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

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 Γ in C_1 ∥ ... ∥ C_n” to represent a program consisting of client programs C_1, ..., 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 \( \Pi \) with respect to the abstraction \( \Gamma \) 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 \( \Gamma \), without worrying about the implementation details in \( \Pi \) (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 \( \Gamma \), which are traditionally used for sequential data types and shared-memory concurrent objects. To facilitate the client reasoning, each \( \Gamma \) is also accompanied with a conflict relation \( \Rightarrow \) 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 (\( \triangleright \) in Fig. 1, also in Sec. 5). ACC establishes an abstract view of execution based on the atomic specifications \( \Gamma \), 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 \( \Rightarrow \)) 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 \( \Pi \) of CRDT operations and the atomic specification \( \Gamma \), 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 (\( \triangleright \) in Fig. 1, also in Sec. 7). Together with the contextual refinement, our logic offers a way to 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 (\( \triangleright \) 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.

\section{Informal Development}

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

\subsection{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: \texttt{addAfter} \((a,b)\) adds the element \(b\) after \(a\) in the list, \texttt{remove} \((a)\) removes the element \(a\) from the list, and \texttt{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 \(\diamond\).
1 var N := ∅, T := ∅;
2 var ts := (∅, cid);
3 operation addAfter(a, b){
4    assume(a = o ∨
5    a ≠ o ∧ (ts, a) ∈ N ∧ a ≠ T);
6    local i := (ts.fst+1, cid);
7    return;
8    addAfter(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((ts, a) ∈ N ∧
20    a ≠ T ∧ a ≠ o);
21    return;
22    gen_eff Rmv(a);
23 }
24 effector Rmv(a){
25    T := T ∪ {a};
26 }
27
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:

N = {(o, ts0, a), (a, ts1, e), (a, ts2, b), (a, ts3, c), (c, ts4, d)}

We assume o 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 ts0 < ts1 < ts2 < ts3 < ts4.

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: (n1, t1) > (n2, t2) holds if (n1 > n2) or (n1 = n2) ∧ (t1 > t2). 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 \( \Pi \text{RGA} \) in \( C_1 \parallel C_2 \), where \( \Pi \text{RGA} \) 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 \( E \) of events recording the execution of all the operations (both originals and effectors), and \( E|_t \) 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 \( E \) (assuming ts0 < ts2 and the initial list contains a only). We also record the arguments and return values (if any) of each operation.

\[
\begin{align*}
(t_1, \text{addAfter}(a, b), ts_1), & (t_2, \text{addAfter}(a, c), ts_2), \\
(t_2, \text{AddAft}(a, ts_1), & (t_1, \text{AddAft}(a, ts_2), c)), \\
(t_1, \text{read}(), & acb, (t_2, \text{read}(), acb)
\end{align*}
\]

The event \( (t_1, \text{addAfter}(a, b), ts_1) \) 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
(t₂, AddAfter(a, ts₁, b)) represents the execution of an effector on t₂ (sent from other nodes). Then the local traces \( E|_{t_2} \) and \( E|_{t_1} \) are the following:

\[
(\text{t₁, addAfter(a, b), ts₁}, (\text{t₁, AddAfter(a, ts₂, c)}, (\text{t₁, read()}, \text{acb}) (\text{t₂, addAfter(a, c), ts₂}), (\text{t₂, AddAfter(a, ts₁, b)}), (\text{t₂, read()}, \text{acb})
\]

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. 2) 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 t₁ and t₂ in Fig. 3(a) may both interpret their local execution traces as the following sequential execution of atomic operations:

\[
\text{addAfter-atom(a, b), addAfter-atom(a, c), (read()), acb}
\]

Here \( \text{addAfter-atom(x, y} \) represents an abstract atomic specification of \( \text{addAfter(x, y)} \). 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 \( \text{acb} \) of the final \( \text{read} \) determines the order between \( \text{addAfter-atom(a, b)} \) and \( \text{addAfter-atom(a, c)} \). Therefore, for the node t₂, the abstract operations have to be executed in a different order from the order of the effectors in its concrete trace \( E|_{t_2} \).

