 (2012), <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li>
14. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011. pp. 401–416 (2011). <https://doi.org/10.1145/2043556.2043593>, <http://doi.acm.org/10.1145/2043556.2043593>
15. Nagar, K., Jagannathan, S.: Automated detection of serializability violations under weak consistency. In: 29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China. pp. 41:1–41:18 (2018). <https://doi.org/10.4230/LIPIcs.CONCUR.2018.41>, <https://doi.org/10.4230/LIPIcs.CONCUR.2018.41>
16. Nichols, D.A., Curtis, P., Dixon, M., Lamping, J.: High-latency, low-bandwidth windowing in the jupiter collaboration system. In: Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology, UIST 1995, Pittsburgh, PA, USA, November 14-17, 1995. pp. 111–120 (1995). <https://doi.org/10.1145/215585.215706>, <https://doi.org/10.1145/215585.215706>
17. Piskac, R., de Moura, L.M., Bjørner, N.: Deciding effectively propositional logic using DPLL and substitution sets. J. Autom. Reasoning **44**(4), 401–424 (2010). <https://doi.org/10.1007/s10817-009-9161-6>, <https://doi.org/10.1007/s10817-009-9161-6>
18. Preguiça, N.M., Baquero, C., Shapiro, M.: Conflict-free replicated data types (crdts). CoRR **abs/1805.06358** (2018), <http://arxiv.org/abs/1805.06358>
19. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Tech. Rep. RR-7506, INRIA, Inria Centre Paris-Rocquencourt (2011)
20. Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings. pp. 386–400 (2011). https://doi.org/10.1007/978-3-642-24550-3_29, https://doi.org/10.1007/978-3-642-24550-3_29
21. Sivaramakrishnan, K.C., Kaki, G., Jagannathan, S.: Declarative programming over eventually consistent data stores. In: Proceedings

- of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 413–424 (2015). <https://doi.org/10.1145/2737924.2737981>, <https://doi.org/10.1145/2737924.2737981>
22. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSOP 2011, Cascais, Portugal, October 23-26, 2011. pp. 385–400 (2011). <https://doi.org/10.1145/2043556.2043592>, <http://doi.acm.org/10.1145/2043556.2043592>
 23. Zeller, P., Bieniusa, A., Poetzsch-Heffter, A.: Formal specification and verification of crdts. In: Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings. pp. 33–48 (2014). https://doi.org/10.1007/978-3-662-43613-4_3, https://doi.org/10.1007/978-3-662-43613-4_3

A Proofs

Lemma 1. Given an execution $\tau \in \llbracket \mathcal{S}_{\mathcal{P}, \Psi} \rrbracket$, and an event $\eta = (\text{id}, o, \sigma_s, \Delta_r, \text{eo}_r) \in L(\tau)$, if for all $\eta_1, \eta_2 \in \Delta_r$ such that $\eta_1 \neq \eta_2$, either $\eta_1^e \circ \eta_2^e = \eta_2^e \circ \eta_1^e$ or $\text{eo}_\eta(\eta_1, \eta_2)$ or $\text{eo}_\eta(\eta_2, \eta_1)$, then η is convergent.

Proof. Let $\Delta_r = \{\eta_1, \dots, \eta_n\}$. To show that η is convergent, we need to show that any two permutations P_1 and P_2 of effectors in Δ_r obeying the global effector order eo_η lead to the same state when applied on the initial state σ_{init} . We will show this using induction on n (i.e. size of Δ_r). If $n = 1$, then there is only one effector and hence only one permutation. Assume that the result holds for $k - 1$. Now, let $n = k$ and consider two permutations P_1 and P_2 . We need to show that

$$\eta_{P_1(1)}^e \circ \dots \circ \eta_{P_1(k)}^e(\sigma_{\text{init}}) = \eta_{P_2(1)}^e \circ \dots \circ \eta_{P_2(k)}^e(\sigma_{\text{init}})$$

