Replication-Aware Linearizability*

Chao Wang
IRIF, CNRS and University Paris Diderot, FR
wangch@irif.fr

Constantin Enea
IRIF, CNRS and University Paris Diderot, FR
cenea@irif.fr

Suha Orhun Mutluergil
IRIF, CNRS and University Paris Diderot, FR
mutluergil@irif.fr

Gustavo Petri
ARM Research, UK
gustavo.petri@arm.com

Abstract

Distributed systems often replicate data at multiple locations to achieve availability and performance despite network partitions. These systems accept updates at any replica and propagate them asynchronously to every other replica. Conflict-Free Replicated Data Types (CRDTs) provide a principled approach to the problem of ensuring that replicas are eventually consistent despite the asynchronous delivery of updates.

We address the problem of specifying and verifying CRDTs, introducing a new correctness criterion called Replication-Aware Linearizability. This criterion is inspired by linearizability, the de-facto correctness criterion for (shared-memory) concurrent data structures. We argue that this criterion is both simple to understand, and it fits most known implementations of CRDTs. We provide a proof methodology to show that a CRDT satisfies replication-aware linearizability which we apply on a wide range of implementations. Finally, we show that our criterion can be leveraged to reason modularly about the composition of CRDTs.

CCS Concepts • **Theory of computation** → **Logic and verification**; • **Software and its engineering** → **Formal software verification**;

Keywords replicated data types, verification, weak-consistency

ACM Reference Format:
Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. 2019. Replication-Aware Linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3314221.3314617

*All authors but first listed in alphabetical order.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314617

1 Introduction

Conflict-Free Replicated Data Types (CRDTs) [20] have recently been proposed to address the problem of availability of a distributed application under network partitions. CRDTs represent a methodological attempt to alleviate the problem of retaining some data-Consistency and Availability under network Partitions (CAP), famously known to be an impossible combination of requirements by the CAP theorem of Gilbert and Lynch [11]. CRDTs are data types designed to favor availability over consistency by replicating the type instances across multiple nodes of a network, and allowing them to temporarily have different views. However, CRDTs guarantee that the different states of the nodes will *eventually* converge to a state common to all nodes [6, 20]. This *convergence property* is intrinsic to the data type design and in general no synchronization is needed, hence achieving availability.

Availability vs. Consistency. To illustrate the problem we consider the implementation of a list-like CRDT object, the Replicated Growable Array (RGA) – due to Roh et al. [19][1] –, used for text-editing applications. RGA supports three operations: 1. `addAfter(a,b)` which adds the character `b` – the concrete type is inconsequential here – immediately after the occurrence of the character `a` assumed to be present in the list,[2] 2. `remove(a)` which removes `a` assumed to be present in the list, and 3. `read()` which returns the list contents.

To make the system available under partitions, RGA allows each of the nodes to have a copy of the list instance. We will call each of the nodes holding a copy a *replica*. RGA allows any of the replicas to modify the *local* copy of the list immediately – and hence return control to the client – and lazily propagate the updates to the other replicas. For instance, assuming that we have an initial list containing the sequence $\mathtt{a}\cdot\mathtt{b}\cdot\mathtt{e}\cdot\mathtt{f}$ [3] and two replicas, $\mathtt{r}_1$ and $\mathtt{r}_2$, if $\mathtt{r}_1$ inserts the letter `c` after `b` (calling `addAfter(b,c)`), while $\mathtt{r}_2$ concurrently inserts the letter `d` after `b` (`addAfter(b,d)`) the replicas will have the states $\mathtt{a}\cdot\mathtt{b}\cdot\mathtt{c}\cdot\mathtt{e}\cdot\mathtt{f}$ and $\mathtt{a}\cdot\mathtt{b}\cdot\mathtt{d}\cdot\mathtt{e}\cdot\mathtt{f}$ respectively. We have solved the availability problem, but we have introduced inconsistent states. This problem is only exacerbated by adding more replicas.

[1] We use a variation of code extracted from [3].
[2] We assume elements are unique, implemented with timestamps.
[3] We use $s_0 \cdot s_1$ to denote the composition of sequences s_0 and s_1.

Convergence. To restore the replicas to a consistent state, CRDTs guarantee that under conflicting operations – that is, operations that could lead to different states – there is a systematic way to *detect conflicts*, and there is a strategy followed by all replicas to *deterministically resolve conflicts*.

In the case of RGA, the implementation adds metadata to each item of the list identifying the originating replica as well as timestamp of the operation in that replica.[4] This metadata is enough to detect when conflicts have occurred. Generally there are a number of assumptions that are necessary for the metadata to detect conflicts (for instance that timestamps increase monotonically with time) which we shall discuss in the following sections. Then, for RGA it is enough to know whether two `addAfter` operations have conflicted by simply comparing the replica identifiers and their timestamps. In fact, this is a sound over-approximation of conflict since two concurrent `addAfter` operations have a real conflict only if their first arguments are the same (e.g. the element `b` in the example aforementioned). In such case, the strategy to resolve the conflict will always choose to order first the character added with the highest timestamp in the resulting list, and in the particular case where the timestamps should be the same, an arbitrary order among replicas will be used. In the example above, and assuming that the character `c` was added with timestamp t_1 and the character `d` was added with timestamp t_2, if $t_2 < t_1$ (for some order $\leq$ between timestamps), the list will converge to $\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c} \cdot \mathtt{d} \cdot \mathtt{e} \cdot \mathtt{f}$. We obtain the same result if $t_1 = t_2$ and assume that we have a replica order $<_r$, we have $\mathtt{r}_2 <_r \mathtt{r}_1$. Using an arbitrary order among replica identifiers is common in CRDT implementations to break ties among elements with equal timestamps. We will generally assume that metadata provides a strict ordering and ignore the details.

If the effects of all operations are delivered to all replicas eventually, the replicas will converge to the same state – assuming a quiescent period of time where no new operations are performed. This allows to eventually recover the consistency of the data type without giving away availability.

Specifications. The simplicity of the list data type allows for a somewhat simple conflict resolution strategy. However, this is not true for many other CRDT implementations. It is therefore critical to provide the programmer with a clear, and precise, specification of the allowed behaviors of the data type under conflicts. Unfortunately this is not an easy task. Many times the programmer has no option but to read the implementation to understand how the metadata is used to resolve conflicts, for instance by reading the algorithms by Shapiro et al. [20] (a case where the algorithms are particularly well documented). Recently Burckhardt [6], Burckhardt et al. [7] have developed a formal framework where CRDTs and other weakly consistent systems can be specified. However, we consider that reading these specifications is far from trivial for the average programmer, let alone writing new specifications. Evidently, having a formal specification is a necessary step towards the verification of the implementations of CRDTs.

Simpler specifications, *not simplistic* specifications. It is important to remark at this point that while it is our goal to make the specification of CRDTs simpler, we believe that it is impossible to make them coincide with their sequential data type counterparts. Most CRDTs will exhibit, due to concurrency and consistency relaxations, behaviors that are not possible in the sequential version of the type they represent. A notable instance is the Multi-Valued-Register (MVR), which resolves conflicts arising from concurrent updates to the register by storing multiple values. Hence, a subsequent read operation to the register might return a set of values rather than a single value. This is certainly a behavior that is not possible for a "traditional" register, and in fact, one that the programmer must be aware of. Our goal is to accurately specify the behaviors of the CRDT, meaning that often times, different implementations of the same underlying data type (say a register) will have different specifications if their conflict resolution allows for different behaviours, for instance the Last-Writer-Wins (LWW) and the MVR registers which will be mentioned later.

Paper Contributions. Inspired by linearizability [13] we propose a *new consistency criterion for CRDTs*, which we call *Replication-Aware Linearizability* (RA-linearizability). RA-linearizability both simplifies CRDT specifications, and allows us to give correctness proof strategies for these specifications. To satisfy RA-linearizability a data type must be so that any execution of a client interacting with an instance of the data type 1. should result in a state that can be obtained as a sequence (or linearization) of its updates – where we assume that all updates are executed sequentially– and 2. any operation reading the state of the data type instance should be justified by executing a *sub-sequence* of the above mentioned sequence of updates. For instance, for the RGA example, the state of the final list (when all updates are delivered) should be reachable by considering a sequence where all `addAfter` operations are executed sequentially.

Equipped with this criterion we show that many existing CRDTs are RA-linearizable. We provide both, their specification, and proofs showing that implementations respect the specification. We provide two different proof methodologies based on the structure of the conflict-resolution mechanism implemented by the CRDT. We categorize CRDT implementations into classes according to their conflict-resolution strategy. Encouragingly, most of the CRDTs by Shapiro et al. [20] can be proved RA-linearizable.

Given that our criterion is inspired by linearizability, we consider if it also preserves the same compositionality properties, i.e. whether the composition of a set of RA-linearizable objects is also RA-linearizable. While we show that this is not true in general, we show that compositionality can be achieved when we concentrate to specific classes of conflict resolution as described above.

[4] We ignore here conflicts due to `remove`. They are discussed in Sec. 2.

```
payload Ti-Tree N, Set Tomb
initial N = ∅, Tomb = ∅
addAfter(a,b) :
  generator :
    precondition : a = ∘ or (a != ∘ and (_,_,a) ∈ N and
        a ∉ Tomb)
    let t_b = getTimestamp()
  effector(a, t_b, b) :
    N = N ∪ {(a, t_b, b)}
remove(a) :
  generator :
    precondition : (_,_,a) ∈ N and a ∉ Tomb and a ≠ ∘
  effector(a) :
    Tomb = Tomb ∪ {a}
read() :
  let ret-list = traverse(N, Tomb)
  return ret-list
```

Listing 1. Replicated Growable Array (RGA) pseudo-code.

Finally, we have mechanized our methodologies to prove RA-linearizability. We use the verification tool Boogie [4] to encode our specifications, CRDTs, and prove the correctness of the implementations (proof scripts are available at [1]).

Complete proofs of the results in this paper and more details can be found in [9].

2 Overview

We give an informal description of our system model, and illustrate our contribution with two compelling CRDT implementations from [3, 20].

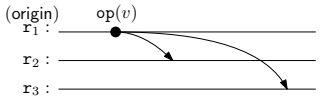


Figure 1. System Model.