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 \( \alpha_t \) over these operations. Our FC requires, for every prefix \( \mathcal{E}' \) of \( \mathcal{E} \), the sub-trace \( \mathcal{E}'|_{t} \) that t sees locally may correspond to an abstract trace \( \mathcal{E}'' \) following the total order \( \alpha_t \), such that performing \( \mathcal{E}'|_{t} \) has the same effects as performing \( \mathcal{E}''|_{t} \), that is, \textit{they generate the same state (modulo the state abstraction), and the same return value if \( \mathcal{E}'|_{t} \) ends with a query operation.}

In the example both \( \alpha_t \) and \( \alpha_{t₁} \) order \( \text{addAfter-atom(a, b)} \) before \( \text{addAfter-atom(a, c)} \). For t₂, we consider its local traces of all the prefixes of \( \mathcal{E} \):

\[
\begin{align*}
\mathcal{E}_1 : & (t₂, \text{addAfter(a, c), ts₂}) \\
\mathcal{E}_2 : & (t₂, \text{addAfter(a, c), ts₂}, (t₂, \text{AddAfter(a, ts₁, b)})) \\
\mathcal{E}_3 : & (t₂, \text{addAfter(a, c), ts₂}, (t₂, \text{AddAfter(a, ts₁, b)}), (t₂, \text{read()}, \text{acb}))
\end{align*}
\]

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

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 \textit{the existence} of an order \( \alpha_t \) on each node t to order the abstract operations, and says nothing about what the \( \alpha_t \) 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 \( \alpha_t \) 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 \( \alpha_t \) 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), \( \alpha_t \) has to order \( \text{addAfter-atom(a, b)} \) before \( \text{addAfter-atom(a, c)} \), which is different from the real-time order of concrete operations in \( E|_{t₁} \). 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 \text{read} of t₂ is made after the arrival of the effector of \( \text{addAfter(a, b)} \) from t₁. In this case we say \( \text{addAfter(a, b)} \) is \textit{visible to} \( u := \text{read(c)} \). 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 \( \alpha_t \) of abstract operations. SEC would be obvious if all \( \alpha_t \) are the same. However, as we would see below, this requirement is overly restrictive and cannot be satisfied by some CRDTs.
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 ③ 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 ③, while the tag of ③ is smaller than that of ②. To interpret the final sequence apqced, node t1 has to order the abstract operation ③ 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 t1 is ①②③④. Similarly, the only possible order for t2 is ②①③④. So ① and ② (also ③ and ④) must be ordered differently by t1 and t2.

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 \( a \) and \( b \) are conflicting, represented as \( a \Rightarrow b \), 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 \( \text{addAfter}(a, q) \) and \( \text{addAfter}(a, p) \) and \( \text{addAfter}(a, q) \) and \( \text{addAfter}(c, q) \) are 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 \( E \)) and its abstract specification (represented as a pair \( (Γ, ⊲⊳) \), where \( Γ \) is the abstract atomic specification of the operations, and \( ⊲⊳ \) 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, ACC\((Π, (Γ, ⊲⊳))\) requires that, for any execution trace \( E \) of \( Π \), each node \( t \) can find a total order \( ar_t \) over abstract atomic operations in \( Γ \), such that:

- For each prefix \( E' \), there is a corresponding sequence \( E'' \) of abstract operations. \( E'' \) follows the order \( ar_t \) and generates the same effects with \( E'|t \);
- \( ar_t \) preserves the local visibility order on \( t \);
- 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 \( ⊲⊳ \).