If $P_1(1) = P_2(1)$, then the result follows because by the inductive hypothesis, which states that if the number of events is $k - 1$ then all permutations lead to the same state, and hence

$$\eta_{P_1(2)}^e \circ \dots \circ \eta_{P_1(k)}^e(\sigma_{\text{init}}) = \eta_{P_2(2)}^e \circ \dots \circ \eta_{P_2(k)}^e(\sigma_{\text{init}})$$

If $P_1(1) \neq P_2(1)$, let j be such that $P_1(1) = P_2(j)$. Now, for all $1 \leq k < j$, neither $\text{eo}_\eta(\eta_{P_1(1)}, \eta_{P_2(k)})$ nor $\text{eo}_\eta(\eta_{P_2(k)}, \eta_{P_1(1)})$ because $\eta_{P_2(k)}$ appears before $\eta_{P_1(1)}$ in P_2 and $\eta_{P_1(1)}$ appears before $\eta_{P_2(k)}$ in P_1 . Hence, we must have

$$\eta_{P_1(1)}^e \circ \eta_{P_2(k)}^e = \eta_{P_2(k)}^e \circ \eta_{P_1(1)}^e$$

Hence, we now interchange $\eta_{P_2(j)}^e$ and $\eta_{P_2(k)}^e$ without changing the overall function obtained from the composition, bringing $\eta_{P_2(j)}^e$ at the front of P_2 . Now, the result directly follows as above from the inductive hypothesis. \square

Lemma 2. EC, CC, PSI, RB and SC are behaviorally stable.

Proof. Directly follows from the definition of behavioral stability and definition of the consistency policies. \square

Lemma 3. Given a CRDT \mathcal{P} and a behaviorally stable consistency specification Ψ , if for all $\tau \in \mathcal{WF}(\mathcal{S}_{\mathcal{P}, \Psi})$, for all $\eta_1, \eta_2 \in L(\tau)$ such that $\eta_1 \neq \eta_2$, η_1 and η_2 commute modulo the consistency policy Ψ , then \mathcal{P} achieves SEC under Ψ .

Proof. Given an execution $\tau \in \mathcal{WF}(\mathcal{S}_{\mathcal{P}, \Psi})$, and an event $\eta \in L(\tau)$, let $\eta = (\text{id}, o, \sigma_{\text{init}}, \Delta_r, \text{eo}_r)$. Consider $\eta_1, \eta_2 \in \Delta_r$. Since $\eta_1, \eta_2 \in L(\tau)$, they commute modulo the consistency policy. By behavioral stability of Ψ , we have

$$\text{eo}_\tau(\eta_1, \eta_2) \Rightarrow \text{eo}_\eta(\eta_1, \eta_2)$$

This implies that either $\eta_1^e \circ \eta_2^e$ or $\text{eo}_\eta(\eta_1, \eta_2)$ or $\text{eo}_\eta(\eta_2, \eta_1)$. By Lemma 1, we have the fact the η is convergent. Hence, by definition of SEC, the result directly follows. \square

Theorem 1. Given a CRDT \mathcal{P} and a behaviorally stable consistency policy Ψ , if \mathcal{P} satisfies non-interference to commutativity under Ψ , then \mathcal{P} achieves SEC under Ψ .

Proof. To prove this, we will show that if \mathcal{P} satisfies non-interference to commutativity under Ψ , then any two events in any well-formed execution commute modulo Ψ , which by Lemma 3 shows that \mathcal{P} achieves SEC under Ψ . We use induction on the length of execution (note that length of an execution is the number of events it contains).

Base Case: For execution of length 1, it is trivially true since there is only one event.

Inductive Case: Suppose the statement holds for all executions of length less than or equal to k . Consider an execution τ of length $k + 1$.

$$\tau = C_{\text{init}} \rightarrow \dots \rightarrow C'' \xrightarrow{\eta'} C' \xrightarrow{\eta} C$$

In the above execution, η is the $(k + 1)$ th event, and η' is its predecessor event. By the inductive hypothesis, all events in τ before η commute with each other modulo Ψ , and hence we only need to show that η commutes modulo Ψ with all other events in τ . Let $\eta = (\text{id}, o, \sigma_{\text{init}}, \Delta_r, \text{eo})$. There are two cases based on whether η' is present in Δ_r :

Case-1 : $\eta' \notin \Delta_r$. In this case, we can interchange the order of the two events, since the event η can also occur from configuration C'' . By doing so, we get an execution of length k containing η and by the inductive hypothesis, η would commute with all other events in this execution modulo Ψ . Hence, we only need to show now that η' commutes with η modulo Ψ .