We consider the implementation of CRDTs, and we focus on the behaviors of an *an instance* of the data type, generically called an *object*. We assume that objects are replicated among several *replicas*. Fig. 1 shows the execution of an operation `op(v)` evolving as follows: (i) a client submits an operation to some replica called *origin*, (ii) If the operation reads and updates the object state, the reading action is only performed at the origin. This part of the operation is called the *generator* (cf. [20]). Then, if the operation modifies the state – e.g. `addAfter` for RGA – an update is generated to be executed in every replica. This part of the operation shall be called the *effector*. We assume that effectors are executed immediately at the origin. This is represented by the dot at the origin replica in Fig. 1. (iii) Finally, the effector is delivered to each replica, and their states are updated consequently, represented by the target of the arrows. This model corresponds to *operation-based* CRDTs. Our results also apply to *state-based* CRDTs, where replicas exchange states instead of operations (Sec. 6).

2.1 RGA CRDT Implementation

Listing 1 presents the code of RGA in a style following that of Shapiro et al. [20] (a version of the RGA introduced in [3]).

The keyword `payload` declares the state used to represent the object: a variable `N` of type `Ti-Tree`, and a variable `Tomb` of type `Set`. The effectful operations `addAfter` and `remove` have two labels marked in red: `generator` and `effector`, corresponding to the reading and updating part of the operations as described above. Notice that the effector can use as arguments values produced by the generator. The `precondition` annotation indicates facts that are assumed about the state prior to the execution.

Reconsidering Fig. 1 the source of the arrows represents the execution of a `generator` jointly with the `effector` at replica r_1, and the target of the arrows represents the delivery and execution of the effector at replicas r_2 and r_3.

Each replica maintains a *Timestamp Tree* (`Ti-Tree`) containing in every tree node a pair with: the element added to the list (for instance the character `b`), and a timestamp associated to it (t_b) used to resolve conflicts. We will encode the tree as a set of triples (corresponding to nodes) of the form (a, t_b, b) representing an element `b` in the tree with timestamp t_b and whose parent is item `a` also present in the tree. The tree-ness property is ensured by construction.

The `generator` portion of `addAfter(a,b)` has a precondition requiring `a` to exist in the tree before the insertion of `b` (the data structure is initialized with a preexisting element ∘). The generator then samples a timestamp t_b for `b` which is assumed to be larger than any timestamp presently in the `Ti-Tree N` of the origin replica.[5] The `effector` portion of `addAfter(a,b)` adds the triple (a, t_b, b) in the replica's own copy of `N`. This ensures that the tree structure is consistent with the causality of insertions in the data structure. A client of the object will only ever attempt to add an element after another element which it has already seen as mandated by the `addAfter` API. Hence, the parent node of any node was inserted before it, and is causally related to it. Similarly, nodes that are not related to each other on any path of the tree (eg. siblings) are not causally related. An example of such a tree is shown in the left most box of Fig. 2: elements `c` and `b` were concurrently added after `a`, and `a` was added first after the initial element ∘.

From a `Ti-tree`, we can obtain a list by traversing the tree in pre-order, with the proviso that siblings are ordered according to their timestamps with the *highest timestamp visited first*. The leftmost box in Fig. 2 shows a tree that results in the list $a \cdot b \cdot c$ assuming the timestamp order $t_a < t_c < t_b$.

Fig. 2 shows two concurrent operations `addAfter(c,d)` and `addAfter(c,e)` executing in two different replicas starting both with the state depicted on the left. Then, the two trees result in different lists in each replica before the operations are mutually propagated.

We have so far ignored `remove`. Consider the case where a replica executes `addAfter(a,b)` on a replica while another one executes `remove(a)`. If the `addAfter(a,b)` effector reaches some replica after the effector of `remove(a)` there is a problem since the precondition of the effector of `addAfter(a,b)` requires that the element `a` be present in the `Ti-tree` of the replica. To avoid this kind of conflict,

[5] Also, t_b cannot be sampled by another replica (as we discussed in Sec. 1 this can be ensured by tagging the timestamps with replica identifiers).

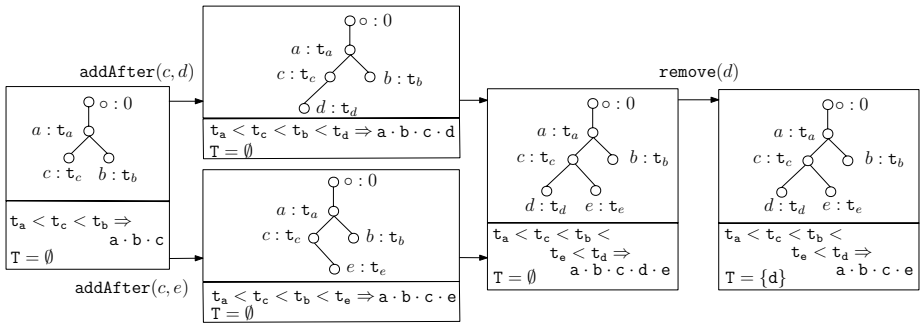


Figure 2. Example of RGA conflict resolution.

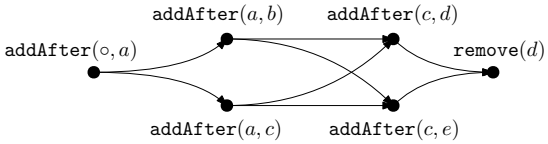


Figure 3. A history for the RGA object.

rendering the operations commutative, RGA does not really remove elements from the `Ti-tree`. Instead, an additional data structure called a *tombstone* is used to keep track of elements that have been conceptually erased and should not be considered when reading the list. Here, the marking of tombstones is a set `Tomb` of elements. The last column of Fig. 2 shows the result of a `remove` operation.

The method `read` performs the pre-order traversal explained before, where all elements in the tombstone `Tomb` are omitted. In each of the boxes of Fig. 2 the list shown represents the result of a `read` operation in the state depicted.

Operations, histories and linearizability. We consider an abstract view of executions of a CRDT object called a *history*. Informally a history is a set of operations with a partial order representing the ordering constraints imposed on the execution of each operation. We represent the execution of an operation with a label of the form $\mathsf{m}(a) \Rightarrow b$ representing a call to method m with arguments a and returning the value b. When the values are unimportant we shall use the meta-variable ℓ to denote a label. The partial order mentioned above represents the *visibility* relation among operations. We say that an operation with label ℓ_1 is visible to an operation with label ℓ_2 if at the time when ℓ_2 was executed at the origin replica, the effects of ℓ_1 had been applied in the state of the replica executing ℓ_2. A history is a pair $(\mathsf{L}, \prec)$ containing a set of labels L and a visibility relation $\prec$ between labels. Fig. 3 pictures a history where the last three operations are exactly those of the execution in Fig. 2. Each node represents a label and arrows represent that the operation at the source of the arrow is visible to the operation at the target. Since we assume that visibility is transitive we ignore redundant arrows.

A similar notion of history is used in the context of *linearizability* [13]. The only difference is that the order $\prec$ relates two operations the first of which returns before the other one started. A history $(\mathsf{L}, \prec)$ is called linearizable if there exists a *sequential* history $(\mathsf{L}, \prec_{\mathsf{seq}})$ ($\prec_{\mathsf{seq}}$ is a total order), called *linearization*, s.t. $(\mathsf{L}, \prec_{\mathsf{seq}})$ is a valid execution, and $\prec \subseteq \prec_{\mathsf{seq}}$.

CRDTs are not linearizable since operations are propagated lazily, so two replicas can see non-coinciding sets of operations. We relax linearizability to adapt it to CRDTs as follows: 1. we require that the sequential history be consistent with the visibility relation among operations instead of the returns-before order, and 2. operations that only read the state of the object are allowed to see a *sub-sequence* of the linearization, instead of the whole prefix as in the case of linearizability. (We will discuss an additional relaxation in Sec. 2.2).

Intuition of RGA RA-linearizability. To simplify, consider the linearization of two concurrent operations adding after a common element: `addAfter(a,b)` and `addAfter(a,c)`. This example corresponds to the history shown in the first three nodes of Fig. 3 from left to right. Because these operations are concurrent they are not related by visibility so our criterion allows for any ordering among them. Let us show that these operations can always be ordered in a way that the result of future reads will match this ordering. From the previous explanation we know that the order between `b` and `c` in the resulting list will be determined by their corresponding timestamps ($\mathtt{t_b}$ and $\mathtt{t_c}$). Assuming that the ordering is that given in the tree of the first column of Fig. 2, we know that we can order the operations as `addAfter(a,c)` followed by `addAfter(a,b)` which when executed sequentially obviously results in $\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}$ as shown. The timestamp metadata of RGA gives us a strategy to build the operation sequence that corresponds to a sequential specification. A concrete linearization of these operations is:

$$\mathtt{addAfter}(\circ, a) \cdot \mathtt{addAfter}(a, c) \cdot \mathtt{addAfter}(a, b)$$

Unfortunately this simple linearization strategy is not always applicable. Consider now a similar case where after issuing the `addAfter` operations the replicas attempt to immediately read the state. As explained in Fig. 2, a possible behavior is that the first replica returns $\circ \cdot \mathtt{a} \cdot \mathtt{b}$ while the second returns $\circ \cdot \mathtt{a} \cdot \mathtt{c}$. If we consider the linearization given above, the result $\circ \cdot \mathtt{a} \cdot \mathtt{b}$ is not possible, since `c` was added before `b` was added. This is because the reading replica has not yet seen $\mathtt{addAfter}(\mathtt{a}, \mathtt{c})$. To overcome this problem we allow methods that read the state to see a *sub-sequence* of the global linearization. Thus, we can consider the sequence

```
payload Set S
initial S = ∅
add(a) :
  generator :
    let k = getUniqueIdentifier()
    return k
  effector(a, k) :
    S = S ∪ {(a, k)}
remove(a) :
  generator :
    let R = {(a,k) | (a,k) ∈ S}
    return R
  effector(R) :
    S = S \ R
read() :
  let A = {a : ∃ k. (a,k) ∈ S}
  return A
```

Listing 2. Pseudo-code of the OR-Set CRDT.

$$\texttt{addAfter}(\circ, a) \cdot \texttt{addAfter}(a, c) \cdot \texttt{read}() \Rightarrow (\circ \cdot a \cdot c) \cdot \texttt{addAfter}(a, b) \cdot \texttt{read}() \Rightarrow (\circ \cdot a \cdot b)$$

where the last `read` ignores the red label $\texttt{addAfter}(a, c)$. These are only two cases of conflicting concurrent operations, in Sec. 4 we show that all operations can be ordered such that they correspond to a sequential execution thereof.

