Abstraction for Conflict-Free Replicated Data Types

Hongjin Liang
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, Jiangsu, China
hongjin@nju.edu.cn

Xinyu Feng*
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, Jiangsu, China
xyfeng@nju.edu.cn

Abstract

Strong eventual consistency (SEC) has been used as a classic notion of correctness for Conflict-Free Replicated Data Types (CRDTs). However, it does not give proper abstractions of *functionality*, thus is not helpful for modular verification of client programs using CRDTs. We propose a new correctness formulation for CRDTs, called Abstract Converging Consistency (ACC), to specify both data consistency and functional correctness. ACC gives abstract atomic specifications (as an abstraction) to CRDT operations, and establishes consistency between the concrete execution traces and the execution using the abstract atomic operations. The abstraction allows us to verify the CRDT implementation and its client programs separately, resulting in more modular and elegant proofs than monolithic approaches for whole program verification. We give a generic proof method to verify ACC of CRDT implementations, and a rely-guarantee style program logic to verify client programs. Our Abstraction theorem shows that ACC is equivalent to contextual refinement, linking the verification of CRDT implementations and clients together to derive functional correctness of whole programs.

CCS Concepts: • **Theory of computation → Program verification**; **Abstraction**; *Distributed algorithms*; • **Software and its engineering → Correctness**; **Semantics**.

Keywords: Replicated Data Types, Eventual Consistency, Contextual Refinement, Program Logic, Modular Verification

ACM Reference Format:
Hongjin Liang and Xinyu Feng. 2021. Abstraction for Conflict-Free Replicated Data Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454067

*Corresponding author.

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454067

1 Introduction

Replicated data types are distributed implementations of data types that replicate data in different nodes of geographically distributed systems to improve availability and performance. A correct implementation needs to ensure that clients accessing different replicas have a consistent view of the data. Unfortunately, the CAP theorem [7] shows that, in the presence of network partitions, it is impossible to achieve both availability and strong consistency.

Conflict-Free Replicated Data Types (CRDTs) [20] are recently proposed to address the tensions between availability and consistency. On the one hand, CRDTs are designed to have availability. The nodes executing CRDTs can process client requests without synchronization. Later the updates are sent to other nodes, asynchronously and possibly in different orders. On the other hand, since concurrent updates may conflict, CRDTs follow certain carefully-designed strategies to resolve conflicts and provide a weak form of consistency. For instance, the last-writer-wins registers [20] resolve conflicts between concurrent writes by enforcing a global total order among the writes using time-stamps. The main strategy of add-wins sets [20] is to enforce that an add always wins over a concurrent remove of the same element. Benefiting from the conflict resolution strategies, CRDTs guarantee *strong eventual consistency* (SEC) [20], where two nodes are guaranteed to converge (i.e., having identical states) once they have received the same set of updates.

Unfortunately, SEC fails to specify the functional correctness of CRDTs. It is unclear to what extent a CRDT algorithm really implements the desired data type. For instance, can the last-writer-wins registers ensure that every read receives the most recent write, and what is the most recent write? Do the add-wins sets always behave like sequential sets, and what does "behaving like sequential sets" mean exactly? More importantly, without proper abstraction about functionality of CRDTs, it is difficult to verify *client programs* of CRDTs in a modular and layered way.

We use "**let** Π **in** $C_1 \parallel \ldots \parallel C_n$" to represent a program consisting of client programs $C_1, \ldots, C_n$, and the implementation Π of a CRDT. The clients run on distributed nodes and access the CRDT by invoking the operations defined in Π. To reason about the behaviors of the whole program, we need to verify both the correctness of the CRDT implementation Π and the behaviors of the client programs. A proper abstraction Γ for the CRDT would allow us to verify them separately. As shown in Fig. 1, we only need to verify the correctness of the CRDT implementation Π with respect to the abstraction Γ once and for all, no matter in what context (i.e., the collection of clients) it is used. Then we reason about the clients as if they were using the abstract object Γ, without worrying about the implementation details in Π (e.g., time-stamps or various auxiliary data).

However, building a general abstraction mechanism and a framework for verifying functional correctness of CRDTs and their clients turns out to be extremely challenging, mostly because of the diversity of conflict resolution strategies. We observe that the strategies can be divided into two classes. Most CRDTs use uniform conflict resolution strategies (UCR), such as time-stamps, which do not give privilege to particular operations, while add-wins sets and remove-wins sets use operation-dependent conflict resolution strategies "X-wins". The latter case relies on the functionality and the semantic relationship between operations, which makes the reasoning much more difficult than the former case.

Contributions. In this paper, we try to build abstraction and verification frameworks for CRDTs of both classes. The abstraction is in the form of atomic object specifications Γ, which are traditionally used for sequential data types and shared-memory concurrent objects. To facilitate the client reasoning, each Γ is also accompanied with a conflict relation ⋈ which specifies non-commutative abstract operations of the object (see Sec. 4). Our specifications are simple, allowing one to easily tell what abstract data type a CRDT algorithm really implements. They are also abstract enough to hide low-level implementation details such as time-stamps.

For UCR-CRDTs, Fig. 1 gives an overview of our framework. We propose **Abstract Converging Consistency** (ACC), a new formulation of correctness (① in Fig. 1, also in Sec. 5). ACC establishes an abstract view of execution based on the atomic specifications Γ, so reflects the desired functionality. The abstract views of execution sequences may be different on different nodes, but they must be coherent on conflicting abstract operations (related in ⋈) so that SEC is guaranteed.

We prove the **Abstraction Theorem** (see Sec. 6), showing that ACC is equivalent to a contextual refinement between the concrete implementation Π of CRDT operations and the atomic specification Γ, where the specification is executed in *a novel abstract operational semantics*. The Abstraction Theorem allows one to reason about client programs at a high abstraction level, by replacing concrete CRDT implementations with the specifications. It decouples the verification of clients and CRDTs, as shown in Fig. 1. The contextual refinement can be viewed as an alternative and more client-friendly correctness formulation for UCR-CRDTs.

Based on the abstraction, we present a **rely-guarantee-style program logic** for verifying client programs at the high abstraction level (② in Fig. 1, also in Sec. 7). Together with the contextual refinement, our logic offers a way to

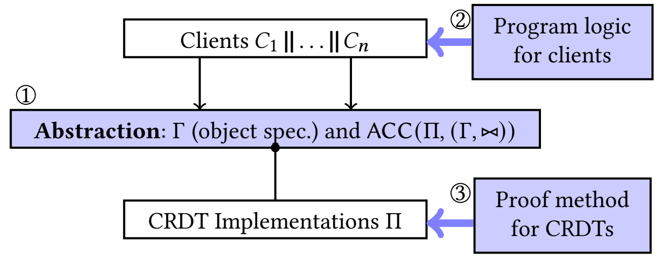


Figure 1. Our abstraction and verification framework.

verify the functional correctness of the whole system. We have applied our logic to reason about several interesting client programs (see our technical report [15]).

We also develop **a proof method** for systematically verifying ACC (③ in Fig. 1, also in Sec. 8). *We have applied it to verify seven major UCR-CRDT algorithms [20]*, including the replicated counter (with both increment and decrement operations), the grow-only set, the last-writer-wins (LWW) register, the LWW-element set, the 2P-set, the continuous sequence, and the replicated growable array (RGA).

To the best of our knowledge, our work gives the first framework for compositional verification of whole programs, including both UCR-CRDT implementations and client code, based on contextual refinement and the abstraction theorem. We actually show that different implementation algorithms for the same data type, such as the continuous sequence and RGA for lists, or the LWW-element set and the 2P-set for sets, can be verified using the *same* abstract specification. Verifying a client program of the data type in our framework guarantees its correctness no matter which specific implementation algorithm it uses.

For X-wins CRDTs, we extend the specification with the explicit operation-dependent conflict resolution strategy, and propose XACC as an extension of ACC for correctness definition. We still establish the Abstraction Theorem, by giving a more relaxed abstract semantics to clients with object specifications. We also verify the functional correctness of the add-wins set and remove-wins set with respect to XACC.

2 Informal Development

Below we discuss the main challenges to formalize the correctness of CRDTs, and give an overview of our approaches.

2.1 The RGA Example

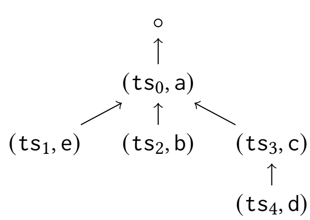
As a motivating example, Fig. 2 shows a simplified version [1] of the RGA algorithm [18] which in practice is the core algorithm for collaboratively edited documents. RGA implements a list object with three operations: `addAfter(a,b)` adds the element `b` after `a` in the list, `remove(a)` removes the element `a` from the list, and `read()` returns the whole list. For simplicity, we assume that the elements are unique, an element is added or removed at most once, and the list always contains a sentinel element ∘.

```
1 var N := ∅, T := ∅;
2 var ts := (0, cid);

3 operation addAfter(a, b){
4   assume(a = ◦ ∨
5     a ≠ ◦ ∧ (_,_,a) ∈ N ∧ a ∉ T);
6   local i := (ts.fst+1, cid);
7   return;
8   gen_eff AddAft(a, i, b);
9 }

10 effector AddAft(a, i, b){
11   N := N ∪ {(a, i, b)};
12   if (ts < i) ts := i;
13 }

14 operation read(){
15   return trav(N,T);
16   gen_eff IdEff;
17 }

18 operation remove(a){
19   assume((_,_,a) ∈ N
20     ∧ a ∉ T ∧ a ≠ ◦);
21   return;
22   gen_eff Rmv(a);
23 }

24 effector Rmv(a){
25   T := T ∪ {a};
26 }
```

Figure 2. The Replicated Growable Array (RGA).

For CRDTs, each operation has *two phases*. In the *first* phase, a client on the node issues the operation. We call the node the *origin* of the operation. The origin node performs some initial local computation and responds to the client's request using the `return` command. It also generates an *effector* (see `gen_eff` in lines 8, 16 and 22), which captures the updates on the shared (replicated) state. The effector is executed immediately at the origin node, and is broadcast to all other nodes. In the *second* phase, each node applies the effector asynchronously over its local replica. Note that *read-only queries* (e.g., the `read()` operation) generate the identity effector `IdEff` (line 16 in Fig. 2). We do not need to broadcast `IdEff` since it does not change the state.

RGA represents the list using a time-stamped tree. Every tree node (a, i, b) consists of a key element b, a time-stamp i associated with b, and the key element a of its *parent node*. It is added by the operation `addAfter(a, b)`. Then a tree is encoded as a set of triples. For instance, the tree above can be represented by the set N:



$$\mathsf{N} = \{(\circ, \mathsf{ts}_0, \mathsf{a}), (\mathsf{a}, \mathsf{ts}_1, \mathsf{e}), (\mathsf{a}, \mathsf{ts}_2, \mathsf{b}), (\mathsf{a}, \mathsf{ts}_3, \mathsf{c}), (\mathsf{c}, \mathsf{ts}_4, \mathsf{d})\}$$

We assume ◦ is the root node of the tree. Besides the tree N, the algorithm also uses T as a *tombstone set* recording all the elements that are removed. Each replica state also contains `ts` to record the newest time-stamp at the replica.

The read-only *query* operation `read()` calls the function `trav`. It first orders the sibling nodes on the tree N in decreasing time-stamp order, and then traverses the tree by depth-first search. From the resulting list, all the elements in the tombstone set T are removed and the list consisting of the remaining elements is returned. For instance, suppose the tombstone set T for the tree N shown above is {e}. The `read()` should return acdb if $\mathsf{ts}_0 < \mathsf{ts}_1 < \mathsf{ts}_2 < \mathsf{ts}_3 < \mathsf{ts}_4$.

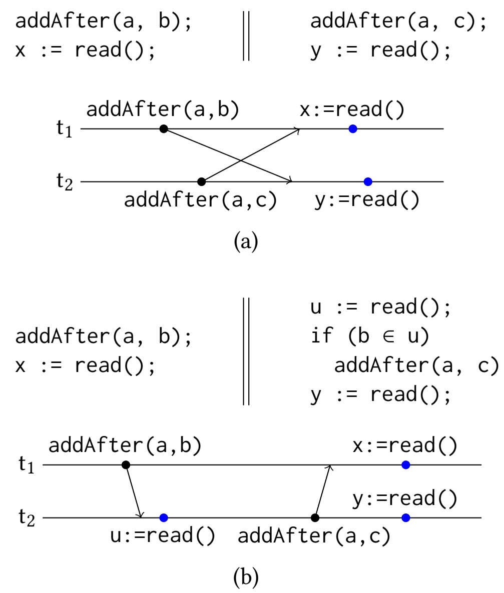


Figure 3. Clients of RGA and their executions.