Let $\eta' = (\text{id}', o', \sigma_{\text{init}}, \Delta_r', \text{eo}')$. If $\Delta_r \cup \Delta_r'$ is empty, then condition (1) in the NON-INTERF directly applies, since both these events can directly occur from the initial empty configuration C_{init} . Hence, η and η' would commute modulo Ψ .

Now suppose that $\Delta_r \cup \Delta'_r$ contains at least one event. Let η'' be the latest event in the execution τ occurring in $\Delta_r \cup \Delta'_r$. We can construct an execution τ' which does not contain η'' which would end in events γ' and γ which are analogous to η' and η except that they don't contain η'' in their history. Note that every other event which contains η'' in its history and other events which contain these removed events in their history would also have to be removed to get τ' . However, since η'' is the latest event in the histories of η and η' , every other event in their history would still be present in τ' . Hence, $\gamma = (\text{id}, o, \sigma_{\text{init}}, \Delta_r \setminus \{\eta''\}, \text{eo}_r)$ and $\gamma' = (\text{id}', o', \sigma_{\text{init}}, \Delta'_r \setminus \{\eta''\}, \text{eo}'_r)$, where eo_r and eo'_r are same as the original local effector orders eo and eo' but applied on the new histories.

Since the length of τ' is less than or equal to k (because at least one event is guaranteed to be removed), by the inductive hypothesis γ' and γ commute modulo Ψ . Now, we will instantiate condition (2) of NON-INTERF with η_1 being analogous to γ' and η_2 analogous to γ .

Specifically, let $\sigma_1 = \pi'(\sigma_{\text{init}})$ where π' is the sequence of effectors in $\Delta'_r \setminus \{\eta''\}$ obeying the total order eo'_r . Let $\eta_1 = (\text{id}', o_1, \sigma_1, \{\}, \{\})$. Then,

$$\begin{aligned} \gamma'^e &= o'(\sigma_{\text{init}}, \pi') \\ &= o'(\pi'(\sigma_{\text{init}}), \epsilon) \\ &= o'(\sigma_1, \epsilon) \\ &= \eta_1^e \end{aligned}$$

In the above derivation, we first use the property that for any CRDT operation o , state σ and function f , $o(\sigma, f) = o(f(\sigma), \epsilon)$.

Similarly, let $\sigma_2 = \pi(\sigma_{\text{init}})$ where π is the sequence of effectors in $\Delta_r \setminus \{\eta''\}$ obeying the total order eo_r . Let $\eta_2 = (\text{id}, o, \sigma_2, \{\}, \{\})$. Similar to above, we can derive that $\gamma^e = \eta_2^e$. Since both γ' and η_1 , γ and η_2 have the same effectors and the same visibility relation, by behavioral stability of Ψ , in the execution $C_{\text{init}} \xrightarrow{\eta_1} C_1 \xrightarrow{\eta_2} C_2$, η_1 and η_2 commute modulo Ψ .

The interfering operation η_3 is constructed to be analogous to η'' . That is, $\eta_3^e = \eta''^e$. Then, by NON-INTERF, for all executions $C_{\text{init}} \xrightarrow{\eta_3} C'_1 \xrightarrow{\eta'_1} C'_2 \xrightarrow{\eta'_2} C'_3$ such that $\sigma_s(\eta_1) = \sigma_1$, $\sigma_s(\eta_2) = \sigma_2$ and $\text{vis}_\tau(\eta_1, \eta_2) \Leftrightarrow \text{vis}_{\tau'}(\eta_1, \eta_2)$, η_1 and η_2 commute modulo Ψ . We pick one such execution depending on whether η'' was visible to η and/or η' .