2.2 OR-Set CRDT Implementation

The Observed-Remove Set (OR-Set) [20] implements a set with operations: `add(a)`, `remove(a)`, `read()`. The code of OR-Set is shown in Listing 2 (we assume return values for `add(a)` and `remove(a)` for technical reasons).

Although the meaning of these methods is self-evident from their names, the results of conflicting concurrent operations is not evident. Consider for example the case where two replicas add a certain element `a` and then one of them removes that element. If we consider an interleaving based execution of these operations there are two options depending on the interleaving: i) If `remove(a)` is the last operation then the expected set is empty, since the two consecutive `add(a)` are idempotent, and the `remove` would remove the only occurrence of `a`. This interleaving is the one depicted with solid arrows in Fig. 4. ii) On the other hand, if the operation `add(a)` of the non-removing process comes last, as depicted with the dashed arrows in Fig. 4, the final set could contain the element `a`. As we have explained before, the operations can arrive in different orders to different replicas. To guarantee convergence, OR-Set must ensure that regardless of the ordering, the resulting set will be the same. To that end, OR-Set `add` operations will tag each added element with a unique identifier. Then, a remove operation will only remove the element-identifier pairs which has already seen. For instance, in the case (ii) above, the remove of `a` will only remove the element that has been previously added by the same replica, since this item has been observed by the `remove` operation – and thus its identifier is known to it. The concurrent `add(a)` operation will have an identifier that has not been observed by the `remove` Therefore the item will not be removed, even if the effectors of the two adds are performed in a replica before the effector of the remove.

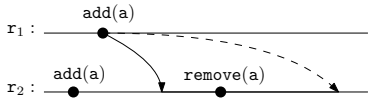


Figure 4. Interleaving-based Set.

Intuition of OR-Set Linearizability. It is easy to find examples where the implementation of OR-Set can produce executions that cannot be justified by the standard definition of linearizability (even with the relaxations discussed in Sec. 2.1) assuming a standard Set specification. Fig. 5a shows one such example. Clearly any linearization of the visibility relation in this execution should order the `add` and `remove` updates before the `read` queries, and the linearization of the updates should end with a `remove`. Therefore, the final set returned by each of the two `read` queries should have at most one element (the `read` queries see all the updates in the execution), contrary to their return value in this execution.

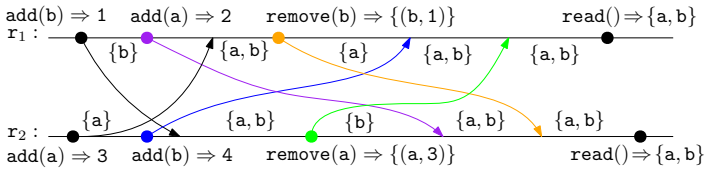
This execution shows that the `remove` operation behaves as both a query (observing a certain number of adds of the element to be removed) and an update (by removing said observed elements). To cope with such cases, we will consider in our definition that query-update operations can be split into a query part corresponding to the generator, which only reads the state – and hence is allowed to see a sub-sequence of the linearization of updates – and an update part corresponding to the effector which will use the results of the prior query. For instance, `remove` will be split into a query part `readIds` where only the elements visible at the time of the remove are selected, and an update part `remove` where only those elements selected are erased. Any identifier not in the set returned by `readIds` will remain in the set after the update part of `remove`. Evidently, this requires some mechanism for "marking" the adds that are concerned. We will consider that each add has a unique identifier. Fig. 5b shows this rewriting. The result of the rewriting admits a linearization consistent with the specification of Set, as explained above.

3 Replication-Aware Linearizability

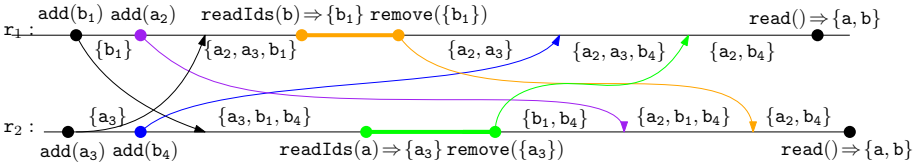
In this section we formalize the intuitions developed in Sec. 2. We define the semantics of CRDT objects (§ 3.1), specifications (§ 3.2), and our notion of RA-linearizability (§ 3.3). For lack of space, our formalization focuses only on operation-based CRDTs. However, the notion of RA-linearizability applies to state-based CRDTs as well (see Section 6).

3.1 The Semantics of CRDT objects

To formalize the semantics of CRDT objects and our correctness criterion we use several semantic domains defined in Fig. 6. We will use operation labels of the form $\mathsf{o.m}(a) \overset{i,ts}{\Rightarrow} b$ to represent the call of a method $\mathsf{m} \in \mathbb{M}$ of object $\mathsf{o} \in \mathbb{O}$, with argument $a \in \mathbb{D}$, resulting in the value $b \in \mathbb{D}$, and generating the timestamp ts. Since there might be multiple calls to the same method with the same arguments and result, labels



(a) OR-Set non-linearizable execution. Each line represents operations issued to the same replica.



(b) Label rewriting of an OR-Set execution. Pairs (a, k) of an element a and identifier k are written as a_k.

Figure 5. OR-Set Linearizability vs. RA-linearizability.

are tagged with a unique identifier i. We may omit the object o, the identifier i, the timestamp ts, or the return value b when they are not important. The order relation on $\mathbb{T}$ is denoted by $<$. Abusing notations, we assume that the set $\mathbb{T}$ contains a distinguished minimal element $\perp$ which we shall use for operations that do not generate a timestamp such as the method `remove` of RGA. The timestamp ts of a label $\ell = \mathsf{o}.\mathsf{m}(a) \overset{i,ts}{\Rightarrow} b$ is denoted $\mathsf{ts}(\ell)$. The set of all operation labels is denoted by $\mathbb{L}$.

Given a CRDT object o, its semantics is defined as a labeled transition system (LTS) $[\![\mathsf{o}]\!] = (\mathsf{GC}, \mathbb{A}, \mathsf{gc}_0, \rightarrow)$, where GC is a set of global configurations, $\mathbb{A}$ is the set of transition labels called *actions*, gc_0 is the initial configuration, and $\rightarrow \subseteq \mathsf{GC} \times \mathbb{A} \times \mathsf{GC}$ is the transition relation.

Our semantics assumes the following two properties of the propagation of effectors: (i) the effector of each operation is applied exactly once at each replica, and (ii) if the effector of operation ℓ_1 is applied at the origin replica of ℓ_2 before ℓ_2 happens, then for every replica r, the effector of ℓ_2 will be applied only after the effector of ℓ_1 has already been applied. These are commonly referred to as *causal delivery*. We assume causal delivery because our formalization focuses on operation-based CRDTs. However, the notion of RA-linearizability and the compositionality results in Section 5 apply to state-based CRDTs as well, even if the network infrastructure doesn't satisfy causal delivery (see [9]).

A global configuration $(\mathsf{G}, \mathsf{vis}, \mathsf{DS})$ is a "snapshot" of the system that records all the operations that have been executed. $\mathsf{G} \in [\mathbb{R} \rightarrow \mathsf{LC}]$ [6] stores the local configuration of each replica (LC denotes the set of local configurations). A local configuration (L, σ) contains the state σ of a replica and the set L of labels of operations that originate at this replica, or whose effectors have been executed (or applied) at this replica. When $\ell \in \mathsf{L}$, we say that ℓ is *visible* to the replica or that the replica *sees* ℓ. The set of replica states σ is denoted by Σ. The relation $\mathsf{vis} \subseteq \mathcal{P}(\mathbb{L} \times \mathbb{L})$ is the *visibility* relation between operations, i.e., $(\ell_1, \ell_2) \in \mathsf{vis}$, where ℓ_2 is an operation originated at a replica r, if the effector of ℓ_1 was executed at r before ℓ_2 was executed. When $(\ell_1, \ell_2) \in \mathsf{vis}$, we say that ℓ_1 is *visible* to ℓ_2, or that ℓ_2 *sees* ℓ_1. As it will be clear from the definition of the transition relation, vis is a *strict partial order*. Finally, $\mathsf{DS} \in [\mathbb{L} \rightarrow \Delta]$ associates to each operation label $\ell \in \mathsf{L}$ an effector $\delta \in [\Sigma \rightarrow \Sigma]$, which is the replica state transformer generated when the operation was executed at the origin replica (Δ denotes the set of effectors). For some fixed initial replica state σ_0, the initial global configuration is defined by $\mathsf{gc}_0 = (\mathsf{G}_0, \emptyset, \emptyset) \in \mathsf{GC}$, where G_0 maps each replica r into $(\emptyset, \sigma_0)$.

The transition relation between global configurations is defined in Fig. 7. The first rule describes a replica r in state σ executing an invocation of method m with argument a. We use a function θ to represent the behavior of the generators of all methods collectively (the code under the `generator` labels), i.e., $\theta(\sigma, \mathsf{m}, a)$ stands for applying the generator of

[6] We use $[A \rightarrow B]$ to denote the set of total functions from A to B.

$$\begin{array}{llll} \mathsf{o} \in \mathbb{O} & \text{CRDT Objects} & a, b \in \mathbb{D} & \text{Data} \\ \mathsf{r} \in \mathbb{R} & \text{Replicas} & ts \in \mathbb{T} & \text{Timestamps} \\ \mathsf{m} \in \mathbb{M} & \text{Methods} & \mathsf{L} \subseteq \mathbb{L} & \text{Label Set} \end{array}$$

$$\ell \equiv \mathsf{o}.\mathsf{m}(a) \overset{i,ts}{\Rightarrow} b \in \mathbb{L} \quad \text{Operation Label}$$

Figure 6. Semantic Domains.