The `addAfter(a,b)` operation generates the time-stamp `i` for b. Here time-stamps are implemented using pairs (n, t), where n is a natural number and t is a node ID (we write `cid` for the current node ID). Every two time-stamps are comparable: $(n_1, \mathsf{t}_1) > (n_2, \mathsf{t}_2)$ holds if $(n_1 > n_2)$ or $(n_1 = n_2) \wedge (\mathsf{t}_1 > \mathsf{t}_2)$. The effector of `addAfter(a,b)` simply adds `(a,i,b)` into the tree N and refreshes the time-stamp `ts` at the recipient node. The effector of `remove(a)` adds a into T.

Clients. The top of Fig. 3(a) shows a simple client program of RGA. It consists of two client threads calling the RGA operations. We represent the whole program as **let** Π_{RGA} **in** $C_1 \parallel C_2$, where Π_{RGA} denotes the RGA implementation in Fig. 2.

The bottom of Fig. 3(a) shows an execution of the program, assuming the clients running on two distinct nodes t_1 and t_2. The dots denote the client requests at the origin node (and the blue dots denote read-only queries). An arrow means sending an effector to a certain node.

We model an execution trace as a sequence $\mathcal{E}$ of events recording the execution of all the operations (both originals and effectors), and $\mathcal{E}|_{\mathsf{t}}$ as the subsequence consisting of only events occurring on the node t. So the execution shown in Fig. 3(a) is defined as the following trace $\mathcal{E}$ (assuming $\mathsf{ts}_1 < \mathsf{ts}_2$ and the initial list contains a only). We also record the arguments and return values (if any) of each operation.

$$\begin{array}{l}(\mathsf{t}_1, \mathtt{addAfter}(\mathsf{a}, \mathsf{b}), \mathsf{ts}_1), (\mathsf{t}_2, \mathtt{addAfter}(\mathsf{a}, \mathsf{c}), \mathsf{ts}_2), \\ (\mathsf{t}_2, \mathtt{AddAft}(\mathsf{a}, \mathsf{ts}_1, \mathsf{b})), (\mathsf{t}_1, \mathtt{AddAft}(\mathsf{a}, \mathsf{ts}_2, \mathsf{c})), \\ (\mathsf{t}_1, \mathtt{read}(), \mathsf{acb}), (\mathsf{t}_2, \mathtt{read}(), \mathsf{acb})\end{array}$$

The event $(\mathsf{t}_1, \mathtt{addAfter}(\mathsf{a}, \mathsf{b}), \mathsf{ts}_1)$ represents the invocation of an operation on the origin node t_1, where the time-stamp ts_1 is generated for the corresponding effector. The event

$(\mathsf{t}_2, \mathsf{AddAft(a, ts_1, b)})$ represents the execution of an effector on t_2 (sent from other nodes). Then the local traces $\mathcal{E}|_{\mathsf{t}_1}$ and $\mathcal{E}|_{\mathsf{t}_2}$ are the following:

$$(\mathsf{t}_1, \mathsf{addAfter(a,b)}, \mathsf{ts}_1), (\mathsf{t}_1, \mathsf{AddAft(a, ts_2, c)}), (\mathsf{t}_1, \mathsf{read()}, \mathsf{acb})$$

$$(\mathsf{t}_2, \mathsf{addAfter(a,c)}, \mathsf{ts}_2), (\mathsf{t}_2, \mathsf{AddAft(a, ts_1, b)}), (\mathsf{t}_2, \mathsf{read()}, \mathsf{acb})$$

Note that each node only sees its own read-only queries.

2.2 Functional Correctness (FC) of CRDTs

Correctness of CRDTs should capture both SEC and functionality of the data types, so that we can reason about the behaviors of clients (e.g., those in Fig. 3) without looking into the code of CRDT implementation (e.g., the RGA algorithm in Fig. 2), assuming the correctness of CRDT. It is easy to see that the RGA algorithm guarantees SEC since all the effectors produced by the algorithm are commutative with each other, but what is the expected functionality? From clients' point of view, the object is shared by all client threads and may be updated concurrently through the provided operations. Ideally we want to allow the client to maintain a simple *atomic* view of each object operation, so that we can interpret the client's behaviors in terms of executions of a sequence of these abstract atomic operations. For instance, the nodes t_1 and t_2 in Fig. 3(a) may both interpret their local execution traces as the following sequential execution of atomic operations:

$$\mathsf{addAfter\text{-}atom(a,b)}, \mathsf{addAfter\text{-}atom(a,c)}, (\mathsf{read()}, \mathsf{acb})$$

Here `addAfter-atom(x,y)` represents an abstract atomic specification of `addAfter(x,y)`. Its effects are applied atomically to the RGA object. It is abstract and does not generate any effectors or time-stamps. Note that the result `acb` of the final `read` determines the order between `addAfter-atom(a,b)` and `addAfter-atom(a,c)`. Therefore, for the node t_2, the abstract operations have to be executed in a different order from the order of the effectors in its concrete trace $\mathcal{E}|_{\mathsf{t}_2}$.

Unlike SEC, which is about the consistency of data replica on *different* nodes, the functional correctness (FC) is defined from the viewpoint of each *individual* node (or client). It specifies the consistency between the execution trace of concrete operations on a node and the corresponding abstract execution trace.

Defining FC. The above example shows that each node t may interpret an execution $\mathcal{E}$ in terms of a sequential execution of the corresponding atomic operations, which we describe by a total order ar_{t} over these operations. Our FC requires, for every prefix $\mathcal{E}'$ of $\mathcal{E}$, the sub-trace $\mathcal{E}'|_{\mathsf{t}}$ that t sees locally may correspond to an abstract trace $\mathcal{E}''$ following the total order ar_{t}, such that performing $\mathcal{E}'|_{\mathsf{t}}$ has the same effects as performing $\mathcal{E}''$, that is, *they generate the same state (modulo the state abstraction), and the same return value if* $\mathcal{E}'|_{\mathsf{t}}$ *ends with a query operation.*

In the example both ar_{t_1} and ar_{t_2} order `addAfter-atom(a,b)` before `addAfter-atom(a,c)`. For t_2, we consider its local traces of all the prefixes of $\mathcal{E}$:

$$\mathcal{E}_1 : (\mathsf{t}_2, \mathsf{addAfter(a,c)}, \mathsf{ts}_2)$$
$$\mathcal{E}_2 : (\mathsf{t}_2, \mathsf{addAfter(a,c)}, \mathsf{ts}_2), (\mathsf{t}_2, \mathsf{AddAft(a, ts_1, b)})$$
$$\mathcal{E}_3 : (\mathsf{t}_2, \mathsf{addAfter(a,c)}, \mathsf{ts}_2), (\mathsf{t}_2, \mathsf{AddAft(a, ts_1, b)}), (\mathsf{t}_2, \mathsf{read()}, \mathsf{acb})$$

We can check that $\mathcal{E}_1$ generates the same state as the atomic execution of `addAfter-atom(a,c)` (since the trace consists of only one event, it trivially satisfies the total order ar_{t_2}), and $\mathcal{E}_2$ corresponds to

$$\mathsf{addAfter\text{-}atom(a,b)},\ \mathsf{addAfter\text{-}atom(a,c)}$$

For $\mathcal{E}_3$, we also check the final return value is the same with such a query in the abstract trace.

2.3 Ordering of Operations and ACC

Both SEC and FC above are defined in a declarative manner and are not very informative to the clients of CRDTs. For instance, FC only requires the *existence* of an order ar_{t} on each node t to order the abstract operations, and says nothing about what the ar_{t} is like. So the clients still cannot tell the execution orders between CRDT operations.

To help reason about client programs, we want to specify the ordering of operations that CRDTs can enforce. More specifically, for each total order ar_{t} of abstract operations on each node t, we want to give more constraints to tell how to relate it to the concrete execution order, and how to relate different ar_{t} on different nodes so that SEC is guaranteed.

For instance, a direct mapping of each concrete step to the corresponding abstract atomic one following the real-time order on a node usually does not work. In the example shown in Fig. 3(a), ar_{t_2} has to order `addAfter-atom(a,b)` before `addAfter-atom(a,c)`, which is different from the real-time order of concrete operations in $\mathcal{E}|_{\mathsf{t}_2}$. *Then what are the appropriate orders of the abstract operations?*

Preserving the visibility order. Consider the client of RGA in Fig. 3(b). In the execution, the first `read` of t_2 is made after the arrival of the effector of `addAfter(a,b)` from t_1. In this case we say `addAfter(a,b)` is *visible* to `u:=read()`. In general, an operation *a* is visible to an operation *b* at the node t if the effector of *a* has been applied at t before t issues *b*. The visibility order encodes the "happens-before" relations between operations for a certain node.

Naturally we expect `u`, `x` and `y` to read out `ab`, `acb` and `acb` respectively (assuming the initial list contains `a` only). This means, when we map the concrete steps at a thread to a sequence of abstract atomic operations, the abstract executions should follow the visibility order.

Different nodes may observe different orders. In FC we require each node t to maintain an order ar_{t} of abstract operations. SEC would be obvious if all ar_{t} are the same. However, as we would see below, this requirement is overly restrictive and cannot be satisfied by some CRDTs.

```
addAfter(a, p);        ||   addAfter(c, e);
addAfter(c, d);        ||   addAfter(a, q);
u := read();           ||   v := read();
```

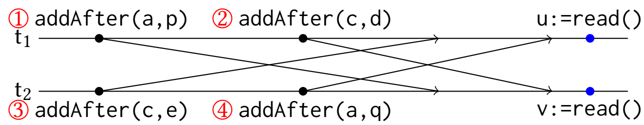


Figure 4. A client of continuous sequence. Assuming the initial sequence is ac, is it possible for u and v to read apqced?

Consider the program in Fig. 4. It is also a client of CRDT sequence, but implemented using the continuous sequence algorithm [20] instead of RGA. The continuous sequence tags each `addAfter` operation with a real number, the value of which reflects the intended position of the newly added element (assuming tags of elements on the sequence are in increasing order). For instance, assuming the initial sequence is ac, operation ① will tag p with a real number between the tags of a and its subsequent element c. The `read` operation then orders the elements by their tags and returns the resulting sequence. Note that the tags are different from the time stamps in RGA, and the happens-before order does *not* imply the order of tags. For instance, we know the tag generated by ② is greater than ①, but the tag of ④ is smaller than ③.

In this example it is possible to read apqced at the end, as long as the tag generated by ① happens to be smaller than that of ④, while the tag of ③ is smaller than that of ②. To interpret the final sequence apqced, node t_1 has to order the abstract operation ④ before ①, and order ② before ③. In addition, it needs to preserve the visibility order, as we explained before. So it needs to order ① before ②. Therefore, the only acceptable order for t_1 is ④①②③. Similarly, the only possible order for t_2 is ②③④①. So ① and ② (also ③ and ④) must be ordered differently by t_1 and t_2.

Therefore we should allow different nodes to have different local views of the abstract executions. In particular, *the visibility orders of operations originated in other nodes may not be respected.* We can also find similar examples in other CRDTs such as the add-wins set.

However, the orders cannot be arbitrarily different because we need to guarantee SEC. They have to be consistent in some way. *What kind of consistency should be enforced then?*

Conflicting operations should follow the same order. CRDTs achieve SEC by turning non-commutative abstract operations into commutative effectors. Arbitrary orderings of commutative operations always lead to the same state.

We say two abstract operations f_1 and f_2 are *conflicting*, represented as $f_1 \bowtie f_2$, if they are not commutative. In Fig. 4, `addAfter(a, p)` and `addAfter(a, q)` are conflicting, but `addAfter(a, p)` and `addAfter(c, d)` are not.