Let us consider the case where η'' was visible to η' , so that $\eta'' \in \Delta'_r$. Then, $\eta'_1 = (\text{id}', o, \sigma_1, \{\eta_3\}, \{\})$. We will now show that $\eta'_1{}^e = \eta'^e$:

$$\begin{aligned} \eta'_1{}^e &= o'(\sigma_1, \eta_3^e) \\ &= o'(\sigma_1, \eta''^e) \\ &= o'(\pi'(\sigma_{\text{init}}, \eta''^e)) \\ &= o'(\sigma_{\text{init}}, \pi' \circ \eta''^e) \\ &= \eta'^e \end{aligned}$$

Note that for the last step, we use the fact that η'' commutes with all events in Δ'_r modulo Ψ by the inductive hypothesis. Since η'' is the latest occurring event in τ , it cannot be related by \mathbf{eo}' before any other event in Δ'_r . Hence, η'' is either the last event in the total order \mathbf{eo}' (in which case the above derivation directly follows) or is not the last event, but commutes with every other event after it in τ' , due to which we get the same result.

Similarly, depending on whether $\eta'' \in \Delta_r$, we choose an appropriate η'_2 such that $\eta_2^e = \eta^e$. Now, since $\eta \equiv \eta'_2$ and $\eta' \equiv \eta'_1$ and η'_1 and η'_2 commute modulo Ψ , by behavioral stability of Ψ , η and η' also commute modulo Ψ .

Case-2 : $\eta' \in \Delta_r$, let us first consider any other event η'' in τ . Here, we repeat the above procedure to prove the commutativity of the two events η and η'' by considering η' as the interfering event.

We construct an execution τ' which is the same as τ except that it does not contain η' . This execution will end in the event γ which is analogous to η except that it does not contain η' in its history. Hence, $\gamma = (\text{id}, o, \sigma_{\text{init}}, \Delta_r \setminus \{\eta'\}, \mathbf{eo}_r)$, where \mathbf{eo}_r is \mathbf{eo} restricted to $\Delta_r \setminus \{\eta'\}$. The event η'' will also be present in τ' . Since the length of τ' is k , by the inductive hypothesis, γ and η'' commute modulo Ψ . Now, we will instantiate NON-INTERF-2 with $\eta_1 \equiv \eta''$ and $\eta_2 \equiv \gamma$.

Specifically, suppose that $\eta'' = (_, o'', \sigma_{\text{init}}, \Delta'', \mathbf{eo}'')$. Then $\eta_1 = (_, o'', \sigma_1, \{\}, \{\})$, where $\sigma_1 = \pi''(\sigma_{\text{init}})$, where π'' is the sequence of effectors of Δ'' in the order \mathbf{eo}'' . Similarly, $\eta_2 = (_, o, \sigma_2, \{\}, \{\})$, where $\sigma_2 = \pi(\sigma_{\text{init}})$ where π is the sequence of effectors of $\Delta_r \setminus \{\eta'\}$ in the order \mathbf{eo}_r . Note that here we assume that $\eta'' \notin \Delta_r$. The other case where $\eta'' \in \Delta_r$ can be similarly handled by placing η_1 in the history of η_2 . Now, by the behavioral stability of Ψ , in the execution, $C_{\text{init}} \xrightarrow{\eta_1} C_1 \xrightarrow{\eta_2} C_2$, η_1 and η_2 commute modulo Ψ .

We let $\eta_3 \equiv \eta'$ be the interfering event. Then, in the execution $C_{\text{init}} \xrightarrow{\eta_3} D_1 \xrightarrow{\eta_1} D_2 \xrightarrow{\eta_2} D_3$ where $\eta_2 = (_, o, \sigma_2, \{\eta_3\}, \{\})$, by NON-INTERF-2, η_1 and η_2 commute modulo Ψ . However, $\eta_2 \equiv \eta$ and hence by the behavioral stability of Ψ , η'' and η commute modulo Ψ .