$$\frac{\textsc{Operation} \quad \mathsf{G}(\mathsf{r}) = (\mathsf{L}, \sigma) \quad \theta(\sigma, \mathsf{m}, a) = (b, \delta, ts) \quad \delta(\sigma) = \sigma' \quad \ell = \mathsf{o}.\mathsf{m}(a) \overset{(i,ts)}{\Rightarrow} b \quad \mathit{unique}(i) \quad ts \neq \perp \Rightarrow (\forall \ell' \in \mathsf{L}.\ \mathsf{ts}(\ell') < ts) \quad \forall \ell' \in \mathsf{labels}(\mathsf{vis}).\ \mathsf{ts}(\ell') \neq ts}{(\mathsf{G}, \mathsf{vis}, \mathsf{DS}) \xrightarrow{\mathsf{gen}_\mathsf{r}(\ell)} (\mathsf{G}[\mathsf{r} \leftarrow (\mathsf{L} \cup \{\ell\}, \sigma')], \mathsf{vis} \cup (\mathsf{L} \times \{\ell\}), \mathsf{DS}[\ell \leftarrow \delta])}$$

$$\frac{\textsc{Effector} \quad \mathsf{G}(\mathsf{r}) = (\mathsf{L}, \sigma) \quad \ell \in \mathsf{min}_{\mathsf{vis}}(\mathsf{labels}(\mathsf{vis}) \setminus \mathsf{L}) \quad \mathsf{DS}(\ell) = \delta \quad \delta(\sigma) = \sigma'}{(\mathsf{G}, \mathsf{vis}, \mathsf{DS}) \xrightarrow{\mathsf{eff}_\mathsf{r}(\ell)} (\mathsf{G}[\mathsf{r} \leftarrow (\mathsf{L} \cup \{\ell\}, \sigma')], \mathsf{vis}, \mathsf{DS})}$$

Figure 7. Operational Semantics of CRDTs. $C[a \leftarrow b]$ denotes the in-place update of element a of the domain of C with value b; $\mathit{unique}(i)$ to ensure that i is a unique identifier; and $\mathsf{labels}(\mathsf{vis}) = \{\ell : \exists \ell'.\ (\ell, \ell') \in \mathsf{vis} \vee (\ell', \ell) \in \mathsf{vis}\}$.

m with argument a on the replica state σ. We assume that timestamps are consistent with the visibility relation vis, i.e., the timestamp ts generated by θ is strictly larger than all the timestamps of operations visible to r, and that each timestamp can be generated only once. This transition is labeled by $\mathsf{gen}_{\mathsf{r}}(\ell)$ where ℓ is the label of this invocation. We may ignore the index r when it is not important.

The second rule describes a replica r in state σ executing the effector δ that corresponds to an operation ℓ originated in a different replica. This transition is labeled by $\mathsf{eff}_{\mathsf{r}}(\ell)$.

We say that a method $\mathsf{m} \in \mathbb{M}$ is a *query* if it always results (by applying the generator) in an identity effector δ (i.e. $\delta(\sigma) = \sigma$ for all replica states σ). We shall call an *update* any method m which is not a query – that is, whose effectors are not the identity function – and whose resulting effector and return value do not depend on the initial state σ of the origin replica. More formally, assuming a functional equivalence relation $\equiv$ between effectors that relates any two effectors that have the same effect (modulo the values of timestamps or unique identifiers) m is called an update when $\theta(\sigma, \mathsf{m}, a)|_2 \equiv \theta(\sigma', \mathsf{m}, a)|_2$, for every $a \in \mathbb{D}$ and two states $\sigma, \sigma' \in \Sigma$ (for a tuple x, $x|_k$ denotes the projection of x on the k-th component). A method m which is not a query nor an update is called a *query-update*. For instance, the methods `addAfter` and `remove` of RGA, and `add` of OR-Set, are updates, the method `remove` of OR-Set is a query-update, and the `read` methods of both the RGA and the OR-Set are queries. We denote by Queries, Updates, and Query-Updates, the sets of operation labels $\mathsf{o}.\mathsf{m}(a) \overset{i,ts}{\Rightarrow} b$ where m is a query, an update, or query-update respectively.

An execution of the object o is a sequence of transitions $\mathsf{gc}_0 \xrightarrow{a_0} \mathsf{gc}_1 \xrightarrow{a_1} \ldots$. A *trace* tr is the sequence of actions $a_0 \cdot a_1 \ldots$ labeling the transitions of an execution. The set of traces of an object o is denoted by $\mathbb{T}\mathsf{r}(\mathsf{o})$. A *history* is a pair $(\mathsf{L}, \mathsf{vis})$ where $\mathsf{vis} \subseteq \mathsf{L} \times \mathsf{L}$ is an acyclic relation over the set of labels L. Given an execution e ending in a global configuration $(\mathsf{G}, \mathsf{vis}, \mathsf{DS})$, the *history* of e, denoted by $h(e)$, is the pair $(\mathsf{labels}(\mathsf{vis}), \mathsf{vis})$. Note that the relation vis is a strict partial order in this case. Also, the history of a trace tr, denoted by $h(tr)$, is the history of the execution that corresponds to tr. The set of histories $\mathbb{H}\mathsf{ist}(\mathsf{o})$ of an object o is the set of histories h of an execution e of o. A pictorial representation of an execution (trace) can be found in Fig. 5a while an example of a history can be found in Fig. 3.

3.2 Sequential Specifications

RA-linearizability provides an explanation for concurrent executions of CRDT objects in the form of linearizations, which can be constrained using standard sequential specifications.

Definition 3.1 (Sequential Specification)**.** A *sequential specification* (specification, for short) Spec is a set of tuples $(\mathsf{L}, \mathsf{seq})$, where L is a set of labels and seq is a sequence including all the labels in L.

To describe sequential specifications in a succinct way we will provide an operational description. To that end, we will associate to specifications a notion of abstract state, which we shall generally denote by ϕ and its domain shall be denoted by Φ. Then, to each valid label ℓ we will associate a transition relation $\phi \overset{\ell}{\hookrightarrow} \phi'$ which, given an abstract state ϕ and provided that the label ℓ can be applied in ϕ, produces a new abstract state ϕ'. In the specific case where the label ℓ assumes a certain precondition pre over the initial abstract state ϕ we will use Hoare-style preconditions and write $\big(\phi \mid \mathsf{pre}(\phi)\big) \overset{\ell}{\hookrightarrow} \phi'$. In this way, a sequential specification is the set of label sequences that are obtained by the successive application of the transition relation starting from a given initial state ϕ_0.

Example 3.2 (Sequential Specification of RGA)**.** Each abstract state $\phi = (l, T)$ contains a sequence l of elements of a given type and a set T of elements appearing in the list. The element l is the list of all input values, whether already removed or not; while T stores the removed values and is used as *tombstone set*. The sequential specification Spec(RGA) of list with add-after interface is defined by:

$$\begin{aligned}
\big(\,(l_1 \cdot b \cdot l_2, T) \mid a \text{ fresh}\,\big) &\xhookrightarrow{\mathtt{addAfter}(b,a)} (l_1 \cdot b \cdot a \cdot l_2, T)\\
\big(\,(l, T) \mid b \in l \text{ and } b \neq \circ\big) &\xhookrightarrow{\mathtt{remove}(b)} (l, T \cup \{b\})\\
(l, T) &\xhookrightarrow{\mathtt{read}()\Rightarrow(l/T)} (l, T)
\end{aligned}$$

where we denote by l/T the list resulting from removing all elements of T from l. The method $\mathtt{addAfter}(b, a)$ puts a immediately after b in l, assuming that each value is put into list at most once. Method $\mathtt{remove}(b)$ adds b into T. Finally $\mathtt{read}() \Rightarrow s$ returns the list content excluding any element appearing in T. Assume that the initial value of list is $(\circ, \emptyset)$, and $\circ$ is never removed. We will sometimes ignore the value $\circ$ from the return of `read`.

Example 3.3 (Sequential Specification of OR-Set)**.** As explained in Fig. 5b, the fact that the OR-Set `remove` method is a query-update induces a rewriting of the operation labels in a history. This rewriting introduces update operations $\mathtt{add}(a, id)$, for some identifier id, instead of simply $\mathtt{add}(a)$, and $\mathtt{remove}(S)$, for some set S of pairs element-identifier, instead of $\mathtt{remove}(a)$, and a new query operation $\mathtt{readIds}(a)$ that returns a set of pairs element-identifier. These operations are specified as follows. The abstract state ϕ is a set of tuples (a, id), where a is a data and id is a identifier. The sequential specification Spec(OR-Set) of OR-Set is given by the transitions:

$$\begin{aligned}
\phi &\xhookrightarrow{\mathtt{readIds}(a)\Rightarrow S} \phi \quad [S = \{(a, id) \mid (a, id) \in \phi\}]\\
\phi &\xhookrightarrow{\mathtt{remove}(S)} \phi \setminus S\\
\big(\,\phi \mid (a, id) \notin \phi\,\big) &\xhookrightarrow{\mathtt{add}(a, id)} \phi \cup \{(a, id)\}\\
\phi &\xhookrightarrow{\mathtt{read}(a)\Rightarrow A} \phi \quad [A = \{a \mid \exists\, id, (a, id) \in \phi\}]
\end{aligned}$$

Here $\mathtt{readIds}(a) \Rightarrow S$ returns the set of pairs with data a, $\mathtt{remove}(S)$ removes S from the abstract state, $\mathtt{add}(a, id)$ puts

$\{(a, id)\}$ into the abstract state, and $\texttt{read}() \Rightarrow A$ returns the value of the OR-Set.

3.3 Definition of Replication-Aware Linearizability

We now provide the definition of RA-linearizability which characterizes histories of CRDT objects. To simplify the presentation, we consider first the case where all the labels in the history are either queries or updates (query-updates are considered later). The intuition of RA-linearizability is that there is a *global* sequence (or linearization) of the update operations in an execution which can produce the state of *each* replica when *all* the updates are visible to them. Each query should be justified by considering the sub-sequence of the global sequence restricted to the updates that are visible to that query. To be precise:

Definition 3.4. A history $h = (\mathsf{L}, \mathsf{vis})$ with $\mathsf{L} \subseteq \text{Queries} \uplus \text{Updates}$ is RA-linearizable w.r.t. a sequential specification Spec, if there exists a sequence $(\mathsf{L}, \mathsf{seq})$ such that:

(i) seq is consistent with vis, that is: $\mathsf{vis} \cup \mathsf{seq}$ is acyclic,
(ii) the projection of seq to *updates* is admitted by Spec, i.e. $\mathsf{seq}{\downarrow_{\text{Updates}}} \in$ Spec, where we denote by $\mathsf{seq}{\downarrow_S}$ the restriction of the order seq to the set S, and
(iii) for each query $\ell_{\mathsf{qr}} \in \mathsf{L}$, the sub-sequence of updates visible to ℓ_{qr} together with ℓ_{qr} is itself admitted by Spec, i.e., $\mathsf{seq}{\downarrow_{\mathsf{vis}^{-1}(\ell_{\mathsf{qr}}) \cap \text{Updates}}} \cdot \ell_{\mathsf{qr}} \in$ Spec.