Naturally, to reach the same state, we require the abstract executions on different nodes execute conflicting operations in the same order. In Fig. 4, the abstract executions ④①②③ and ②③④① order ④ and ① (② and ③) the same way.

Abstract Converging Consistency (ACC). We formalize our correctness notion of CRDTs as Abstract Converging Consistency (ACC), which is a relation between the concrete implementation of a CRDT (represented as Π) and its abstract specification (represented as a pair $(\Gamma, \bowtie)$, where Γ is the abstract atomic specification of the operations, and $\bowtie$ is a symmetric binary relation between conflicting operations).

ACC requires FC defined in Sec. 2.2, and the order constraints over abstract executions described in this section. More specifically, $\mathrm{ACC}(\Pi, (\Gamma, \bowtie))$ requires that, for any execution trace $\mathcal{E}$ of Π, each node t can find a total order ar_{t} over abstract atomic operations in Γ, such that:

- For each prefix $\mathcal{E}'$, there is a corresponding sequence $\mathcal{E}''$ of abstract operations. $\mathcal{E}''$ follows the order ar_{t} and generates the same effects with $\mathcal{E}'|_{\mathrm{t}}$;
- ar_{t} preserves the local visibility order on t; and
- For any two nodes t_1 and t_2, ar_{t_1} and ar_{t_2} can be different, but they must assign the same order for conflicting operations specified in $\bowtie$.

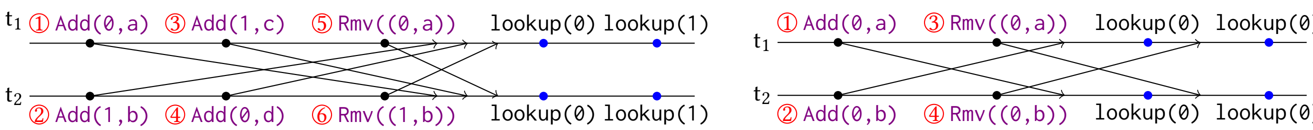
We can prove that ACC defined above guarantees SEC.

Note that the last point only requires the *existence* of a consistent ordering of conflicting operations, with no further constraints. This is not a problem for UCR-CRDTs that use uniform operation-independent conflict resolving strategies. However, for CRDTs like add-wins and remove-wins sets, we may rely on the specific strategy (X-wins) to reason about the behaviors of clients. In this case we need to further refine the above ACC definition.

2.4 Extended ACC for X-Wins CRDTs

We show an execution of add-wins sets in Fig. 5(a). A set provides three operations: `lookup(e)`, `add(e)` and `remove(e)`. The add-wins set algorithm assigns a unique tag to each element when it is added. In Fig. 5 we highlight the tags by labeling the dots with effectors rather than originals. We use 0 and 1 to represent the elements in the set, and a and b for the tags. So an element may be added to the set multiple times but each time with a different tag. The `remove` operation removes all the occurrences of the element in the local replica. The effector of `remove` carries the set of element-tag pairs removed locally. On receiving the effector, the remote hosts remove only these pairs from their local replicas.

For instance, in Fig. 5(a) when t_2 issues a `remove(1)` request (operation ⑥), it sees only `(1, b)` in the local replica and sends the effector `Rmv((1, b))` to t_1. When it arrives at t_1, the pairs `(1, b)` and `(1, c)` are both in t_1's replica, but only `(1, b)` is removed. Therefore the subsequent `lookup(1)`



(a) (b)

Figure 5. Executions of the add-wins set.

still returns true. This illustrates the *add-wins* conflict resolving strategy: for concurrent `add` (③) and `remove` (⑥), the abstract view is to execute `add` after `remove`.

It is interesting to see that the *add-wins* conflict resolving strategy is different from the time-stamp-based approaches since it is tied with the functionality of specific operations. As the dual, there is also the remove-wins set algorithm which applies the *remove-wins* strategy. Note that the add-wins set and the remove-wins set assume *causal delivery* between `add` and `remove` operations. This is also different from other CRDTs, which do not need to rely on causal delivery.

The add-wins sets and remove-wins sets may have different behaviors, which are observable by clients. If the client relies on the specific strategy and cares about the difference, our above ACC definition would be too abstract to distinguish them. We solve this problem by introducing a *won-by* relation $\blacktriangleleft$ in the abstract specification to describe the conflict resolving strategy. We have `remove(e)` $\blacktriangleleft$ `add(e)` for add-wins set, and the reverse for remove-wins set. Since we only need to resolve conflicts for conflicting operations, the $\blacktriangleleft$ relation is a subset of the conflict relation $\bowtie$. Correspondingly, we refine the third point of ACC in 2.3 with an extra requirement that all the ar_{t} respect the $\blacktriangleleft$ order.

Unfortunately, this simple extension of ACC would not work. Consider the execution shown in Fig. 5(b). For each node, we can see the two `lookup` operations return `true` and `false` respectively. However, we cannot find a total order ar satisfying ACC. For t_1, we have to order ① before ③ (to preserve the visibility order), and ③ before ② (to respect the $\blacktriangleleft$ order). Therefore ④ has to be the last operation, otherwise the abstract execution cannot generate the same return values as the concrete one, failing FC. However, ordering ④ after the concurrent ① would violate the $\blacktriangleleft$ order.

This problem is caused by our over-simplified interpretation of the "add-wins" conflict-resolving strategy, which says we should *always* order `remove(e)` before `add(e)` if they are *concurrent*. However, in our example, when ④ arrives at t_1, the effect of ① has already been canceled out by ③. Therefore at this moment whether ① has been executed before or not should make no difference.

To address this problem, we give a more precise description of the strategy, which says concurrent `remove(e)` should be ordered before `add(e)` only if the effect of `add(e)` is still reflected in the state (i.e., its effect has not been canceled out by others). Since the cancellation of effects is functionality dependent, we introduce another *canceled-by* relation $\triangleright$ over abstract operations in the specification. Informally, we let the operation f be canceled by f' ($f \triangleright f'$) if the following two requirements hold:

- f may win others as specified in $\blacktriangleleft$; and
- for any other abstract operations $f_1, \ldots, f_n$ ($n \geq 0$) in between, the abstract operation sequence $f, f_1, \ldots, f_n, f'$ has the same effects as $f_1, \ldots, f_n, f'$.

Therefore, for add-wins sets, we have `add(e)` $\triangleright$ `remove(e)` but not the inverse (which violates the first requirement).

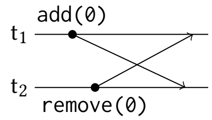
We relax the third point of ACC accordingly, and ignore the canceled operations when we check the consistency between the total orders ar_{t} for different nodes t. This relaxed ACC allows the total orders ar_{t_1} and ar_{t_2} in Fig. 5(b) to be defined as ①③②④ and ②④①③, respectively. When ④ is executed at t_1, we only need to check that ③ and ④ are ordered consistently, and ignore ① and ② because they have been canceled (by ③ and ④ respectively) at this moment. Also because ③ and ④ are not conflicting (they are commutative), it is okay to order them differently in ar_{t_1} and ar_{t_2}.

With the more refined specification, we can redefine the correctness as $\mathsf{XACC}(\Pi, (\Gamma, \bowtie, \blacktriangleleft, \triangleright))$. It also assumes causal delivery of messages, as required by add-wins and remove-wins sets. Note that UCR-CRDTs satisfying $\mathsf{ACC}(\Pi, (\Gamma, \bowtie))$ in Sec. 2.3 also satisfy $\mathsf{XACC}(\Pi, (\Gamma, \bowtie, \emptyset, \emptyset))$ — Since their conflict resolving policies are not tied with particular operations, we can simply set $\blacktriangleleft$ and $\triangleright$ to be empty.

Compositionality. Like linearizability, our definition of ACC/XACC is compositional. That is, for a set of CRDTs $\Pi_1, \ldots, \Pi_n$, if every Π_i satisfies $\mathsf{XACC}(\Pi_i, (\Gamma_i, \bowtie_i, \blacktriangleleft_i, \triangleright_i))$, then the clients can use them together and view them as a single big object satisfying $\mathsf{XACC}(\overrightarrow{\Pi}, (\overrightarrow{\Gamma}, \overrightarrow{\bowtie}, \overrightarrow{\blacktriangleleft}, \overrightarrow{\triangleright}))$, where $\overrightarrow{\Pi}$ represents the disjoint union of all the operations $\Pi_1 \uplus \ldots \uplus \Pi_n$, and $\overrightarrow{\Gamma}$, $\overrightarrow{\bowtie}$, $\overrightarrow{\blacktriangleleft}$ and $\overrightarrow{\triangleright}$ are defined similarly. Note here we assume the CRDTs do not share data.

2.5 Abstraction and Client Reasoning

It is important to note that the goal of this work is *not* to give axiomatic definitions to tell the validity of a single execution trace, although we use traces above (e.g., those shown in Figs. 3 and 4) to explain the key ideas. Our goal is to support *static program verification*, where we need to consider all the execution traces that can be possibly generated by the program, and the reasoning is based on the program text without actually running it. This is much more challenging than reasoning about a single trace.



For instance, if we look at the execution of a CRDT set on the right, it is easy to tell what the final state is: it must contain 0 for add-wins sets, but mustn't for remove-wins sets. Knowing the concrete implementation mechanism, the result can be easily predicted. The deceiving simplicity may make one doubt the need of abstraction. However, if we consider the simple client program (`add(0); || remove(0);`) that generates the trace, we know it may generate both results (since there are other possible executions where one operation happens before the other), no matter which CRDT set we use[1]. This example shows that we have to consider all possible ordering of operations for program reasoning, which can be very complicated in non-trivial clients. Abstracting away the implementation details and taking an atomic view of operations can greatly simplify the reasoning.

Remark. Picking the appropriate abstraction level for CRDT specifications is one of the key challenges we need to address. On the one hand, the abstractions need to hide as much implementation detail as possible. On the other hand, they need to be useful for client reasoning, i.e., it does not abstract away important functionality properties of the data type.

For X-wins CRDTs, we need to decide whether or not to hide the functionality-dependent "X-wins" strategies. It might be possible to have a weaker ACC definition that unifies UCR and X-Wins CRDTs, but it would not support the reasoning about some special clients whose functionality depends on the differences between add-wins sets, remove-wins sets and UCR sets. Consider the following client:

```
add(0);remove(0);    ||  add(0);remove(0);
x := read();         ||  y := read();
```

At the end the post-condition $0 \in x \Rightarrow 0 \notin y$ holds when the client uses the remove-wins set or UCR sets (e.g., the LWW-element set) but *not* when it uses the add-wins set. Abstracting away the differences of these sets would prevent the verification of the above program.

3 Basic Technical Settings

Figure 6 shows the syntax of the language. The whole program P consists of n clients C, each running on different nodes. They share the *object* Π, which is replicated on all the nodes. Each client executes sequentially, accessing the local *client* state in the node. It can also access the *object* state through the command $x := f(E)$, which calls the operation f of the object with the argument E.

We model the object Π as a mapping from an operation name f and its argument to the actual operation over the object state. When a client calls an operation, it executes in two steps. First the operation is applied over the object state and generates a return value and an effector δ. The

[1]Note it is indeed possible to construct clients that can distinguish add-wins sets from remove-wins sets, as discussed in the following remarks.

$$
\begin{array}{rrcl}
(\mathit{OpName}) & f & \in & \mathit{String} \\
(\mathit{Effector}) & \delta & \in & \mathit{LocalState} \rightharpoonup \mathit{LocalState} \\
(\mathit{ODecl}) & \Pi & \in & \mathit{OpName} \times \mathit{Val} \rightharpoonup \mathit{LocalState} \rightharpoonup \mathit{Val} \times \mathit{Effector} \\
(\mathit{Expr}) & E & ::= & x \mid n \mid E{+}E \mid \ldots \\
(\mathit{CltStmt}) & C & ::= & x := f(E) \mid \textbf{skip} \mid C;C \mid \textbf{if}\ (E)\ C\ \textbf{else}\ C \mid \ldots \\
(\mathit{Prog}) & P & ::= & \textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n
\end{array}
$$

Figure 6. Syntax of the programming language.

effector δ captures the operation's effect over the object state. It is *broadcast* to all nodes, including the one where the client request originates. Then the effector δ is applied on the local replica of the object data on each node. Note that on the origin node of the client request, the generation of the effector and the execution of it over the local replica are done atomically. To simplify the presentation we assume each program uses only one object. As we explained in Sec. 2.4, our correctness definition ACC is compositional and the results still hold when there are more objects.