Finally, let us now prove the commutativity of η and η' , where we will again repeat the above argument by picking the latest event in the histories of η and η' as the interfering event.

Let $\eta' = (_, o', \sigma_{\text{init}}, \Delta'_r, \mathbf{eo}')$. First, if $\Delta_r \cup \Delta'_r = \{\eta'\}$, then we can directly apply NON-INTERF-1 since both these events can directly occur from the initial empty configuration, and hence they would commute modulo Ψ .

Otherwise, let η'' be the latest event in $(\Delta_r \cup \Delta'_r) \setminus \{\eta'\}$. We construct an execution τ' which is the same as τ except it does not contain η'' and ends in events γ' and γ analogous to η' and η except that they don't contain η'' in their history. Note that every other event which contains η'' in its history also needs to be removed (and events containing these removed events and so on). However, since η'' is the latest event in the combined histories of η and η' , no other event in their history will be removed.

Hence, $\gamma = (\text{id}, o, \sigma_{\text{init}}, (\Delta_r \cup \{\gamma'\}) \setminus \{\eta'', \eta'\}, \mathbf{eo}_r)$ and $\gamma' = (_, o', \sigma_{\text{init}}, \Delta'_r \setminus \{\eta''\}, \mathbf{eo}')$. Now, the length of τ' is less than or equal to k , and hence by the

inductive hypothesis, γ and γ' commute modulo Ψ . Now, we will instantiate NON-INTERF-2 with $\eta_1 \equiv \gamma'$ and $\eta_2 \equiv \gamma$.

Specifically, $\eta_1 = (_, o', \pi'(\sigma_{\text{init}}), \{\}, \{\})$ where π' is the sequence of effectors in $\Delta_r \setminus \{\eta''\}$ in the order eo . $\eta_2 = (_, o, \pi(\sigma_{\text{init}}), \{\eta_1\}, \{\})$, where π is the sequence of effectors in $\Delta_r \setminus \{\eta'', \eta'\}$. Then, by the behavioral stability of Ψ , in the execution $C_{\text{init}} \xrightarrow{\eta_1} C_1 \xrightarrow{\eta_2} C_2$, η_1 and η_2 commute modulo Ψ .

We let $\eta_3 \equiv \eta''$ be the interfering event. Then, in the execution $C_{\text{init}} \xrightarrow{\eta_3} D_1 \xrightarrow{\eta'_1} D_2 \xrightarrow{\eta'_2} D_3$ where $\eta'_1 = (_, o', \pi'(\sigma_{\text{init}}, \{\eta_3\}, \{\}))$ and $\eta'_2 = (_, o, \pi(\sigma_{\text{init}}, \{\eta_3, \eta'_1\}, \{\eta_3, \eta'_1\}))$, by NON-INTERF-2, η_1 and η_2 commute modulo Ψ . However, $\eta_1 \equiv \eta'$ and $\eta_2 \equiv \eta$. Hence, by the behavioral stability of Ψ , η' and η commute modulo Ψ . \square

B CRDT Definitions

Simple Set:

$$\begin{aligned} S &\in \mathbb{IP}(E) \\ \text{Add}(a) : S, & \quad \lambda S'. S' \cup \{a\} \\ \text{Remove}(a) : S, & \quad \lambda S'. S' \setminus \{a\} \\ \text{Lookup}(a) : S, & \quad a \in S \end{aligned}$$

ORSet:

$$\begin{aligned} S &\in \mathbb{IP}(E \times I) \\ \text{Add}(a, i) : S & \quad \lambda S'. S' \cup \{(a, i)\} \\ \text{Remove}(a) : S & \quad \lambda S'. S' \setminus \{(a, i) : (a, i) \in S\} \\ \text{Lookup}(a) : S & \quad \exists (a, i) \in A \end{aligned}$$