We say that $(\mathsf{L}, \mathsf{seq})$ is an *RA-linearization* of h w.r.t. Spec.

The sequences of operations provided in Sec. 2.1 and 2.2 are RA-linearizations.

We now consider the case where histories include query-updates. In such case, we apply Definition 3.4 on a rewriting of the original history where each query-update is decomposed into a label representing the generator and another label representing the effector. A mapping $\gamma : \mathbb{L} \to \mathbb{L}^{\leq 2}$, where $\mathbb{L}^{\leq 2}$ is the set of labels and pairs of labels in $\mathbb{L}$, is called a *query-update rewriting*. We assume that every query or update label is mapped by γ to a singleton and that the γ image of such a label preserves its status, i.e., $\gamma(\ell)$ is a query, resp., update, whenever ℓ is a query, resp., update. Also, query-updates labels ℓ are mapped to pairs $\gamma(\ell) = (\ell_1, \ell_2)$ where ℓ_1 is a query and ℓ_2 is an update. These assumptions are important when applying Definition 3.4 on the rewriting of a history, since this definition relies on a partitioning of the labels into queries and updates. For a history $h = (\mathsf{L}, \mathsf{vis})$, its γ-rewriting is a history $\gamma(h) = (\mathsf{L}', \mathsf{vis}')$ where

- L' is obtained by replacing each label ℓ in L with $\gamma(\ell)$ (a label may be replaced by two labels),
- whenever a (query-update) label ℓ is mapped by γ to a pair (ℓ_1, ℓ_2), we have that the query is ordered before the update, formally $(\ell_1, \ell_2) \in \mathsf{vis}'$,
- vis' preserves the order between labels which are mapped to singletons, and for any query-update label ℓ mapped to a pair (ℓ_1, ℓ_2), the query ℓ_1 sees exactly the same set of operations as ℓ and any operation which saw ℓ must see ℓ_2. Formally, whenever $(\ell, \ell') \in \mathsf{vis}$ we have that $(\mathsf{upd}(\gamma(\ell)), \mathsf{qry}(\gamma(\ell'))) \in \mathsf{vis}'$, where for a label ℓ, $\mathsf{qry}(\gamma(\ell))$ (resp., $\mathsf{upd}(\gamma(\ell))$), is $\gamma(\ell)$ when $\gamma(\ell)$ is a singleton, or its first (resp., second) component when $\gamma(\ell)$ is a pair.

Example 3.5 (Query-Update Rewriting of OR-Set)**.** As shown in Fig. 5b, the query-update rewriting for OR-Set is defined by: $\gamma(\texttt{add}(a) \Rightarrow k) = \texttt{add}(a, k)$, $\gamma(\texttt{read}() \Rightarrow \texttt{A}) = \texttt{read}() \Rightarrow \texttt{A}$, and $\gamma(\texttt{remove}(a) \Rightarrow R) = (\texttt{readIds}(a) \Rightarrow R, \texttt{remove}(R))$.

The following extends Definition 3.4 to arbitrary histories using the rewriting defined above.

Definition 3.6 (Replication-Aware Linearizability)**.** A history $h = (\mathsf{L}, \mathsf{vis})$ is RA-linearizable w.r.t. Spec, if there exists a query-update rewriting γ s.t. $\gamma(h)$ is RA-linearizable w.r.t. Spec. An RA-linearization w.r.t. Spec of $\gamma(h)$ is called an RA-linearization w.r.t. Spec and γ of h.

A set H of histories is called RA-linearizable w.r.t. Spec when each $h \in H$ is RA-linearizable w.r.t. Spec. A data type implementation is RA-linearizable w.r.t. Spec if for any object o of the data type, $\mathbb{H}\text{ist}(\texttt{o})$ is linearizable w.r.t. Spec.

Reasoning with specifications. To illustrate the benefit of using RA-linearizability let us consider a simple system where two replicas execute a sequence of operations on a shared OR-Set object:

$$\texttt{add(a); rem(a); X = read()} \;\|\; \texttt{add(a); Y = read()}$$

We are interested in checking that the following post-condition holds after the execution of these operations:

$$\texttt{a} \in \texttt{X} \Rightarrow \texttt{a} \in \texttt{Y}$$

Rewriting the program according to the specification of OR-Set discussed before, we obtain the following, where the variable R represents the set of value timestamp pairs observed by the readIds operation as defined by the rewriting:

$$\begin{bmatrix} \texttt{add(a, i}_1\texttt{);} \\ \texttt{readIds(a)} \Rightarrow \texttt{R;} \\ \texttt{rem(R);} \\ \texttt{X = read();} \\ \{\texttt{a} \in \texttt{X} \Rightarrow (a, i_2) \notin \texttt{R}\} \end{bmatrix} \;\|\; \begin{bmatrix} \texttt{add(a, i}_2\texttt{);} \\ \texttt{Y = read();} \\ \{(a, i_2) \notin \texttt{R} \Rightarrow \texttt{a} \in \texttt{Y}\} \end{bmatrix}$$

Post-condition : $\{\texttt{a} \in \texttt{X} \Rightarrow \texttt{a} \in \texttt{Y}\}$

Since OR-Set is RA-linearizable w.r.t. the specification in Example 3.3 (proved in Section 4.1), the possible values of X and Y can be computed by enumerating their RA-linearizations. The post-condition follows from the conjunction of the assertions in each replica. Let us consider the validation of the assertion of right hand side with the following RA-linearization:

We have in red color and with solid arrows the operations of the right hand side replica, and in blue with dashed arrows the left ones. Let us consider the sub-sequence of the linearization that is visible to the last operation (Y = read()). Since the first operation $(\texttt{add(a, i}_2\texttt{)})$ is issued on the same replica, it must

be visible to it. Let us now consider different cases for the operations of the other replica that are visible to the read: (a) if the remove operation rem(R) is not visible to it, then the assertion is trivially true, because (a, i_2) is in the resulting set according to the specification, and therefore the consequent of the assertion is valid. Assume from now on that rem(R) is visible to it, there are two cases (b) if (a, i_2) does not belong to R the consequent of the assertion is valid, since the addition of (a, i_2) is necessarily visible to the read operation, and we conclude as before, (c) on the other hand, if $(a, i_2) \in R$ we have that the antecedent of the implication is falsified, and therefore the assertion is also valid.

Here we have considered only one RA-linearization, but it is not hard to see that commuting the operations of the different replicas renders the same argument. Importantly, this reasoning was done entirely at the level of the RA-linearizations (i.e. the specification) of the data type.

For the assertion on the left hand side replica, since visibility includes the order between operations issued on the same replica, we get that $\mathsf{add}(a, i_1)$ is ordered before $\mathsf{readIds}(a) \Rightarrow R$ in every RA-linearization. Since $\mathsf{add}(a, i_1)$ is also visible to $\mathsf{readIds}(a) \Rightarrow R$, we get that $(a, i_1) \in R$. Similarly, every RA-linearization will order rem(R) before the read() on the left replica, which implies that if $a \in X$, then $(a, i_2) \notin R$. Assuming the contrary, i.e., $(a, i_2) \in R$, implies that $R = \{(a, i_1), (a, i_2)\}$ and since rem(R) is visible and linearized before $X = \mathsf{read}()$, we get that $a \notin X$.

4 Proving Replication-Aware Linearizability

We describe a methodology for proving that CRDT objects are RA-linearizable which relies on two properties: (1) the effectors of any two concurrent operations (i.e., not visible to each other) commute, which is inherent to CRDT objects, and (2) the existence of a refinement mapping [2, 17] showing that each effector produced by an operation ℓ, respectively each query ℓ, is simulated by the execution of ℓ (or its counterpart through a query-update rewriting γ) in the specification Spec. This methodology is used in two forms depending on how the linearization is defined along an execution, which may affect the precise definition of the refinement mapping.

4.1 Execution-Order Linearizations

We first consider the case of CRDT objects, e.g., OR-Set, for which the order in which operations are executed at the origin replica defines a valid RA-linearization. We say that such objects admit *execution-order linearizations*. We start by formalizing the two properties we use to prove RA-linearizability.

Given a history $h = (\mathsf{L}, \mathsf{vis})$, we say that two operations ℓ_1 and ℓ_2 are *concurrent*, denoted $\ell_1 \bowtie_{\mathsf{vis}} \ell_2$, when $(\ell_1, \ell_2) \notin \mathsf{vis}$ and $(\ell_2, \ell_1) \notin \mathsf{vis}$. In general, CRDTs implicitly require that the effectors of concurrent operations commute:

Commutativity: for every trace tr with $h(tr) = (\mathsf{L}, \mathsf{vis})$, and every two operations $\ell_1, \ell_2 \in \mathsf{L}$, if $\ell_1 \bowtie_{\mathsf{vis}} \ell_2$, then

$$\forall \sigma \in \Sigma.\ \delta_{\ell_1}(\delta_{\ell_2}(\sigma)) = \delta_{\ell_2}(\delta_{\ell_1}(\sigma))$$

where δ_{ℓ_1} and δ_{ℓ_2} are the effectors of ℓ_1 and resp., ℓ_2.

Example 4.1. For OR-Set, two `add`, resp., `remove`, effectors commute because they both add, resp., remove, element-id pairs, while an `add` and a `remove` effector commute when they are concurrent because the element-id pairs removed by the `remove` effector are different from the pair added by the `add` effector (since the `add` is not visible to `remove`).

Commutativity implies that for every linearization lin of the operations in an execution, which is consistent with the visibility relation, every replica state σ in that execution can be obtained by applying the delivered effectors in the order defined by lin (between the operations corresponding to those effectors). Indeed, by the causal delivery assumption, the order in which effectors are applied at a given replica is also consistent with visibility. Therefore, the only differences between the order in which effectors were applied to obtain σ in that execution and the linearization order lin involve effectors of concurrent operations, which commute.