We assume an effector is delivered to a node at most once, but it may never reach a target node. Also we do *not* assume FIFO message channels. Most of the CRDTs can work under these assumptions. When stronger assumptions are needed (e.g., causal delivery), we can add extra constraints over execution traces.

Events and event traces. The clients C_i in the program $\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n$ are executed following the standard interleaving semantics. The semantics generates events when CRDT operations are executed. An execution trace is the sequence of events generated during the interleaving execution. We define the events e and execution traces $\mathcal{E}$ below:

$$
\begin{array}{rrcl}
(\mathit{Event}) & e & ::= & (\mathit{mid}, \mathrm{t}, (f, n, n', \delta)) \mid (\mathit{mid}, \mathrm{t}, (f, n), \delta) \\
(\mathit{ETrace}) & \mathcal{E} & ::= & \epsilon \mid e :: \mathcal{E}
\end{array}
$$

Here ϵ represents an empty list. The event $(\mathit{mid}, \mathrm{t}, (f, n, n', \delta))$ is called an *origin event*. It is generated when the object operation f is called on the node t with the argument n, and the return value n' and the effector δ are generated by applying $\Pi(f, n)$ over the local replica. It also contains a unique ID *mid* for the original request of the operation. When the effector δ is delivered to and executed at another node t', the node t' generates the event $(\mathit{mid}, \mathrm{t}', (f, n), \delta)$. It records not only the local node ID t' and the effector, but also the information about the original operation, including the operation name f, the argument n, and the ID *mid*.

We define $\mathcal{T}(P, \mathcal{S})$ as the *prefix closure* of the event traces that can be generated by executing P from the initial state $\mathcal{S}$. We also define $\mathcal{T}(\Pi, \mathcal{S})$ as the prefix closure of the event traces that can be generated by *any* set of clients accessing Π with the initial state $\mathcal{S}$.

4 Specifications for CRDTs

The specification of a CRDT object consists of two parts, the operation specification Γ and the conflict relation $\bowtie$, as shown in Fig. 7. Γ maps operation names and arguments

$$
\begin{array}{rl}
(\mathit{OSpec}) & \Gamma \in \mathit{OpName} \times \mathit{Val} \rightharpoonup \mathit{AbsState} \rightarrow \mathit{Val} \times \mathit{AbsState} \\
(\mathit{Action}) & \alpha \in \mathit{AbsState} \rightarrow \mathit{AbsState} \\
& \bowtie \in \mathscr{P}(\mathit{Action} \times \mathit{Action})
\end{array}
$$

Figure 7. Object specifications $(\Gamma, \bowtie)$.

to *abstract atomic operations* of the type $\mathit{AbsState} \rightarrow \mathit{Val} \times \mathit{AbsState}$. That is, each atomic operation applies over an abstract object state and generates the resulting abstract state and a return value. We assume it is a total function because as a specification we do not want it to get stuck whenever a client applies the operation.

We use AbsState to represent the set of object states $\mathcal{S}$ at the abstract level. They may abstract away the implementation dependent information of the concrete states. For instance, the concrete state of RGA consists of a time-stamped tree N and a tombstone T, as shown in Sec. 2.1, while the abstract state is simply a sequence (e.g., acdb).

Since the return value of an operation is meaningful only to the origin node, while the state transformation needs to be performed on all replicas, we use $\mathrm{opr}(\Gamma(f, n))$ to represent the effects of $\Gamma(f, n)$, which does a state transformation. We call the transformation an action (represented as α).

The conflict relation $\bowtie$ needs to be a *symmetric* binary relation over *non-commutative* actions. For sets, add(x) and remove(x) conflict with each other. For RGA,

$$
\begin{array}{l}
\mathtt{addAfter}(a, b) \bowtie \mathtt{addAfter}(c, d) \ \text{ iff } \ \{a, b\} \cap \{c, d\} \neq \emptyset, \\
\mathtt{addAfter}(a, b) \bowtie \mathtt{remove}(c) \ \text{ iff } \ c \in \{a, b\}\,.
\end{array}
$$

Well-defined specifications must satisfy $\mathsf{nonComm}(\Gamma, \bowtie)$, which requires that all the non-commutative actions in Γ should be specified in $\bowtie$.

Definition 1. $\mathsf{nonComm}(\Gamma, \bowtie)$ iff $\forall f_1, n_1, f_2, n_2, \alpha_1, \alpha_2,$

$$
\begin{array}{l}
\alpha_1 = \mathsf{opr}(\Gamma(f_1, n_1)) \wedge \alpha_2 = \mathsf{opr}(\Gamma(f_2, n_2)) \wedge \neg(\alpha_1 \bowtie \alpha_2) \\
\implies \alpha_1 \,;\, \alpha_2 = \alpha_2 \,;\, \alpha_1
\end{array}
$$

where $\alpha \,;\, \alpha' \stackrel{\text{def}}{=} \lambda \mathcal{S}.\, \alpha'(\alpha(\mathcal{S}))$.

As we explained in Sec. 2.5, add-wins and remove-wins sets should be specified with further information about the conflicting resolving strategies, i.e., the won-by ($\blacktriangleleft$) and canceled-by ($\triangleright$) relations over conflicting actions. In the following sections we first present our results for UCR-CRDTs that do not need $\blacktriangleleft$ and $\triangleright$, and show the extension of them to support these X-wins algorithms in Sec. 9.

We assume $\bowtie$ is symmetric and $\mathsf{nonComm}(\Gamma, \bowtie)$ holds throughout the paper. We overload $\bowtie$ over operations, and also over events, written as $(f, n) \bowtie_\Gamma (f', n')$ and $e \bowtie_\Gamma e'$ respectively (the subscript Γ is used to extract actions corresponding to (f, n), (f', n'), e and e').

5 Abstract Converging Consistency

As shown in Def. 2, $\mathsf{ACC}_\varphi(\Pi, (\Gamma, \bowtie))$ is parameterized with an abstraction function φ, which maps concrete object states to abstract ones, i.e., $\varphi \in \mathit{LocalState} \rightharpoonup \mathit{AbsState}$.

$$
\begin{array}{l}
\mathsf{ExecRelated}_\varphi(\mathsf{t}, (\mathcal{E}, \mathcal{S}), (\Gamma, ar)) \ \text{ iff } \ \forall \mathcal{E}' \leqslant \mathcal{E}. \\
\quad \forall (\mathcal{S}'_a, n') = \mathsf{aexec}(\Gamma, \varphi(\mathcal{S}), \mathsf{visible}(\mathcal{E}', \mathsf{t}) \downharpoonright ar). \\
\qquad \varphi(\mathsf{exec\_st}(\mathcal{S}, \mathcal{E}'|_\mathsf{t})) = \mathcal{S}'_a \wedge \\
\qquad\quad (\forall e = \mathsf{last}(\mathcal{E}'|_\mathsf{t}).\, \mathsf{is\_orig}_\mathsf{t}(e) \implies \mathsf{rval}(e) = n') \\
\mathsf{Coh}(ar, ar', (\Gamma, \bowtie)) \ \text{ iff } \\
\quad \forall e_1, e_2.\, (e_1 \ ar \ e_2) \wedge (e_2 \ ar' \ e_1) \implies \neg(e_1 \bowtie_\Gamma e_2)
\end{array}
$$

Figure 8. Auxiliary definitions for ACC.

Definition 2. $\mathsf{ACC}_\varphi(\Pi, (\Gamma, \bowtie))$ iff

$$
\forall \mathcal{S}, \mathcal{E}.\ \mathcal{E} \in \mathcal{T}(\Pi, \mathcal{S}) \wedge \mathcal{S} \in \mathit{dom}(\varphi) \implies \mathsf{ACT}_\varphi(\mathcal{E}, \mathcal{S}, (\Gamma, \bowtie))
$$

It requires every event trace $\mathcal{E}$ of Π to satisfy ACT shown in Def. 3, which formalizes the idea in Sec. 2.3.

Definition 3. $\mathsf{ACT}_\varphi(\mathcal{E}, \mathcal{S}, (\Gamma, \bowtie))$ iff $\exists ar_1, \ldots, ar_n,$

$$
\begin{array}{l}
\forall \mathsf{t}.\ \mathsf{totalOrder}_{\mathsf{visible}(\mathcal{E}, \mathsf{t})}(ar_\mathsf{t}) \wedge (\xrightarrow[\mathsf{t}]{\mathsf{vis}}_{\mathcal{E}} \subseteq ar_\mathsf{t}) \wedge \\
\mathsf{ExecRelated}_\varphi(\mathsf{t}, (\mathcal{E}, \mathcal{S}), (\Gamma, ar_\mathsf{t})) \wedge \forall \mathsf{t}' \neq \mathsf{t}.\ \mathsf{Coh}(ar_\mathsf{t}, ar_{\mathsf{t}'}, (\Gamma, \bowtie))
\end{array}
$$

where we define ExecRelated and Coh in Fig. 8.

Before explaining ACT, we first introduce the notations for visibility of events. In the execution $\mathcal{E}$ an origin event e is *visible to* another event e' originated from the node t (i.e., $e \xrightarrow[\mathsf{t}]{\mathsf{vis}}_{\mathcal{E}} e'$), if the effector of e has reached t before e' is issued. We also use $\mathsf{visible}(\mathcal{E}, \mathsf{t})$ to represent the set of origin events whose effectors have reached t.

ACT says that each node t may have its own arbitration order ar_t, which is a total order over the origin events on $\mathcal{E}$ visible to t. Each ar_t must preserve the visibility order on t (i.e., $\xrightarrow[\mathsf{t}]{\mathsf{vis}}_{\mathcal{E}} \subseteq ar_\mathsf{t}$).

On functional correctness, ACT requires that the concrete execution on node t should correspond to the execution of the abstract events following the arbitration order ar_t (see $\mathsf{ExecRelated}_\varphi(\mathsf{t}, (\mathcal{E}, \mathcal{S}), (\Gamma, ar_\mathsf{t}))$). As defined in Fig. 8, ExecRelated says that every state in t's concrete execution can be mapped (via φ) to the state in the abstract execution trace, and that every request issued by t gets the same return value as the abstract one. The definition checks on every prefix $\mathcal{E}'$ of the concrete trace $\mathcal{E}$. We use $\mathsf{visible}(\mathcal{E}', \mathsf{t}) \downharpoonright ar$ to represent a serialization of the set $\mathsf{visible}(\mathcal{E}', \mathsf{t})$ following the total order ar. Then $\mathsf{aexec}(\Gamma, \mathcal{S}_a, \mathcal{E})$ executes the sequence of abstract operations on $\mathcal{E}$, starting from the initial abstract state $\mathcal{S}_a$. It returns the final state $\mathcal{S}'_a$ and the return value n' of the last operation. Similarly, we use $\mathsf{exec\_st}(\mathcal{S}, \mathcal{E})$ to represent the final state generated by executing the effectors on $\mathcal{E}$ from the initial state $\mathcal{S}$. We omit their definitions here.

The arbitration orders on different nodes can be different, but must be coherent to guarantee SEC. The coherence requires that conflicting actions are given the same arbitration order by all the nodes (see $\mathsf{Coh}(ar_\mathsf{t}, ar_{\mathsf{t}'}, (\Gamma, \bowtie))$), as defined in

Fig. 8). Combined with ($\xrightarrow[\mathsf{t}]{\mathsf{vis}}_{\mathcal{E}} \subseteq ar_{\mathsf{t}}$) for every t, Coh actually ensures that ar_{t} must agree with other nodes' visibility orders on *conflicting* operations.

Properties of ACC. Our ACC guarantees SEC. Below we first define the convergence of event traces in Def. 4. It is a property about the concrete level execution only, and it captures the SEC requirement.