ORSet with Tombstones:

$$\begin{aligned} S &\in \mathbb{IP}(E \times I) \times \mathbb{IP}(E \times I) \\ \text{Add}(a, i, (A, R)) & \quad \lambda (A', R'). (A' \cup \{(a, i)\}, R') \\ \text{Remove}(a, (A, R)) & \quad \lambda (A', R'). (A', R' \cup \{(a, i) : (a, i) \in A\}) \\ \text{Lookup}(a, (A, R)) & \quad \exists (a, i) \in A \wedge (a, i) \notin R \end{aligned}$$

USet:

$$\begin{aligned} S &\in \mathbb{IP}(E) \\ \text{Add}(a) : S & \quad \lambda S'. \text{if } (a \notin S) \text{ then } S' \cup \{a\} \text{ else } S' \end{aligned}$$

```

Remove(a):S
  λS'.if(a ∈ S and a ∈ S') then S'\{a} else S'
Lookup(a):S
  a ∈ S

RGAs:
A ∈ IP(E × I × I), R ∈ IP(I)
AddRight(e,a,i):(A,R)
  λ(A',R').
    if (∃ (_,e,_) ∈ AUA' and e ∉ R and ∀ (_,e',_) ∈ A.e'<i) then
      (A' ∪ {(a,i,e)},R)
    else
      (A',R')

Remove(i):(A,R)
  λ(A',R').
    if (∃ (_,i,_) ∈ A) then
      (A', R' ∪ {i})
    else
      (A',R')

RGAs-No-Tomb:
S ∈ IP(E × I × I)
AddRight(e,a,i):S
  λS'.
    if (∃ (_,e,_) ∈ SUS' and ∀ (_,e',_) ∈ S.e'<i) then
      S' ∪ {(a,i,e)}
    else
      S'

Remove(i):S
  λS'.
    if (∃ (_,i,_) ∈ S) then
      S'\{(_,i,_) }
    else
      S'

2P2P-Graph:
VA ∈ IP(E), VR ∈ IP(E), EA ∈ IP(E × E), ER ∈ IP(E × E)
AddVertex(v):(VA,VR,EA,ER)
  λ(VA',VR',EA',ER').
    (VA' ∪ {v},VR',EA',ER')

RemoveVertex(v):(VA,VR,EA,ER)
  λ(VA',VR',EA',ER').
    if (v ∈ VA \ VR and ∀ (u,w) ∈ EA \ ER.u ≠ v and w ≠ v and v ∈ VA') then

```

```

    (VA', VR' ∪ {v}, EA', ER')
  else
    (VA', VR', EA', ER')

```

```

AddEdge(u, v): (VA, VR, EA, ER)
λ(VA', VR', EA', ER').
  if (u ∈ VA \ VR and v ∈ VA \ VR) then
    (VA', VR', EA' ∪ {(u, v)}, ER')
  else
    (VA', VR', EA', ER')

```

```

RemoveEdge(u, v): (VA, VR, EA, ER)
λ(VA', VR', EA', ER').
  if ((u, v) ∈ EA \ ER and u ∈ VA \ VR and v ∈ VA \ VR and (u, v) ∈ EA') then
    (VA', VR', EA', ER' ∪ {(u, v)})
  else
    (VA', VR', EA', ER')

```

Graph-with-ORSet

```

V : ORSet(E), E : ORSet(E × E)

```

```

AddVertex(v): (V, E)
λ(V', E').
  (ORSet.Add(v): V(V'), E')

```

```

RemoveVertex(v): (V, E)
λ(V', E').
  if (ORSet.Lookup(v): V and ∄(u, w). (ORSet.Lookup((u, w)): E or
    ORSet.Lookup((u, w)): E') and (u = v or w = v)) then
    (ORSet.Remove(v): V(V'), E')
  else
    (V', E')

```

```

AddEdge(u, v): (V, E)
λ(V', E').
  if (ORSet.Lookup(u): V and ORSet.Lookup(v): V and ORSet.Lookup(u): V'
    and ORSet.Lookup(v): V') then
    (V', ORSet.Add((u, v)): E(E'))
  else
    (V', E')

```

```

RemoveEdge(u, v): (V, E)
λ(V', E').
  (V', ORSet.Remove((u, v)): E(E'))

```