Lemma 4.2. *Let ρ be an execution of an object* o *satisfying* Commutativity, $h = (\mathsf{L}, \mathsf{vis})$ *the history of ρ, and* $(\mathsf{L}, \mathsf{seq})$ *a linearization of the operations in* L *(possibly, rewritten using a query-update rewriting γ), consistent with* vis*. For each local configuration* $(\mathsf{L}_\mathsf{r}, \sigma_\mathsf{r})$ *in ρ,*

$$\sigma_\mathsf{r} = \delta_{\ell_n}(\ldots(\delta_{\ell_1}(\sigma_0))\ldots)$$

where δ_ℓ denotes the effector of operation ℓ, σ_0 is the initial replica state, and $\mathsf{seq} \downarrow_{\mathsf{L}_\mathsf{r}} = \ell_1 \ldots \ell_n$.

In order to relate the CRDT object with its specification we use refinement mappings, which are "local" in the sense that they characterize the evolution of a single replica in isolation. A *refinement mapping* abs associates replica states with states of the specification, such that any update or query applied on a replica state σ can be mimicked by the corresponding operation in the specification starting from $\mathsf{abs}(\sigma)$. Moreover, the resulting states in the two steps must be again related by abs. Formally, given a query-update rewriting γ, we define Refinement as the existence of a mapping abs such that:

Simulating effectors: For every effector δ corresponding to a (query-)update operation ℓ, and every state $\sigma \in \Sigma$,

$$\sigma' = \delta(\sigma) \Rightarrow \mathsf{abs}(\sigma) \xhookrightarrow{\mathsf{upd}(\gamma(\ell))} \mathsf{abs}(\sigma')$$

where $\hookrightarrow$ is the transition function of Spec.

Simulating generators: For every query m, and every $\sigma \in \Sigma$,

$$\theta(\sigma, \mathsf{m}, a) = (b, \_, \_) \Rightarrow \mathsf{abs}(\sigma) \xhookrightarrow{\ell} \mathsf{abs}(\sigma)$$

where $\ell = \mathsf{m}(a) \Rightarrow b$. Recall that $\theta(\sigma, \mathsf{m}, a)$ stands for applying the generator of m with argument a on the

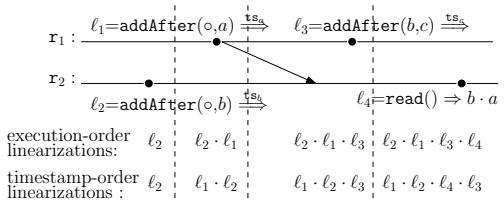


Figure 8. Execution-order and timestamp-order linearizations for RGA. Here $ts_a < ts_b < ts_c$.

state σ. Also, for every query-update m, and $\sigma \in \Sigma$,

$$\theta(\sigma, \mathsf{m}, a) = (b, \_, \_) \Rightarrow \mathsf{abs}(\sigma) \xhookrightarrow{\mathsf{qry}(\gamma(\ell))} \mathsf{abs}(\sigma).$$

Example 4.3. Consider the OR-Set object, its specification in Example 3.3, and the query-update rewriting in Example 3.5. Also, let abs be a refinement mapping defined as the identity function. The effector of an add(a) ⇒ k operation, rewritten by γ to add(a, k), and the add(a, k) operation of the specification have the same effect. Similarly, the effector of a query-update remove(a) ⇒ R operation, rewritten by γ to (readIds(a) ⇒ R, remove(a, R)), and the remove(a, R) operation of the specification have the same effect. Applying the query operation read() on a state σ results in the same return value A as applying the same query in the context of the specification on the state $\mathsf{abs}(\sigma) = \sigma$. Finally, for the query-update remove(a) ⇒ R, executing its generator in a state σ results in the same return value R as executing the query readIds(a) ⇒ R introduced by the query-update rewriting in the specification state $\mathsf{abs}(\sigma) = \sigma$.

Next, we show that any object o satisfying Commutativity and Refinement is RA-linearizable. Given a history $h = (\mathsf{L}, \mathsf{vis})$ of a trace tr, the *execution-order linearization* of h is the sequence $(\gamma(\mathsf{L}), \mathsf{seq})$ such that $\gamma(\ell_1)$ occurs before $\gamma(\ell_2)$ in seq iff $\mathsf{gen}(\ell_1)$ occurs before $\mathsf{gen}(\ell_2)$ in tr, for every two labels $\ell_1, \ell_2 \in \mathsf{L}$. An object o *admits* execution-order linearizations if for any history $h = (\mathsf{L}, \mathsf{vis})$ of a trace tr, the execution-order linearization is an RA-linearization of h w.r.t. Spec and γ.

Theorem 4.4. *Any object that satisfies* Commutativity *and* Refinement *admits execution-order linearizations.*

4.2 Timestamp-Order Linearizations

CRDT objects such as RGA in Listing 1, that use timestamps for conflict resolution, may not admit execution-order linearizations. For instance, Fig. 8 shows an execution of RGA where two replicas r_1 and r_2 execute two addAfter invocations, and an addAfter invocation followed by a read invocation, respectively. An execution-order linearization which by definition, is consistent with the order in which the operations are applied at the origin replica, will order addAfter(∘, b) before addAfter(∘, a). The result of applying these two operations in this order in the specification Spec(RGA) (defined in Example 3.2) is the list $a \cdot b$. However, if the timestamp ts_a of a is smaller than the timestamp ts_b of b, a read that sees these two operations will return the list $b \cdot a$, which is different than the one obtained in the context of Spec(RGA). Therefore, we consider a variation of the proof methodology described in Sec. 4.1 where the linearizations are additionally *consistent with the order of timestamps* generated by the operations. For instance, in the execution of Fig. 8, addAfter(∘, a) will be ordered before addAfter(∘, b) because ts_a is smaller than ts_b (irrespective of the order between the generators). Moreover, to extend the notion of timestamp ordering to operations ℓ that don't generate timestamps, i.e., invocations of remove and read, we consider a "virtual" timestamp which is defined as the *maximal* timestamp of any operation visible to ℓ (or $\perp$ if no operation is visible to ℓ), and require that the linearization is consistent with the order between both "real" and "virtual" timestamps. For instance, the "virtual" timestamp of the read in Fig. 8 is ts_b because it sees addAfter(∘, a) and addAfter(∘, b). Then, a valid RA-linearization will order the read operation before the other addAfter(b, c) operation, since the timestamp ts_c of the latter is bigger than the "virtual" timestamp ts_b of the read. The operations that have the same timestamp (which is possible due to "virtual" timestamps) are ordered as they execute at the origin replica. For instance, the read with "virtual" timestamp ts_b is ordered after addAfter(∘, b) that has the same timestamp ts_b since it executes later at the origin replica.

Formally, for a history $h = (\mathsf{L}, \mathsf{vis})$, we define the timestamp $\mathsf{ts}_h(\ell)$ of a label ℓ in the context of the history h to be $\mathsf{ts}_h(\ell) = \mathsf{ts}(\ell)$ if $\mathsf{ts}(\ell) \neq \perp$ and $\mathsf{ts}_h(\ell) = \max\{\mathsf{ts}(\ell') : (\ell', \ell) \in \mathsf{vis}\}$, otherwise. Given a history $h = (\mathsf{L}, \mathsf{vis})$ of a trace tr, the *timestamp-order linearization* of h is the sequence $(\mathsf{L}, \mathsf{seq})$ such that $\gamma(\ell_1)$ occurs before ℓ_2 in seq iff $\mathsf{ts}_h(\ell_1) < \mathsf{ts}_h(\ell_2)$ or $\mathsf{gen}(\ell_1)$ occurs before $\mathsf{gen}(\ell_2)$ in tr, for every two labels $\ell_1, \ell_2 \in \mathsf{L}$. An object o *admits* timestamp-order linearizations if for any history $h = (\mathsf{L}, \mathsf{vis})$ of a trace tr, the timestamp-order linearization is an RA-linearization of h w.r.t. Spec. [7]

Proving admittance of timestamp-order linearizations relies on Commutativity and a slight variation of Refinement where intuitively, an effector generating a timestamp ts has to be simulated by a specification operation only when it is applied on a state σ that doesn't "store" a greater timestamp than ts (other effectors are treated as before). Formally, the set $\mathsf{ts}(\sigma)$ of timestamps in a state σ contains all the timestamps ts generated by effectors applied to obtain σ. For RGA, the set of timestamps in a state σ is the set of all timestamps stored in its timestamp tree. We define $\mathsf{Refinement}_{\mathsf{ts}}$ by modifying the "Simulating effectors" part of Refinement as follows:

Simulating effectors: For every effector δ of an operation ℓ,

$$\forall \sigma \in \Sigma.\ \mathsf{ts}(\ell) \not< \mathsf{ts}(\sigma) \wedge \sigma' = \delta(\sigma) \Rightarrow \mathsf{abs}(\sigma) \xhookrightarrow{\ell} \mathsf{abs}(\sigma')$$

[7] For simplicity, we ignore query-update rewritings. The CRDTs with timestamp-order linearizations we investigated don't require such rewritings.

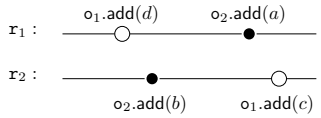


Figure 9. A history of two OR-Sets. Each operation is visible only at the origin, so visibility is given by the horizontal lines.

Example 4.5. Let us consider the RGA object, its specification in Example 3.2, and a refinement mapping abs which relates a replica state $(\mathtt{N}, \mathtt{Tomb})$ with a specification state (l, T) where the sequence l is given by the function `traverse` in `read` queries when ignoring tombstones, i.e., $l = \mathtt{traverse}(\mathtt{N}, \emptyset)$, and $T = \mathtt{Tomb}$. It is obvious that `remove` effectors and `read` queries are simulated by the corresponding specification operations. Effectors of $\mathtt{addAfter}(\mathtt{a},\mathtt{b}) \overset{ts_\mathtt{b}}{\Rightarrow}$ operations are simulated by the specification operation $\mathtt{addAfter}(\mathtt{a},\mathtt{b})$ only when $ts_\mathtt{b}$ is greater than all the timestamps stored in the replica state where it applies. Thus, let $(\mathtt{N}, \mathtt{Tomb})$ be a replica state such that $ts < ts_\mathtt{b}$ for every ts with $(\_, ts, \_) \in \mathtt{N}$. The result of applying the effector δ of $\mathtt{addAfter}(\mathtt{a},\mathtt{b}) \overset{ts_\mathtt{b}}{\Rightarrow}$ is to add b as a child of a. Then, applying `traverse` on the new state will result in a sequence where b is placed just after a because it has the highest timestamp among the children of a. This corresponds exactly to the sequence obtained by applying the operation $\mathtt{addAfter}(\mathtt{a},\mathtt{b})$ in the context of the specification.