Definition 4. $\mathsf{CvT}_{\varphi}(\mathcal{E}, \mathcal{S})$ iff

$$\begin{aligned}&\forall \mathcal{E}', \mathcal{E}'', \mathsf{t}, \mathsf{t}'.\ \mathcal{E}' \leqslant \mathcal{E} \wedge \mathcal{E}'' \leqslant \mathcal{E} \wedge \mathsf{visible}(\mathcal{E}', \mathsf{t}) = \mathsf{visible}(\mathcal{E}'', \mathsf{t}')\\ &\qquad \Longrightarrow \varphi(\mathsf{exec\_st}(\mathcal{S}, \mathcal{E}'|_{\mathsf{t}})) = \varphi(\mathsf{exec\_st}(\mathcal{S}, \mathcal{E}''|_{\mathsf{t}'}))\end{aligned}$$

$\mathsf{CvT}_{\varphi}(\mathcal{E}, \mathcal{S})$ says, whenever the two nodes t and t′ see the same set of operations, executing the corresponding sub-traces on t and t′ results in states corresponding to the same abstract state. Note we allow t and t′ to pick different time points in the execution trace $\mathcal{E}$ (see $\mathcal{E}' \leqslant \mathcal{E}$ and $\mathcal{E}'' \leqslant \mathcal{E}$, which says $\mathcal{E}'$ and $\mathcal{E}''$ can be different prefixes of $\mathcal{E}$), because there is no global time on the nodes. Besides, the two resulting states do not have to be identical. Instead, they only need to be mapped to the same abstract state. This way we allow the implementation-dependent data in the concrete states to be different. The convergence of an object Π, written as $\mathsf{Cv}_{\varphi}(\Pi)$, requires every event trace $\mathcal{E}$ of Π to satisfy CvT.

Lemma 5. If $\mathrm{ACC}_{\varphi}(\Pi, (\Gamma, \bowtie))$, then $\mathsf{Cv}_{\varphi}(\Pi)$.

Another important property of $\mathrm{ACC}_{\varphi}(\Pi, (\Gamma, \bowtie))$ is its *compositionality*, as we explained in Sec. 2.4.

6 Abstraction Theorem

To simplify the reasoning of clients of CRDTs, we give an *abstract operational semantics* of client programs, based on the abstract specification $(\Gamma, \bowtie)$. The abstract version of the client program is defined below:

$$(AProg)\ \mathbb{P} ::= \textbf{with } (\Gamma, \bowtie) \textbf{ do } C_1 \parallel \ldots \parallel C_n$$

It is safe to reason about clients at the abstract level as long as the CRDT implementation Π contextually refines $(\Gamma, \bowtie)$.

Definition 6. $\Pi \sqsubseteq_{\varphi} (\Gamma, \bowtie)$ iff, for all clients $C_1, \ldots, C_n$ and state $\mathcal{S} \in dom(\varphi)$, for all $\lfloor \mathcal{E} \rfloor$ and σ_c,

$$\begin{aligned}&(\lfloor \mathcal{E} \rfloor, \sigma_c) \in \mathcal{T}_{\mathbf{s}}(\textbf{let } \Pi \textbf{ in } C_1 \parallel \ldots \parallel C_n, \mathcal{S}) \Longrightarrow\\ &\qquad (\mathsf{obsv}_{\varphi}(\lfloor \mathcal{E} \rfloor), \sigma_c) \in \mathcal{T}_{\mathbf{s}}(\textbf{with } (\Gamma, \bowtie) \textbf{ do } C_1 \parallel \ldots \parallel C_n, \varphi(\mathcal{S}))\end{aligned}$$

Informally, $\Pi \sqsubseteq_{\varphi} (\Gamma, \bowtie)$ says, for any clients and initial states, executing the clients with Π does not generate more *observable behaviors* than the execution using $(\Gamma, \bowtie)$ in the abstract operational semantics (presented below). $\mathcal{T}_{\mathbf{s}}(P, \mathcal{S})$ and $\mathcal{T}_{\mathbf{s}}(\mathbb{P}, \mathcal{S})$ are defined similarly as $\mathcal{T}(P, \mathcal{S})$ (Sec. 3), but they additionally record the final client state σ_c. Also in the extended trace $\lfloor \mathcal{E} \rfloor$ they record all the intermediate *object* states together with the events. The function $\mathsf{obsv}_{\varphi}(\lfloor \mathcal{E} \rfloor)$ maps the extended trace $\lfloor \mathcal{E} \rfloor$ in the concrete semantics to an abstract trace. Each concrete event is mapped to an abstract one, and every recorded object state is mapped through the state abstraction function φ to an abstract object state.

Theorem 7 (Abstraction Theorem)**.**
$\mathrm{ACC}_{\varphi}(\Pi, (\Gamma, \bowtie)) \iff \Pi \sqsubseteq_{\varphi} (\Gamma, \bowtie)$.

Abstract operational semantics describes the execution of programs in the form of **with** $(\Gamma, \bowtie)$ **do** $C_1 \parallel \ldots \parallel C_n$. Clients are executed following the interleaving semantics. On each node, we always keep the initial object state $\mathcal{S}_0$. We also maintain a sequence ξ_{t} of the abstract operations that the node t has received. We can view ξ_{t} as a *runtime representation* of the arbitration order ar_{t} used in ACC. Given $\mathcal{S}_0$ and ξ_{t}, we can always generate the current object state on the fly by executing all the operations on ξ_{t} from $\mathcal{S}_0$.

When a node issues an operation, it puts the operation at the very end of its local ξ to get a new sequence ξ'. This reflects the preservation of the visibility order, as required in ACC, because at this moment the node has seen all the operations on ξ and therefore they all need to be ordered before the new operation. We also start from $\mathcal{S}_0$ and execute all the operations on ξ' to get the return value of the last operation. The node then broadcasts the operation itself (instead of effectors) to all the other nodes.

When a node receives an operation sent from others, it can non-deterministically insert the operation into any position of the local sequence ξ, as long as the resulting ξ' is *coherent* with every other ξ_{t} on node t. The coherence requirement is similar to $\mathsf{Coh}(ar_{\mathsf{t}}, ar_{\mathsf{t}'}, (\Gamma, \bowtie))$ defined in Fig. 8. It requires that conflicting operations follow the same order in all sequences (ξ' and all the other ξ_{t}). If we cannot find an insertion position in the local ξ so that the resulting ξ' satisfies the coherence requirement, the execution gets stuck. The semantics of the program can be viewed as the set of the stuck-free executions.

Since the operation lists ξ on all nodes must be coherent during the execution, we can prove that the abstract semantics inherently guarantees the convergence of the abstract object states. Then, the contextual refinement $\Pi \sqsubseteq_{\varphi} (\Gamma, \bowtie)$ can ensure $\mathsf{Cv}_{\varphi}(\Pi)$, the convergence of the concrete object. With the Abstraction Theorem (Thm 7), we can derive Lem. 5 again: $\mathrm{ACC}_{\varphi}(\Pi, (\Gamma, \bowtie))$ can ensure $\mathsf{Cv}_{\varphi}(\Pi)$ too.

7 Program Logic for Client Verification

To reason about clients using a CRDT object Π, we apply the Abstraction Theorem, and verify the clients using the more abstract object specifications $(\Gamma, \bowtie)$ instead.

We design a Hoare-style program logic to verify *functional correctness* of client programs, specified in the form of pre- and post-conditions. The top level judgment is in the form of $\vdash \{\mathcal{P}\}\textbf{with } (\Gamma, \bowtie) \textbf{ do } C_1 \parallel \ldots \parallel C_n\{\mathcal{Q}\}$, where $\mathcal{P}$ and $\mathcal{Q}$ are traditional Hoare-logic state assertions over both client and object states. To enable thread-local reasoning, we borrow ideas from shared-memory concurrency verification and base our logic on rely-guarantee reasoning [11]. Each C_{t} is verified in the form of $R, G; \Gamma, \bowtie \vdash_{\mathsf{t}} \{p\}C_{\mathsf{t}}\{q\}$, where R and G

$$\{\mathtt{s} = \mathtt{a}\}$$
```
              ‖ u := read();          ‖ v := read();
addAfter(a, b); ‖ if (b ∈ u)          ‖ if (c ∈ v)
              ‖   addAfter(a, c);     ‖   addAfter(c, d);
              ‖ x := read();          ‖ y := read();
```
$$\{\mathtt{d} \in \mathtt{x} \Rightarrow (\mathtt{s} = \mathtt{x} = \mathtt{acdb}) \wedge (\mathtt{y} = \mathtt{x} \vee \mathtt{y} = \mathtt{acd})\}$$

Figure 9. Correctness of a client program of RGA.

are rely and guarantee assertions, specifying the interactions between the current thread t and its environment threads.

The key challenge for the logic is to deal with the *weak* behaviors produced by the abstract semantics in Sec. 6, where client threads can reorder actions, which is reminiscent of weak memory models of languages like C11.

A motivating example. Figure 9 shows a client program of RGA and its specification. The precondition says the initial list s is a. The postcondition shows that x and y must be equal, if all the operations have been applied before the reads. It also tells which values x and y may read. Since we do *not* assume causal delivery, when the thread t_3 receives addAfter(a,c) from the thread t_2, it may *not* have received addAfter(a,b) from the thread t_1, though addAfter(a,c) is issued only after t_2 receives addAfter(a,b). As a result, it is possible that y reads acd. But, when t_3 finally receives addAfter(a,b), it must insert addAfter(a,b) *before* addAfter(a,c) (in the abstract semantics) to restore the causality (required by the coherence check). It is impossible for y to read abcd.

Assertions. It seems difficult to use traditional state assertions to express the insertion of an action into the past execution. Our idea is to introduce action assertions. We extend the syntax of Hoare logic assertions, p, with several new assertion forms, to specify the set of actions (originate from either the current thread t_c or its environment) and their orders of which t_c has knowledge at each program point. Figure 10 gives the syntax of our assertion language.

The assertions $[\alpha]^i_t$ and $\boxed{\alpha}^i_t$ describe singleton action sets containing only the action α. The former says the action α (with ID i) has been issued from its origin t, but we do not care whether it's on the way or it has arrived at the current node, while the latter says the current node has received α. We may omit the superscript action ID in an assertion when it is clear from the context what the action denotes. For the motivating example of Fig. 9, after t_3 succeeds in the check c ∈ v, its assertion must contain $\boxed{\mathtt{addAfter(a,c)}}_{t_2}$, but only $[\mathtt{addAfter(a,b)}]_{t_1}$.

We write emp for an empty action set. The assertion $p \sqcup q$ allows us to merge two action sets without enforcing new ordering. It can be used to describe non-conflicting actions. For instance, $[\mathtt{addAfter(a,b)}]_{t_1} \sqcup \boxed{\mathtt{remove(e)}}_{t_2}$ says addAfter(a,b) and remove(e) can be ordered either way. It can also describe a set of conflicting but concurrently

$$
\begin{array}{lrl}
(\mathit{StateAssn}) & \mathcal{P},\mathcal{Q} ::= & B \mid \neg\mathcal{P} \mid \mathcal{P}\wedge\mathcal{Q} \mid \mathcal{P}\vee\mathcal{Q} \mid \ldots \\
(\mathit{Assn}) & p,q ::= & \mathcal{P} \mid \mathsf{emp} \mid [\alpha]^i_t \mid \boxed{\alpha}^i_t \mid p\sqcup q \mid p \ltimes [\alpha]^i_t \\
& \mid & p \ltimes \boxed{\alpha}^i_t \mid (p,\bowtie) \ltimes [\alpha]^i_t \mid (p,\bowtie) \ltimes \boxed{\alpha}^i_t \\
& \mid & p \Rrightarrow q \mid \neg p \mid p\wedge q \mid p \vee q \mid p \Rightarrow q \mid \ldots \\
(\mathit{RGAssn}) & R,G ::= & \mathsf{Emp} \mid p \leadsto [\alpha]^i_t \mid R\vee R \mid R \Rightarrow R \mid \ldots
\end{array}
$$

Figure 10. Syntax of the assertion language.

issued actions, so that we do not need to enumerate all the possible execution traces. For instance, when the program (addAfter(a, b); || addAfter(a, c)) terminates, we have $\boxed{\mathtt{addAfter(a,b)}}_{t_1} \sqcup \boxed{\mathtt{addAfter(a,c)}}_{t_2}$.

We use $p \ltimes [\alpha]^i_t$, $p \ltimes \boxed{\alpha}^i_t$, $(p,\bowtie) \ltimes [\alpha]^i_t$ and $(p,\bowtie) \ltimes \boxed{\alpha}^i_t$ to add a new action α and some new orders about α. The assertion $p \ltimes [\alpha]^i_t$ requires α to be ordered after all the actions in p, while $(p,\bowtie) \ltimes [\alpha]^i_t$ enforces the ordering between α and only the actions which have arrived (e.g., boxed actions) in the current view of p and conflict ($\bowtie$) with α. The assertions $p \ltimes \boxed{\alpha}^i_t$ and $(p,\bowtie) \ltimes \boxed{\alpha}^i_t$ have similar meanings, but they also say that α has arrived at the current node. For the thread t_3 of Fig. 9, if the test of c ∈ v is true, it knows the following p_c: $[\mathtt{addAfter(a,b)}]_{t_1} \ltimes \boxed{\mathtt{addAfter(a,c)}}_{t_2}$. It says, t_3 can infer that addAfter(a,b) must be inserted before addAfter(a,c) even though addAfter(a,b) may not have arrived at t_3. After t_3 calls addAfter(c,d), the assertion becomes $(p_c,\bowtie) \ltimes \boxed{\mathtt{addAfter(c,d)}}_{t_3}$. Here t_3 adds only the ordering between the conflicting addAfter(a,c) and addAfter(c,d).

It is always safe to discard some ordering information. That is, $(p \ltimes [\alpha]^i_t) \Rightarrow (p \sqcup [\alpha]^i_t)$ holds. It is also safe to branch on the ordering of actions:

$$([\alpha]^i_t \sqcup [\alpha']^j_{t'}) \Rightarrow [\alpha]^i_t \ltimes [\alpha']^j_{t'} \vee [\alpha']^j_{t'} \ltimes [\alpha]^i_t$$

Standard state assertions, $\mathcal{P}$, can be lifted to action assertions. A set of partially ordered actions satisfies $\mathcal{P}$ if all the final states resulting from executing these actions satisfy $\mathcal{P}$ (as a state assertion). For instance, the following holds:

$$(\mathtt{s} = \mathtt{a} \wedge \mathsf{emp}) \sqcup (\boxed{\mathtt{addAfter(a,b)}}_{t_1} \ltimes \boxed{\mathtt{addAfter(a,c)}}_{t_2}) \Rightarrow \mathtt{s} = \mathtt{acb}$$

When executing the actions, we only execute the actions that have arrived in the current view. As a result,

$$(\mathtt{s} = \mathtt{a} \wedge \mathsf{emp}) \sqcup ([\mathtt{addAfter(a,b)}]_{t_1} \ltimes \boxed{\mathtt{addAfter(a,c)}}_{t_2}) \Rightarrow \mathtt{s} = \mathtt{ac} \vee \mathtt{s} = \mathtt{acb}$$

The assertion $p \Rrightarrow q$ specifies that the states satisfying q result from receiving and applying all the actions on the way in p. It is used when the whole client program terminates (see the PAR rule in Fig. 11, where in Q_t all the actions must have arrived at node t). For instance, the following holds:

$$(\mathtt{s} = \mathtt{a} \wedge \mathsf{emp}) \sqcup ([\mathtt{addAfter(a,b)}]_{t_1} \ltimes \boxed{\mathtt{addAfter(a,c)}}_{t_2}) \Rrightarrow \mathtt{s} = \mathtt{acb}$$

Rely/guarantee assertions. The assertions R and G (see Fig. 10) specify the interface between a thread and its environment. The guarantee G specifies the invocations of object actions made by the thread itself. The rely R specifies the thread's expectations of the object actions that originate from its environment.

The assertion Emp says there is no action issued. The assertion $p \rightsquigarrow [\alpha]^i_{\mathrm{t}}$ says that t invokes the action α when p holds, i.e., p is the prerequisite for t to issue the request α.

Threads can cooperate if the rely condition of a thread t is implied by the guarantee of the other t′. We stabilize the assertion p at each program point of t under its rely R, so that it is resistant to interference from the environment. To stabilize an assertion p with respect to $R = (p' \rightsquigarrow [\alpha]^i_{\mathrm{t}'})$, we do the following steps:

1. Check that the prerequisite p' for the invocation of α is met at p. This requires p to contain the knowledge of all the received actions $\boxed{\alpha'}^j_{\mathrm{t}''}$ in p', though it is possible that some of these actions have not arrived at the current node yet (i.e. they are in brackets in p).
2. If the check in (1) is passed, we add $[\alpha]^i_{\mathrm{t}'}$ to the action set of the current node. We do not need to know whether or not α has arrived at the current node.
3. The knowledge of the action ordering at the current node should also be expanded. For those α' in p' that are prerequisite of α and are also in conflict ($\bowtie$) with α, α' should be ordered before α on all the nodes, since we require all the nodes to observe the same ordering of conflicting actions.

For instance, $p \stackrel{\text{def}}{=} [\texttt{addAfter(a,b)}]_{\mathrm{t}_1}$ is stabilized to the following p_1 under R_1, for the RGA object:

$$\begin{aligned} R_1 &\stackrel{\text{def}}{=} \boxed{\texttt{addAfter(a,b)}}_{\mathrm{t}_1} \rightsquigarrow [\texttt{addAfter(a,c)}]_{\mathrm{t}_2} \\ p_1 &\stackrel{\text{def}}{=} p \vee ([\texttt{addAfter(a,b)}]_{\mathrm{t}_1} \ltimes [\texttt{addAfter(a,c)}]_{\mathrm{t}_2}) \end{aligned} \tag{7.1}$$

In the inference rules (see the CALL-R and LOCAL rules in Fig. 11), we use the stability check $\mathsf{Sta}(p, R, \bowtie)$. It is passed by stabilized assertions only. For (7.1), $\mathsf{Sta}(p_1, R_1, \bowtie)$ holds.

Inference rules. Figure 11 presents the key inference rules. The PAR rule is almost the standard parallel composition rule in rely-guarantee reasoning. We let each thread start its execution from an empty action set (see $\mathcal{P} \wedge \mathsf{emp}$). At the end, we derive the state assertion $\mathcal{Q}_{\mathrm{t}}$ by receiving all the actions in q_{t} (see $q_{\mathrm{t}} \Rrightarrow \mathcal{Q}_{\mathrm{t}}$). In the state assertions, we merge the client state and the object state into one, assuming their variables are from different name spaces. We also assume that the rely/guarantee conditions specify object states only.

In the CALL rule, we first compute the return value n' of the call, using $p \stackrel{\mu}{\twoheadrightarrow} n'$, where $\mu \in \mathit{AbsState} \to \mathit{Val}$ is the return value generator of $\Gamma(f, n)$. $p \stackrel{\mu}{\twoheadrightarrow} n'$ says, applying μ over any final state of executing the actions following the specified order in p returns n'. We then assign n' to x. The assertion q holds after the assignment, following the forward

$$\frac{\begin{array}{c}\forall \mathrm{t} \in [1..n]: \quad R_{\mathrm{t}}, G_{\mathrm{t}}; \Gamma, \bowtie \vdash_{\mathrm{t}} \{\mathcal{P} \wedge \mathsf{emp}\} C_{\mathrm{t}} \{q_{\mathrm{t}}\} \\ (\bigvee_{\mathrm{t}' \neq \mathrm{t}} G_{\mathrm{t}'}) \Rightarrow R_{\mathrm{t}} \qquad q_{\mathrm{t}} \Rrightarrow \mathcal{Q}_{\mathrm{t}}\end{array}}{\vdash \{\mathcal{P}\} \mathbf{with}\ (\Gamma, \bowtie)\ \mathbf{do}\ C_1 \parallel \ldots \parallel C_n \{\bigwedge_{\mathrm{t}} \mathcal{Q}_{\mathrm{t}}\}} \ (\textsc{par})$$

$$\frac{\begin{array}{c}p \Rightarrow E = n \qquad \mathsf{split}(\Gamma(f, n)) = (\mu, \alpha) \qquad p \stackrel{\mu}{\twoheadrightarrow} n' \\ x = n' \wedge \exists v.\, p[v/x] \Rightarrow q \qquad q \rightsquigarrow [\alpha]^i_{\mathrm{t}} \Rightarrow G\end{array}}{\mathsf{Emp}, G; \Gamma, \bowtie \vdash_{\mathrm{t}} \{p\} x := f(E) \{(q, \bowtie) \ltimes \boxed{\alpha}^i_{\mathrm{t}}\}} \ (\textsc{call})$$

$$\frac{\begin{array}{c}\mathsf{Emp}, G; \Gamma, \bowtie \vdash_{\mathrm{t}} \{p\} x := f(E) \{q\} \\ \mathsf{Sta}(\{p, q\}, R, \bowtie) \qquad \mathsf{cmt\text{-}closed}(\{p, q\})\end{array}}{R, G; \Gamma, \bowtie \vdash_{\mathrm{t}} \{p\} x := f(E) \{q\}} \ (\textsc{call-r})$$

$$\frac{\begin{array}{c}R', G'; \Gamma, \bowtie \vdash_{\mathrm{t}} \{p'\} C \{q'\} \\ p \Rightarrow p' \qquad R \Rightarrow R' \qquad q' \Rightarrow q \qquad G' \Rightarrow G\end{array}}{R, G; \Gamma, \bowtie \vdash_{\mathrm{t}} \{p\} C \{q\}} \ (\textsc{csq})$$

$$\frac{\mathsf{Sta}(p, R, \bowtie) \qquad \mathsf{cmt\text{-}closed}(p)}{R, G; \Gamma, \bowtie \vdash_{\mathrm{t}} \{p\} x := E \{\exists v.\, x = E[v/x] \wedge p[v/x]\}} \ (\textsc{local})$$

Figure 11. Selected inference rules.

assignment rule in Hoare logic. Finally we add the newly generated action α to the action set in q, and use the resulting assertion $(q, \bowtie) \ltimes \boxed{\alpha}^i_{\mathrm{t}}$ as the postcondition. The invocation of α following q (i.e., $q \rightsquigarrow [\alpha]^i_{\mathrm{t}}$) needs to satisfy G. The superscript i needs to be the same as specified in G.

One may wonder that it is too restrictive for the CALL rule to require the argument n and return value n' to be constant values. When the precondition p cannot determine a unique argument or return value (i.e., $(p \Rightarrow E = n)$ or $(p \stackrel{\mu}{\twoheadrightarrow} n')$ does not hold), we can first apply a standard disjunction rule to branch on p, and apply the CALL rule on each branch.

Note that in this step we only reason about the behavior of the function call without considering the environment. Therefore we use an empty rely condition Emp here. To allow a weaker R, we can apply the CSQ rule to stabilize the postcondition by weakening $(q, \bowtie) \ltimes \boxed{\alpha}^i_{\mathrm{t}}$. Then we apply CALL-R rule, which requires the pre- and post-conditions be stable with respect to R and satisfy cmt-closed. Here $\mathsf{cmt\text{-}closed}(p)$ iff p is preserved after receiving one or more actions that are already issued in p.

The LOCAL rule allows us to reason about local computation of a thread. The pre- and post-conditions are the same as those in the forward assignment rule in Hoare logic.

Verification of the motivating example. In Fig. 12 we sketch the proof of t_3 in the motivating example of Fig. 9. More examples are in the technical report [15].

We first define the rely/guarantee conditions of each thread. G_{t_1} says that the thread t_1 guarantees the invocation of α_b unconditionally. G_{t_2} says that t_2 calls α_c after it receives α_b. Similarly, G_{t_3} says that t_3 calls α_d after it receives α_c. Here we write $\diamond\boxed{\alpha}^i_{\mathrm{t}}$ for $\boxed{\alpha}^i_{\mathrm{t}} \sqcup \mathsf{true}$.