The proof of an object o admitting timestamp-order linearizations if it satisfies Commutativity and $\mathsf{Refinement}_{\mathsf{ts}}$ is similar to the one of Theorem 4.4.

Theorem 4.6. *Any object that satisfies* Commutativity *and* $\mathsf{Refinement}_{\mathsf{ts}}$ *admits timestamp-order linearizations.*

We remark that the API of a CRDT can impact on whether it is RA-linearizable. For instance, a slight variation of the RGA in Listing 1 with the same state, but with an interface with a method `addAt(a,k)` to insert an element `a` at an index `k`, introduced in [3], would not be RA-linearizable w.r.t. an appropriate sequential specification (see [9]).

5 Compositionality of RA-Linearizability

We investigate the issue of whether the composition of a set of objects satisfying RA-linearizability is also RA-linearizable. While this is not true in general, we show that the composition of objects that admit execution-order or timestamp-order linearizations is RA-linearizable under the assumption that they share the same timestamp generator.

5.1 Object Compositions and RA-Linearizability

Given two objects o_1 and o_2, the semantics of their composition $\mathsf{o}_1 \otimes \mathsf{o}_2$ is the standard product of the LTSs corresponding to o_1 and o_2, respectively. The history of a trace tr of $\mathsf{o}_1 \otimes \mathsf{o}_2$ records a "global" visibility relation between the operations in the trace, i.e., which operations of o_1 or o_2 are visible when issuing an operation of o_1, and similarly, for operations of o_2.

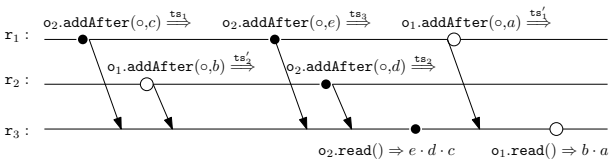


Figure 10. A history in the composition ⊗ of two RGAs.

Formally, $h(tr) = (\mathsf{L}, \mathsf{vis})$ where L is the set of labels occurring in tr, and $(\ell_1, \ell_2) \in \mathsf{vis}$ if there exists a replica r such that $\mathsf{eff}_\mathsf{r}(\ell_1)$ occurs before $\mathsf{gen}_\mathsf{r}(\ell_2)$ in the trace tr. In general, vis may not be a partial order since the causal delivery assumption holds only among operations of the same object. The set of histories $\mathbb{H}\mathsf{ist}(\mathsf{o}_1 \otimes \mathsf{o}_2)$ of the composition $\mathsf{o}_1 \otimes \mathsf{o}_2$ is the set of histories h of a trace tr of $\mathsf{o}_1 \otimes \mathsf{o}_2$.

For two specifications Spec_1 and Spec_2 of two objects o_1 and o_2, respectively, the composition $\mathsf{Spec}_1 \otimes \mathsf{Spec}_2$ is the set of interleavings of sequences in Spec_1 and Spec_2, respectively. We say that the composition $\mathsf{o}_1 \otimes \mathsf{o}_2$ is *RA-linearizable* if every history of $\mathsf{o}_1 \otimes \mathsf{o}_2$ is RA-linearizable w.r.t. $\mathsf{Spec}_1 \otimes \mathsf{Spec}_2$. The extension to a set of objects is defined as usual.

Linearizability [13] ensures that for every history, any per-object linearizations, concerning the operations of a single object, can be combined into a global linearization, concerning all the operations in the history. However, this is not true for our notion of RA-linearizability. A counterexample is given in Fig. 9. The operations of o_1 can be linearized to $\mathsf{o}_1.\mathtt{add(c)} \cdot \mathsf{o}_1.\mathtt{add(d)}$ while the operations of o_2 can be linearized to $\mathsf{o}_2.\mathtt{add(a)} \cdot \mathsf{o}_2.\mathtt{add(b)}$. There is no RA-linearization of this history whose projections on each of the two objects correspond to these per-object linearizations.

5.2 Composition: Execution-Order Linearizability

Although not all per-object RA-linearizations can be combined into global RA-linearizations, this may still be true in some cases. For the history in Fig. 9, the operations of o_1 can also be linearized to $\mathsf{o}_1.\mathtt{add(d)} \cdot \mathsf{o}_1.\mathtt{add(c)}$ which enables a global linearization $\mathsf{o}_1.\mathtt{add(d)} \cdot \mathsf{o}_2.\mathtt{add(a)} \cdot \mathsf{o}_2.\mathtt{add(b)} \cdot \mathsf{o}_1.\mathtt{add(c)}$ whose projection on each object is consistent with the per-object linearization (we take the same linearization for o_2).

We show that in the case of RA-linearizable objects that admit execution-order linearizations, there always exist per-object RA-linearizations that can be combined into global RA-linearizations, hence their composition is RA-linearizable and moreover, it also admits execution-order linearizations.

Theorem 5.1. *The composition of a set of RA-linearizable objects that admit execution-order linearizations is RA-linearizable and admits execution-order linearizations.*

5.3 Composition: Timestamp-Order Linearizability

Theorem 5.1 does not apply to objects that admit timestamp-order linearizations. The "unrestricted" object composition ⊗ allows different objects to generate timestamps independently, and in "conflicting" orders along some execution. For instance, Fig. 10 shows a history with two RGA objects o_1

$$\frac{\begin{array}{c}\text{OPERATION} \qquad \ell = \mathsf{o}_k.\mathsf{m}(a) \overset{(i,ts)}{\Rightarrow} b \text{ with } k \in \{1,2\} \\ (\mathsf{G}_k, \mathsf{vis}_k, \mathsf{DS}_k) \xrightarrow{\mathsf{gen}_\mathsf{r}(\ell)}_k (\mathsf{G}'_k, \mathsf{vis}'_k, \mathsf{DS}'_k) \\ (\mathsf{G}'_{k'}, \mathsf{vis}'_{k'}, \mathsf{DS}'_{k'}) = (\mathsf{G}_{k'}, \mathsf{vis}_{k'}, \mathsf{DS}_{k'}) \text{ for } k' \neq k \qquad \mathsf{G}_1(\mathsf{r}) = (\mathsf{L}_1, \sigma_1) \\ \mathsf{G}_2(\mathsf{r}) = (\mathsf{L}_2, \sigma_2) \qquad ts \neq \bot \Rightarrow (\forall \ell' \in \mathsf{L}_1 \cup \mathsf{L}_2.\ \mathsf{ts}(\ell') < ts) \\ \forall \ell' \in \mathsf{labels}(\mathsf{vis}_1 \cup \mathsf{vis}_2).\ \mathsf{ts}(\ell') \neq ts\end{array}}{((\mathsf{G}_1, \mathsf{vis}_1, \mathsf{DS}_1), (\mathsf{G}_2, \mathsf{vis}_2, \mathsf{DS}_2)) \xrightarrow{\mathsf{gen}_\mathsf{r}(\ell)} (\mathsf{G}'_1, \mathsf{vis}'_1, \mathsf{DS}'_1), (\mathsf{G}'_2, \mathsf{vis}'_2, \mathsf{DS}'_2)}$$

Figure 11. The transition rule for generators in the object composition operator $\otimes_{\mathsf{ts}}$.

and o_2. We assume that $ts_1 < ts_2 < ts_3$ and $ts'_1 < ts'_2$ (the order between other timestamps is not important). The operations of o_1, resp., o_2, can be linearized to

- $\mathsf{o}_1.\mathtt{addAfter}(\circ, \mathsf{a}) \cdot \mathsf{o}_1.\mathtt{addAfter}(\circ, \mathsf{b}) \cdot \mathsf{o}_1.\mathtt{read}() \Rightarrow \mathsf{b} \cdot \mathsf{a}$
- $\mathsf{o}_2.\mathtt{addAfter}(\circ, \mathsf{c}) \cdot \mathsf{o}_2.\mathtt{addAfter}(\circ, \mathsf{d}) \cdot \mathsf{o}_2.\mathtt{addAfter}(\circ, \mathsf{e}) \cdot \mathsf{o}_2.\mathtt{read}() \Rightarrow \mathsf{e} \cdot \mathsf{d} \cdot \mathsf{c}$

These are the only RA-linearizations possible. There is no "global" linearization consistent with these per-object linearizations: ordering $\mathtt{addAfter}(\circ, \mathsf{a})$ before $\mathtt{addAfter}(\circ, \mathsf{b})$ implies that $\mathtt{addAfter}(\circ, \mathsf{e})$ occurs before $\mathtt{addAfter}(\circ, \mathsf{d})$ which contradicts the second linearization above. We solve this problem by constraining the composition operator $\otimes$ such that intuitively, all objects share a common timestamp generator. This ensures that each new timestamp is bigger than the timestamps used by operations delivered to a replica, independently of the object to which they pertain. For instance, the history of Fig. 10 would not be admitted because ts'_1 should be bigger than ts_3 (since the operation that received ts_3 from the timestamp generator originates from the same replica as the operation receiving ts'_1 at a later time) and ts_2 should be bigger than ts'_2. These two constraints together with $ts'_1 < ts'_2$ contradict $ts_2 < ts_3$. While this requires a modification of the algorithms, where the timestamp generator is a parameter, this has no algorithmic or run-time cost, and in fact a similar idea have been suggested in the systems literature (e.g. [10]).

We define a restriction $\otimes_{\mathsf{ts}}$ of the object composition $\otimes$ such that the set of histories $h = (\mathsf{L}, \mathsf{vis})$ in the composition $\mathsf{o}_1 \otimes_{\mathsf{ts}} \mathsf{o}_2$ satisfy the property that the order between timestamps (of all objects) is consistent with the visibility relation vis (i.e., $\mathsf{vis} \cup \prec_h$ is acyclic). With respect to the "unrestricted" composition $\otimes$ defined in Sec. 5.1, we only modify the transition rule corresponding to generators, as shown in Fig. 11. This ensures that a new generated timestamp is bigger than all the timestamps "visible" to the replica executing that generator (irrespectively of the object). The composition operator $\otimes_{\mathsf{ts}}$ is called *shared timestamp generator composition*. Practically, if we were to consider the standard timestamp mechanism used in CRDTs, i.e., each replica maintains a counter which is increased monotonically with every new operation (originating at the replica or delivered from another replica) and timestamps are defined as pairs of replica identifiers and counter values, then $\otimes_{\mathsf{ts}}$ can be implemented using a "shared" counter which increases monotonically with every new operation, independently of the object on which it is applied.