By the PAR rule, we only need to verify each thread independently. For thread t_3, we first stabilize p_a under R_{t_3}, resulting in the assertion (1) in Fig. 12. After finding $\mathsf{c} \in \mathsf{v}$, we

$$
\begin{array}{llll}
p_a \stackrel{\text{def}}{=} (\mathtt{s} = \mathtt{a}) \wedge \mathsf{emp} & & \alpha_b \stackrel{\text{def}}{=} \mathtt{addAfter(a,b)} \\
\alpha_c \stackrel{\text{def}}{=} \mathtt{addAfter(a,c)} & & \alpha_d \stackrel{\text{def}}{=} \mathtt{addAfter(c,d)} \\
G_{\mathrm{t}_1} \stackrel{\text{def}}{=} \mathsf{true} \rightsquigarrow [\alpha_b]_{\mathrm{t}_1} & & R_{\mathrm{t}_1} \stackrel{\text{def}}{=} G_{\mathrm{t}_2} \vee G_{\mathrm{t}_3} \\
G_{\mathrm{t}_2} \stackrel{\text{def}}{=} (\diamond \boxed{\alpha_b}_{\mathrm{t}_1}) \rightsquigarrow [\alpha_c]_{\mathrm{t}_2} & & R_{\mathrm{t}_2} \stackrel{\text{def}}{=} G_{\mathrm{t}_1} \vee G_{\mathrm{t}_3} \\
G_{\mathrm{t}_3} \stackrel{\text{def}}{=} (\diamond \boxed{\alpha_c}_{\mathrm{t}_2}) \rightsquigarrow [\alpha_d]_{\mathrm{t}_3} & & R_{\mathrm{t}_3} \stackrel{\text{def}}{=} G_{\mathrm{t}_1} \vee G_{\mathrm{t}_2}
\end{array}
$$

$\{p_a \vee p_a \sqcup [\alpha_b]_{\mathrm{t}_1} \vee p_a \sqcup ([\alpha_b]_{\mathrm{t}_1} \ltimes [\alpha_c]_{\mathrm{t}_2})\}$ (1)

```
v := read();
if (c ∈ v)
```

$\quad \{p_a \sqcup ([\alpha_b]_{\mathrm{t}_1} \ltimes \boxed{\alpha_c}_{\mathrm{t}_2})\}$ (2)

```
  addAfter(c, d);
```

$\quad \{p_a \sqcup ([\alpha_b]_{\mathrm{t}_1} \ltimes \boxed{\alpha_c}_{\mathrm{t}_2} \ltimes \boxed{\alpha_d}_{\mathrm{t}_3})\}$ (3)

```
y := read();
```

$\{\mathtt{s} = \mathtt{acdb} \Rightarrow \mathtt{y} = \mathtt{s} \vee \mathtt{y} = \mathtt{acd}\}$ (4)

Figure 12. Verification of the client with RGA.

can discard the branches where α_c is not arrived. So we get the assertion (2). Then, t_3 calls `addAfter(c,d)`. The immediate post-condition $(p \sqcup ([\alpha_b]_{\mathrm{t}_1} \ltimes \boxed{\alpha_c}_{\mathrm{t}_2}), \bowtie) \ltimes \boxed{\alpha_d}_{\mathrm{t}_3}$ can be derived from the CALL rule. Using the CSQ rule, we weaken it to the assertion (3), which is stable and cmt-closed. Finally we get the assertion (4). It has the branch $\mathtt{y} = \mathtt{acd}$ because it is possible that t_3 has not yet received α_b by the read.

Logic soundness: If $\vdash \{\mathcal{P}\}\mathbb{P}\{Q\}$, then $\models \{\mathcal{P}\}\mathbb{P}\{Q\}$. The Hoare triple $\models \{\mathcal{P}\}\mathbb{P}\{Q\}$ is defined using the abstract semantics in Sec. 6. The formal model and the soundness proofs are in our technical report [15].

Invariant-based reasoning. Our logic can be easily extended to verify object invariants. We can add an extra invariant assertion I in the judgment, which will be in the form of $I, R, G; \Gamma, \bowtie \vdash_{\mathrm{t}} \{p\}C\{q\}$. Then in the CALL-R rule in Fig. 11 we add the extra requirements $p \Rightarrow I$ and $q \Rightarrow I$.

8 Verifying CRDT Implementations

Our proof method for ACC asks users to first provide specifications $\rightarrowtail$ and $\mathcal{V}$ about implementations:

$$
\begin{array}{lll}
\rightarrowtail \;\in\; \mathscr{P}(\mathit{Effector} \times \mathit{Effector}) & & \text{(the time-stamp order)} \\
\mathcal{V} \;\in\; \mathit{LocalState} \to \mathscr{P}(\mathit{Effector}) & & \text{(the view function)}
\end{array}
$$

The time-stamp order $\rightarrowtail$ is a *partial order between effectors.* It describes the algorithm's conflict-resolution strategy, e.g., the write with a larger time-stamp wins. For the RGA algorithm, we instantiate $\rightarrowtail$ as follows:

$$
\begin{array}{l}
\delta \rightarrowtail \delta' \text{ iff } \exists \mathtt{a,i,b,a',i',b'}.\ \delta = \mathtt{AddAft(a,i,b)} \\
\qquad \wedge\ (\delta' = \mathtt{AddAft(a',i',b')} \wedge \mathtt{i} < \mathtt{i'} \\
\qquad\quad \vee\ \delta' = \mathtt{Rmv(a)} \vee \delta' = \mathtt{Rmv(b)})
\end{array}
$$

Here $\rightarrowtail$ orders the `AddAft` effectors by comparing their time-stamps. It also orders an `AddAft` before the conflicting `Rmv` effectors (which is not time-stamped). Note that $\rightarrowtail$ is specified at the implementation level. One should not confuse it with the won-by order $\blacktriangleleft$ over abstract operations, which we introduce in Sec. 2.4 and Sec. 9.

The view function $\mathcal{V}$ maps each local state $\mathcal{S}$ to a set of effectors that must have been applied before reaching $\mathcal{S}$. With it, our proof method can be local, in that the reasoning of each execution step relies on the current local state on the node only, without referring to the execution traces. For the RGA algorithm, $\mathcal{V}$ is instantiated as follows:

$$
\begin{array}{l}
\mathcal{V}(\mathcal{S}) \stackrel{\text{def}}{=} \{\delta \mid \exists \mathtt{a,i,b}.\ (\mathtt{a,i,b}) \in \mathcal{S}(\mathtt{N}) \wedge \delta = \mathtt{AddAft(a,i,b)} \\
\qquad\qquad \vee\ \exists \mathtt{a}.\ \mathtt{a} \in \mathcal{S}(\mathtt{T}) \wedge \delta = \mathtt{Rmv(a)}\}
\end{array}
$$

Our proof method, $\mathsf{CRDT\text{-}TS}_{\varphi}(\Pi, (\Gamma, \bowtie), \rightarrowtail, \mathcal{V})$, is a conjunction of the following proof obligations:

- Commutative effectors: the effectors generated by Π are all commutative.
- Same return value: the corresponding operations in Π and Γ have the same return value if executed at φ-related states.
- State correspondence: starting from φ-related states $\mathcal{S}$ and $\mathcal{S}_a$, executing a *valid* effector δ (generated from Π) and the corresponding abstract operation should lead to φ-related states. δ is valid if $\rightarrowtail$ does not order it before any δ' visible from $\mathcal{S}$, i.e. $\delta' \in \mathcal{V}(\mathcal{S})$.
- Some simple well-formedness checks for $\rightarrowtail$ and $\mathcal{V}$ to ensure the user-specified $\rightarrowtail$ and $\mathcal{V}$ make sense.

Theorem 8.
$\mathsf{CRDT\text{-}TS}_{\varphi}(\Pi, (\Gamma, \bowtie), \rightarrowtail, \mathcal{V}) \Longrightarrow \mathsf{ACC}_{\varphi}(\Pi, (\Gamma, \bowtie))$.

Examples. Using Theorem 8, we have verified seven CRDT algorithms [20], including the replicated counter (with both increment and decrement operations), the grow-only set, the last-writer-wins (LWW) register, the LWW-element set, the 2P-set, the continuous sequence, and the replicated growable array (RGA). To verify algorithms whose $\bowtie$ is empty (such as the counter), we let $\rightarrowtail$ be $\emptyset$ and $\mathcal{V}$ be $\lambda\mathcal{S}.\ \emptyset$. Proofs of the examples are in the technical report [15].

Using the verification framework. Our verification framework consists of the program logic (in Sec. 7) and the proof method (in Sec. 8). As Fig. 1 shows, one needs to do the following to verify a whole program **let** Π **in** $C_1 \parallel \ldots \parallel C_n$:

- Provide the specifications for CRDTs. The operation specification Γ is the same as the one for sequential data types. It is also easy to come up with the conflict relation $\bowtie$, which is between all the non-commutative abstract operations in Γ.
- Apply the program logic for client reasoning. Similar to standard rely-guarantee reasoning, the user needs to provide the rely/guarantee conditions, intermediate assertions, and do the proofs following the logic rules.
- Apply the proof method for CRDT implementations. All one needs to do is to provide $\rightarrowtail$ and $\mathcal{V}$, and prove the set of proof obligations. The proof obligations are all first-order formulae. They do not universally quantify over execution traces, but only over states and

effectors. Thus they can be discharged without induction, and can potentially be discharged by SMT solvers.

9 X-Wins CRDTs

Algorithms like add-wins sets and remove-wins sets resolve conflicts following a specific X-wins strategy, while the operation X wins only when its effect is not canceled. We generalize ACC to support these algorithms, by enforcing the X-wins strategy specified using the won-by ($\blacktriangleleft$) and canceled-by ($\triangleright$) relations. Like $\bowtie$ (see Fig. 7), they are also binary relation over actions. The full specification is now a quadruple $(\Gamma, \bowtie, \blacktriangleleft, \triangleright)$.

For add-wins sets, `add(x)` wins over concurrent `remove(x)` (`remove(x)` $\blacktriangleleft$ `add(x)`), but it can also be canceled by subsequent `remove(x)` (`add(x)` $\triangleright$ `remove(x)`); while for remove-wins sets, we have the inverse.

$\blacktriangleleft$ and $\triangleright$ can only relate conflicting operations, that is, $\blacktriangleleft \subseteq \bowtie$ and $\triangleright \subseteq \bowtie$. Also $\triangleright$ should be valid in that α' indeed nullifies the effects of α if $\alpha \triangleright \alpha'$. Like $\bowtie$, we also overload $\blacktriangleleft$ and $\triangleright$ over operations and events.

We generalize ACC with the extended specification, and define $\mathsf{XACC}_\varphi(\Pi, (\Gamma, \bowtie, \blacktriangleleft, \triangleright))$. It requires every trace $\mathcal{E}$ of Π to satisfy XACT if $\mathsf{causalDelivery}(\mathcal{E})$. Here we assume causal delivery of messages, which is required by both add-wins and remove-wins sets. It says, if an origin event e_1 happens before another origin event e_2, then for any node t the effector of e_1 reaches t earlier than that of e_2.

Definition 9. $\mathsf{XACT}_\varphi(\mathcal{E}, \mathcal{S}, (\Gamma, \bowtie, \blacktriangleleft, \triangleright))$ iff $\exists ar_1, \ldots, ar_n$,

$$\forall \mathsf{t}.\ \mathsf{totalOrder}_{\mathsf{visible}(\mathcal{E},\mathsf{t})}(ar_\mathsf{t}) \wedge (\xmapsto[\mathsf{t}]{\mathsf{vis}}_{\mathcal{E}} \subseteq ar_\mathsf{t})$$
$$\wedge\ \mathsf{PresvCancel}(ar_\mathsf{t}, \mathsf{t}, \mathcal{E}, (\Gamma, \triangleright)) \wedge \mathsf{ExecRelated}_\varphi(\mathsf{t}, (\mathcal{E}, \mathcal{S}), (\Gamma, ar_\mathsf{t}))$$
$$\wedge\ \forall \mathsf{t}' \neq \mathsf{t}.\ \mathsf{RCoh}_{(\mathsf{t},\mathsf{t}')}((ar_\mathsf{t}, ar_{\mathsf{t}'}), \mathcal{E}, (\Gamma, \bowtie, \blacktriangleleft, \triangleright))$$

where we define RCoh in Fig. 13.

XACT (see Def. 9) is similar to ACT, but it enforces the more relaxed coherence relation RCoh between the arbitration orders on different nodes. As defined in Fig. 13, RCoh requires that the arbitration orders ar_t and $ar_{\mathsf{t}'}$ of the nodes t and t′ enforce the same ordering for conflicting events e_0 and e_1, if neither e_0 or e_1 are canceled (i.e., $\{e_0, e_1\} \subseteq \mathsf{nc\text{-}vis}(\mathcal{E}', \mathsf{t}, (\Gamma, \triangleright)) \cap \mathsf{nc\text{-}vis}(\mathcal{E}'', \mathsf{t}', (\Gamma, \triangleright))$). Moreover, the ordering must follow the won-by order $\blacktriangleleft$ if these two events are concurrent (i.e., neither one happens before the other). It is more relaxed than Coh in that, if either e_0 or e_1 is canceled by others, they can be ordered differently in ar_t and $ar_{\mathsf{t}'}$.