The following theorem shows that the composition of RA-linearizable objects that admit execution-order or timestamp-order linearizations is RA-linearizable, provided that all the objects share the same timestamp generator.

Theorem 5.2. *The shared timestamp generator composition of a set of RA-linearizable objects that admit execution-order or timestamp-order linearizations is RA-linearizable.*

6 Mechanizing RA-Linearizability Proofs

To validate our approach, we considered a range of CRDTs listed in Fig. 12 and mechanized their RA-linearizability proofs using Boogie [4], a verification tool. More precisely, we mechanized the proofs of conditions like Commutativity and Refinement which imply RA-linearizability by the results in Section 4. Beyond operation-based CRDTs (discussed in the paper), we have also considered *state-based* CRDTs, where an update occurs only at the origin, and replicas exchange their *states* instead of operations, and states from other replicas are merged at the replica receiving them. The merge function corresponds to the least upper bound operator in a certain join semi-lattice defined over replica states.

For operation-based CRDTs, we have mechanized the proof of a strenghtening of Commutativity that avoids reasoning about traces and the proof of Refinement (or $\mathsf{Refinement}_{\mathsf{ts}}$). Concerning Commutativity, our proofs encode two effectors as a single procedure which executes on two equal copies of the replica state. In some cases, the precondition of this procedure encodes conditions which are satisfied anytime the two effectors are concurrent, e.g., the effector of an `add` and resp., `remove` of OR-Set are concurrent when the argument `k` of `add` is not in the argument `R` of `remove`. At least for the CRDTs we consider, such characterizations are obvious and apply generically to any conflict-resolution policy based on unique identifiers. In some cases, the effectors commute even if they are not concurrent, so no additional precondition is needed. We prove that the resulting states are identical after performing the effectors in different order in each of the states. Refinement (or $\mathsf{Refinement}_{\mathsf{ts}}$) is reduced to proving that the refinement mapping is an inductive invariant for a lock-step execution of the CRDT implementation and its specification.

For state-based CRDTs, we have identified a set of conditions similar to those of operation-based CRDTs that imply RA-linearizability (see [9]). In this case, we don't rely on the causal delivery assumption. Extending their semantics with an auxiliary variable maintaining a correspondence between replica states and sets of operations that produced them, we extract the visibility relation between operations as in the case of operation-based CRDTs. This enables a similar reasoning about RA-linearizability. In particular, Commutativity is replaced by few conditions that now characterize the relationship between applying operations at a given replica and the merge function.

CRDT	Imp.	Lin.
Counter [20]	OB	EO
PN-Counter [20]	SB	EO
LWW-Register [15]	OB	TO
Multi-Value Reg. [8]	SB	EO
LWW-Element Set [20]	SB	TO

CRDT	Imp.	Lin.
2P-Set [20]	SB	EO
OR-Set [20]	OB	EO
RGA [19]	OB	TO
Wooki [22]	OB	EO

Figure 12. CRDTs proved RA-linearizable and the class of linearizations used. SB: State-Based, OB: Operation-Based, EO: Execution-Order, TO: Timestamp-Order.

7 Related Work

Correctness Criteria. Burckhardt et al. [7] gives the first formal framework where CRDTs and other weakly consistent replicated systems can be specified. Their CRDT specifications are defined in terms of sets of *partial orders* as opposed to our sequential specifications, which we think are easier to reason about when verifying clients. Beyond simpler specifications, RA-linearizability is related to their formalization of causal consistency, called *causal convergence* in [5]. Overall RA-linearizability differs from causal convergence in three points: (1) query-update rewritings, which enable sequential specifications and avoid partial orders, (2) the linearization projected on updates must be admitted by the specification (intuitively, this ensures that the "final" convergence state is valid w.r.t. the specification), and (3) the linearization is required to be consistent with the visibility order from the execution, and not an arbitrary one as in causal convergence. The latter difference makes causal convergence not compositional.

Regarding convergence, RA-linearizability implies that there is a unique total order of updates, and therefore if at some point all updates are visible to all replicas, all subsequent query operations at any replica will return the same value. This is observably equivalent to strong eventual consistency [12, 20, 23]. RA-linearizability is also stronger than the session guarantees of Terry et al. [21], but weaker than sequential consistency [16] and linearizability [13]. RA-linearizable objects that admit execution-order linearizations are close to being linearizable since the operations are linearized as they were issued at the origin replica, relative to wall-clock time. This is similar to linearizability, where each operation appears to take effect instantaneously between the wall-clock time of its invocation its response. Unlike linearizability, RA-linearizability allows queries to return a response consistent with only a subsequence of its linearized-before operations.

Sequential Specifications for CRDTs. Perrin et al. [18] provides Update Consistency (UC), a criterion which to the best of our knowledge is the first to consider sequential specifications and characterize linear histories of operations. However UC is not compositional due to an existential quantification over visibility relations like in causal convergence. Moreover, Perrin et al. [18] doesn't investigate UC proof methodologies.

Jagadeesan and Riely [14] provide a correctness criterion called *SEC*, which differs from RA-linearizability in several points: 1. Firstly, RA-linearizability has a global total order for updates, unlike SEC whose definition is quite complex. 2. Secondly, CRDT specifications in SEC are parameterized by a *dependency* relation at the level of the type's API. Then, SEC assumes that all independent operations commute and disregards their order even when issued by the same client. It is unclear how such a specification could adequately capture systems enforcing session guarantees [21]. 3. While SEC is also compositional, since operations from different objects are assumed independent, a history of two different SEC objects is trivially SEC since the order between operations of different objects is ignored. We find this notion of composition problematic since the composition of specifications cannot capture causality between different objects, a common pattern when writing distributed applications (e.g. for referential integrity in a key-value store). In RA-linearizability the composition of a set of objects respects the client's causality as illustrated by the failure to combine some per-object linearizations in Fig. 9. [8]

Verification of CRDTs There are several works that approach the problem of verifying that a CRDT implementation is correct w.r.t. a specification. In [3, 6, 7] along with the formal specification, proofs of correctness of implementations are given for several CRDTs. Our Refinement property is inspired by the Replication Aware Simulations in [7]. Zeller et al. [23] and Gomes et al. [12] provide frameworks for the verification of CRDTs in Isabelle/HOL. Their proofs are similar to the simulations of [7], albeit in a different specification language also based on partial orders.

8 Conclusion

We presented RA-linearizability, a correctness criterion inspired by linearizability, intended to simplify the specification of CRDTs by resorting to sequential reasoning for the specifications. We provide proof methodologies for RA-linearizability for some well documented CRDTs, and we prove that under certain conditions these proofs guarantee the compositionality of RA-linearizability. In the extended version of this paper [9] we show how our techniques extend to state-based CRDTs.

There are some limitations of RA-linearizability. Firstly, as we showed before, some CRDTs might not be RA-linearizable under a certain API, but a slight change in the API renders them RA-linearizable. We would like to investigate what constitutes an API that enables RA-linearizability specifications. secondly, while we argue that RA-linearizability simplifies specifications, we leave as future work to show whether it can be effectively used to verify client applications of a CRDT.

Acknowledgments

This work is partly supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177).

[8] There are however per-object linearizations for this history which can be merged into a global linearization (see Sec. 5).

References

[1] [n. d.]. https://github.com/menesro/RA-linearizability-proofs

[2] Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

[3] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. 2016. 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*. 259–268. https://doi.org/10.1145/2933057.2933090

[4] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. 364–387. https://doi.org/10.1007/11804192_17

[5] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 626–638. http://dl.acm.org/citation.cfm?id=3009888

[6] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Foundations and Trends in Programming Languages* 1, 1-2 (2014), 1–150. https://doi.org/10.1561/2500000011

[7] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. 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*. 271–284. https://doi.org/10.1145/2535838.2535848

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 205–220. https://doi.org/10.1145/1294261.1294281

[9] Constantin Enea, Suha Orhun Mutluergil, Gustavo Petri, and Chao Wang. 2019. Replication-Aware Linearizability. *CoRR* abs/1903.06560 (2019). arXiv:1903.06560 https://arxiv.org/abs/1903.06560

[10] Vitor Enes, Paulo Sérgio Almeida, and Carlos Baquero. 2017. The Single-Writer Principle in CRDT Composition. In *Proceedings of the Programming Models and Languages for Distributed Computing (PMLDC '17)*. ACM, New York, NY, USA, Article 4, 3 pages. https://doi.org/10.1145/3166089.3168733

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

[12] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying strong eventual consistency in distributed systems. *PACMPL* 1, OOPSLA (2017), 109:1–109:28. https://doi.org/10.1145/3133933

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

[14] Radha Jagadeesan and James Riely. 2018. Eventual Consistency for CRDTs. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 968–995. https://doi.org/10.1007/978-3-319-89884-1_34

[15] Paul R. Johnson and Robert Thomas. 1975. Maintenance of duplicate databases. *RFC* 677 (1975), 1–10. https://doi.org/10.17487/RFC0677

[16] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[17] Nancy A. Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995), 214–233. https://doi.org/10.1006/inco.1995.1134

[18] Matthieu Perrin, Achour Mostéfaoui, and Claude Jard. 2014. Update Consistency in Partitionable Systems. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*. 546–549. http://link.springer.com/content/pdf/bbm%3A978-3-662-45174-8%2F1.pdf

[19] Hyun-Gul Roh, Myeongjae Jeon, Jinsoo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.* 71, 3 (2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

[20] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

[21] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28-30, 1994*. IEEE Computer Society, 140–149. https://doi.org/10.1109/PDIS.1994.331722

[22] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2007. Wooki: A P2P Wiki-Based Collaborative Writing Tool. In *Web Information Systems Engineering - WISE 2007, 8th International Conference on Web Information Systems Engineering, Nancy, France, December 3-7, 2007, Proceedings (Lecture Notes in Computer Science)*, Boualem Benatallah, Fabio Casati, Dimitrios Georgakopoulos, Claudio Bartolini, Wasim Sadiq, and Claude Godart (Eds.), Vol. 4831. Springer, 503–512. https://doi.org/10.1007/978-3-540-76993-4_42

[23] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. 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*. 33–48. https://doi.org/10.1007/978-3-662-43613-4_3