XACT also requires $\mathsf{PresvCancel}(ar_\mathsf{t}, \mathsf{t}, \mathcal{E}, (\Gamma, \triangleright))$. It says, if e_1 is canceled by e_2 and is also visible to e_2 on certain node, the arbitration order ar_t must order e_1 before e_2.

Similar to ACC, XACC also ensures SEC, and is compositional. We prove that both the add-wins and remove-wins sets satisfy XACC.

The Abstraction Theorem. We also revise the abstract operational semantics in Sec. 6, to give clients an abstract view of the X-wins strategy. We then redefine the contextual

$$\begin{aligned}
&\mathsf{RCoh}_{(\mathsf{t},\mathsf{t}')}((ar_\mathsf{t}, ar_{\mathsf{t}'}), \mathcal{E}, (\Gamma, \bowtie, \blacktriangleleft, \triangleright)) \ \text{ iff } \ \forall \mathcal{E}', \mathcal{E}'', e_0, e_1.\\
&\quad \mathcal{E}' \leqslant \mathcal{E} \wedge \mathcal{E}'' \leqslant \mathcal{E} \ \wedge\ e_0 \bowtie_\Gamma e_1 \ \wedge\\
&\qquad \{e_0, e_1\} \subseteq \mathsf{nc\text{-}vis}(\mathcal{E}', \mathsf{t}, (\Gamma, \triangleright)) \cap \mathsf{nc\text{-}vis}(\mathcal{E}'', \mathsf{t}', (\Gamma, \triangleright))\\
&\quad \Longrightarrow ((e_0, e_1) \in ar_\mathsf{t} \cap ar_{\mathsf{t}'} \vee (e_1, e_0) \in ar_\mathsf{t} \cap ar_{\mathsf{t}'}) \ \wedge\\
&\qquad\quad (\mathsf{Concurrent}_{\mathcal{E}}(e_0, e_1) \wedge (e_0 \blacktriangleleft_\Gamma e_1) \Longrightarrow (e_0, e_1) \in ar_\mathsf{t})\\
&\mathsf{nc\text{-}vis}(\mathcal{E}, \mathsf{t}, (\Gamma, \triangleright)) \ \stackrel{\text{def}}{=}\ \{e \mid e \in \mathsf{visible}(\mathcal{E}, \mathsf{t}) \ \wedge\\
&\qquad\qquad \neg(\exists e'.\ e' \in \mathsf{visible}(\mathcal{E}, \mathsf{t}) \wedge (e \triangleright_\Gamma e') \wedge (e \xmapsto{\mathsf{vis}}_{\mathcal{E}} e'))\}
\end{aligned}$$

Figure 13. Auxiliary definitions for XACC.

refinement $\Pi \sqsubseteq_\varphi (\Gamma, \bowtie, \blacktriangleleft, \triangleright)$. It is similar to $\Pi \sqsubseteq_\varphi (\Gamma, \bowtie)$ (Def. 6), but uses the new abstract semantics and assumes causal delivery on concrete executions. Correspondingly, we have a new abstraction theorem showing its equivalence to XACC (see the technical report [15]).

10 Related Work

Attiya et al. [1] propose a functional correctness criterion specifically for the RGA algorithm. They do not use an operational atomic specification as we do, but instead characterize the lists' functionality axiomatically (e.g., by requiring an element be in the list if it has been inserted but not deleted). Both our ACC and their work require different nodes to take the same arbitration order between `addAfter` events. Our ACC is more general and can apply to other data types too.

Jagadeesan and Riely [10] propose a correctness criterion encoding both SEC and functional correctness for CRDTs. Their "sequential specification" is a set of legal sequential traces. It is accompanied with a dependency relation between abstract operations, which plays a similar role as our $\bowtie$ relation. Their correctness definition computes the *dependent cuts* of an execution of CRDT, which is similar to our $\mathsf{visible}(\mathcal{E}, \mathsf{t})$ projected to conflicting actions. They require all nodes to have the same arbitration orders (i.e., linearizations), but over dependent actions only. This is in spirit similar to our approach, which requires the arbitration orders of different nodes to be coherent on conflicting actions. To support add-wins sets, their linearizations view different calls to the same operation as interchangeable (a.k.a. puns). By contrast, our XACC encodes the X-wins strategy of these algorithms directly, taking the effects of cancellation into account. Similar ideas of cancellation can be found in the earlier work on checking serializability [4]. Note that Jagadeesan and Riely [10] do *not* give a proof method for *client* reasoning. Also, they verify the CRDT algorithms case by case without giving a generic proof method.

Wang et al. [22] propose RA-linearizability for CRDTs. Their specifications are *non-atomic*, and often have to expose some low-level implementation details. For instance, their specification for RGA needs the tombstone set of removed elements, and their specification for add-wins sets splits a `remove` into two abstract operations. By contrast, we use atomic and implementation-independent specifications. They also give proof methods for RA-linearizability, which contain some trace-based proof obligations such as commutativity, while our proof obligations for ACC are state-based. Besides, they do *not* provide formal solutions for program logic for client verification.

Gotsman et al. [9] verify data integrity invariants for clients of replicated data types. They do not prove pre- and post-conditions as we do, which can be used specify more interesting functional properties. They introduce a token system with a conflict relation $\bowtie$ to relate operations that need to be causally dependent. We use the same symbol $\bowtie$ to relate non-commmutative abstract operations.

Lewchenko et al. [14] propose conflict-aware replicated data types (CARD), and design a refinement type system that enables verification of pre- and post-conditions for clients of CARD. There is also much work about general verification approaches for distributed systems and their clients (e.g., [19, 24]). Our program logic is customized for clients of CRDTs only. We can utilize certain properties (e.g., SEC) of CRDTs in the verification of clients.

Several papers (e.g., [3, 5, 6, 21, 25]) use concurrent specifications for replicated data types. On the one hand, concurrent specifications are more general than sequential specifications, so they can in principle support any replicated data types. On the other hand, it is unclear how to utilize the concurrent specifications in client reasoning.

Gomes et al. [8] verify SEC of CRDTs in Isabelle/HOL. Their method is based on global execution traces. Our proof method is local and state-based, and verifies functional correctness as well as SEC. Nagar and Jagannathan [16] verify SEC of CRDTs automatically. Their verification is parameterized with consistency policies offered by the underlying network (e.g., whether message delivery is causal). Kaki et al. [12] verify invariants for clients of replicated data types. Their approach is based on symbolic execution with a bound on concurrent operations. They also repair the invariant violations of clients by strengthening the network's consistency policies. It would be interesting to also study our ACC and our client logic with various network consistency policies.

There is also work that verifies eventual consistency and/or causal consistency by model-checking (e.g., [2, 3]), or for some particular data types such as key-value stores [13].

11 Conclusion and Future Work

We develop a theory of data abstraction for CRDTs, with independent proof methods to verify CRDT implementations and client programs respectively. Our Abstraction Theorem, as one of the key results in the theory, decomposes the verification of the two sides so that they can be done independently and modularly. It forms a semantic basis for understanding CRDTs, based on which we believe more proof techniques and tools can be developed in the future.

Limitations. This paper mostly focuses on UCR-CRDTs. For X-wins CRDTs, we formulate XACC and prove both the add-wins set and remove-wins set satisfy XACC. It might be possible to develop a general proof method for verifying XACC, similar to CRDT-TS for verifying ACC (Sec. 8), but one needs to be careful to avoid overfitting, since we do not have many interesting X-wins CRDTs to test the generality. Also, we leave the program logic for clients using X-wins CRDTs as future work. To reason about their clients, one needs to take into account the X-wins strategy specified using the won-by ($\blacktriangleleft$) and canceled-by ($\triangleright$) relations, and ensure soundness of the logic w.r.t. the new abstract operational semantics discussed in Sec. 9.

Our verification of CRDTs is done at the algorithm level. To bridge the real code with the operations defined in Π (see Fig. 6), one only needs refinement proofs for sequential programs since the real implementation code runs sequentially on individual hosts.

This paper considers only *operation-based* CRDTs. Our results may be adapted to support *state-based* CRDTs when assuming causal delivery, but it seems nontrivial to build abstractions that on the one hand reflect the algorithms' resistance to unreliable networks, and on the other hand are still useful for client reasoning. Nair et al. [17] recently propose a proof method for verifying invariant preservation of state-based replicated objects. It would be interesting to incorporate their ideas into our work.

We would also like to further test the applicability of our results by considering new operation-based CRDT algorithms (e.g., those constructed by semidirect products [23]). It is also interesting to mechanize our results in proof assistants and explore the possibility of building tools to automate the verification process.

Acknowledgments

We thank our shepherd Hongseok Yang and anonymous referees for their suggestions and comments on earlier versions of this paper. This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant Nos. 61922039 and 61632005.

References

[1] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. 2016. Specification and Complexity of Collaborative Text Editing. In *PODC 2016*. 259–268. https://doi.org/10.1145/2933057.2933090

[2] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On Verifying Causal Consistency. In *POPL 2017*. 626–638. https://doi.org/10.1145/3009837.3009888

[3] Ahmed Bouajjani, Constantin Enea, and Jad Hamza. 2014. Verifying Eventual Consistency of Optimistic Replication Systems. In *POPL 2014*. 285–296. https://doi.org/10.1145/2535838.2535877

[4] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *POPL 2017*. 458–472. https://doi.org/10.1145/3009837.3009895

[5] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1-2 (Oct. 2014), 1–150. https://doi.org/10.

1561/2500000011

[6] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *POPL 2014*. 271–284. https://doi.org/10.1145/2535838.2535848

[7] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59. https://doi.org/10.1145/564585.564601

[8] 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

[9] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *POPL 2016*. 371–384. https://doi.org/10.1145/2837614.2837625

[10] Radha Jagadeesan and James Riely. 2018. Eventual Consistency for CRDTs. In *ESOP 2018*. 968–995. https://doi.org/10.1007/978-3-319-89884-1_34

[11] Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619. https://doi.org/10.1145/69575.69577

[12] Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. 2018. Safe Replication through Bounded Concurrency Verification. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 164 (2018). https://doi.org/10.1145/3276534

[13] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *POPL 2016*. 357–370. https://doi.org/10.1145/2837614.2837622

[14] Nicholas V. Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černý. 2019. Sequential Programming for Replicated Data Stores. *Proc. ACM Program. Lang.* 3, ICFP, Article 106 (2019). https://doi.org/10.1145/3341710

[15] Hongjin Liang and Xinyu Feng. 2021. Abstraction for Conflict-Free Replicated Data Types (Technical Report). https://plax-lab.github.io/publications/crdt/

[16] Kartik Nagar and Suresh Jagannathan. 2019. Automated Parameterized Verification of CRDTs. In *CAV 2019*. 459–477. https://doi.org/10.1007/978-3-030-25543-5_26

[17] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the Safety of Highly-Available Distributed Objects. In *ESOP 2020*. 544–571. https://doi.org/10.1007/978-3-030-44914-8_20

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

[19] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (2017). https://doi.org/10.1145/3158116

[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. https://hal.inria.fr/inria-00555588

[21] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1 (June 2016), 19:1–19:34. https://doi.org/10.1145/2926965

[22] Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. 2019. Replication-aware Linearizability. In *PLDI 2019*. 980–993. https://doi.org/10.1145/3314221.3314617

[23] Matthew Weidner, Heather Miller, and Christopher Meiklejohn. 2020. Composing and Decomposing Op-Based CRDTs with Semidirect Products. *Proc. ACM Program. Lang.* 4, ICFP, Article 94 (Aug. 2020). https://doi.org/10.1145/3408976

[24] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *PLDI 2015*. 357–368. https://doi.org/10.1145/2737924.2737958

[25] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal Specification and Verification of CRDTs. In *FORTE 2014*. 33–48. https://doi.org/10.1007/978-3-662-43613-4_3