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: \( \text{lookup(e)} \), \( \text{add(e)} \) and \( \text{remove(e)} \). 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 \( \text{Rmv}((1, b)) \) 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 \( \text{lookup}(1) \)
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 $\prec$ in the abstract specification to describe the conflict resolving strategy. We have remove(e) $\prec$ add(e) for add-wins set, and the reverse for remove-wins set. Since we only need to resolve conflicts for conflicting operations, the $\prec$ relation is a subset of the conflict relation $\sqsubseteq$. Correspondingly, we refine the third point of ACC in 2.3 with an extra requirement that all the $a_n$ respect the $\prec$ 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 $\sigma$ satisfying ACC. For $t_1$, we have to order $\{1\}$ before $\{3\}$ (to preserve the visibility order), and $\{3\}$ before $\{2\}$ (to respect the $\prec$ order). Therefore $\{3\}$ has to be the last operation, otherwise the abstract execution cannot generate the same return values as the concrete one, failing FC. However, ordering $\{3\}$ after the concurrent $\{1\}$ would violate the $\prec$ 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 $\{3\}$ arrives at $t_1$, the effect of $\{3\}$ has already been canceled out by $\{2\}$. 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 $\triangleright$; 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 $a_n$ for different nodes $t$. This relaxed ACC allows the total orders $a_n$ and $a_n'$ in Fig. 5(b) to be defined as $(\circ \circ)\circ\circ$ and $(\circ\circ\circ\circ)$, respectively. When $\{3\}$ is executed at $t_1$, we only need to check that $\{2\}$ and $\{3\}$ are ordered consistently, and ignore $\{1\} \triangleright \{2\}$ because they have been canceled by $\{3\} \triangleright \{2\}$ at this moment. Also because $\{2\} \triangleright \{3\}$ are not conflicting (they are commutative), it is okay to order them differently in $a_n$ and $a_n'$.

With the more refined specification, we can redefine the correctness as XACC($\Pi$, ($\Gamma$, $\triangleright$, $\prec$, $\triangleright$)). It also assumes causal delivery of messages, as required by add-wins and remove-wins sets. Note that UCR-CRDTs satisfying ACC($\Pi$, ($\Gamma$, $\triangleright$)) in Sec. 2.3 also satisfy XACC($\Pi$, ($\Gamma$, $\triangleright$, $\emptyset$, $\emptyset$)) — Since their conflict resolving policies are not tied with particular operations, we can simply set $\triangleright$ 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 $\Pi_i$ satisfies XACC($\Pi_i$, ($\Gamma_i$, $\triangleright$, $\prec$, $\triangleright$)), then the clients can use them together and view them as a single big object satisfying XACC($\Pi$, ($\Gamma$, $\triangleright$, $\prec$, $\triangleright$)), where $\Pi$ represents the disjoint union of all the operations $\Pi_1 \cup \ldots \cup \Pi_n$, and $\Gamma$, $\prec$, $\triangleright$ and $\triangleright$ 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. 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);
x := read(); y := read();
```

At the end the post-condition 0 ∈ x ⇒ 0 ∉ 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 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 let Π in C_1 || . . . || C_n are executing 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 E below:

```
(Expr) E ::= x | E + E | ...
(CliSim) C ::= x := f(E) | skip | C ; C | if (E) C else C | ...
(Prog) P ::= let Π in C_1 || . . . || C_n
```

Figure 6. Syntax of the programming language.

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

4 Specifications for CRDTs

The specification of a CRDT object consists of two parts, the operation specification Γ and the conflict relation ≺≺≺≺≺, as shown in Fig. 7. Γ maps operation names and arguments
(OSpec) \( \Gamma \in \text{OpName} \times \text{Val} \rightarrow \text{AbsState} \rightarrow \text{Val} \times \text{AbsState} \)

(Action) \( \alpha \in \text{AbsState} \rightarrow \text{AbsState} \)

\( \Rightarrow \in \mathcal{P}(\text{Action} \times \text{Action}) \)

Figure 7. Object specifications (\( \Gamma, \Rightarrow \)).

to abstract atomic operations of the type \( \text{AbsState} \rightarrow \text{Val} \times \text{AbsState} \). 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 \( \text{AbsState} \) to represent the set of object states \( S \) 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 \( \mathcal{T} \) and a tombstone \( \top \), 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 \( \text{opr} (\Gamma (f, n)) \) to represent the effects of \( \Gamma (f, n) \), which does a state transformation. We call the transformation an action (represented as \( \alpha \)).

The conflict relation \( \Rightarrow \) needs to be a symmetric binary relation over non-commutative actions. For sets, \text{add(x)} and \text{remove(x)} conflict with each other for RGA,

\[
\text{addAfter}(a, b) \Rightarrow \text{addAfter}(c, d) \quad \text{iff} \quad \{a, b\} \cap \{c, d\} \neq \emptyset,
\]

\[
\text{addAfter}(a, b) \Rightarrow \text{remove}(c) \quad \text{iff} \quad c \in \{a, b\}.
\]

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

Definition 1. \( \text{nonComm}(\Gamma, \Rightarrow) \) if \( \forall f_1, f_2, n_1, n_2, \alpha_1, \alpha_2, \)

\[
\alpha_1 = \text{opr}(\Gamma (f_1, n_1)) \land \alpha_2 = \text{opr}(\Gamma (f_2, n_2)) \land \neg(\alpha_1 \Rightarrow \alpha_2)
\]

\( \Rightarrow \alpha_1 \upharpoonleft \alpha_2 = \alpha_2 \upharpoonleft \alpha_1 \)

where \( \alpha \upharpoonleft \alpha' \overset{\text{def}}{=} \lambda S. \alpha'(\alpha(S)) \).

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 (\( \upharpoonleft \)) and canceled-by (\( \upharpoonright \)) relations over conflicting actions. In the following sections we first present our results for UCR-CRDTs that do not need \( \ll \) and \( \gg \), and show the extension of them to support these X-wins algorithms in Sec. 9.

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

5 Abstract Converging Consistency

As shown in Def. 2, \( \text{ACC}(\Pi, (\Gamma, \Rightarrow)) \) is parameterized with an abstraction function \( \varphi \), which maps concrete object states to abstract ones, i.e., \( \varphi \in \text{LocalState} \rightarrow \text{AbsState} \).

\[
\text{ExecRelated}_\varphi(t, (E, S), (\Gamma, ar)) \iff \forall E' \subseteq E.
\]

\[
\forall (S_{n'}', n') = \text{exec}(\varphi(S), \text{visible}(E', t) \mid \text{ar}) \quad \varphi(\text{exec}_{\text{st}}(S, E'), t) = S_d' \land
\quad (\forall e = \text{last}(E')(t). \text{is_orig}(e) \Rightarrow \text{val}(e) = n')
\]

\[
\text{Coh}(ar, ar', (\Gamma, \Rightarrow)) \iff
\quad \forall e_1, e_2, (e_1 ar e_2) \land (e_2 ar' e_1) \Rightarrow \neg(e_1 \Rightarrow e_2)
\]

Figure 8. Auxiliary definitions for ACC.

Definition 2. \( \text{ACC}(\Pi, (\Gamma, \Rightarrow)) \) if

\[
\forall S, E, E \in T (\Pi, S) \land S \in \text{dom}(\varphi) \Rightarrow \text{ACT}(E, S, (\Gamma, \Rightarrow))
\]

It requires every event trace \( E \) of \( \Pi \) to satisfy \( \text{ACT} \) shown in Def. 3, which formalizes the idea in Sec. 2.3.

Definition 3. \( \text{ACT}(E, S, (\Gamma, \Rightarrow)) \) if

\[
\forall t. \text{totalOrder}_{\text{visible}(E)}(\text{an}_t) \land (\forall t \in \text{vis}(\text{E}) \subseteq \text{an}_t) \land
\quad \text{ExecRelated}_\varphi(t, (E, S), (\Gamma, \text{an}_t)) \land \forall t'. \text{Coh}(\text{ar}_t, \text{ar}_t', (\Gamma, \Rightarrow))
\]

where we define \( \text{ExecRelated} \) and \( \text{Coh} \) in Fig. 8.

Before explaining \( \text{ACT} \), we first introduce the notations for visibility of events. In the execution \( E \) an origin event \( e \) is \textit{visible} to another event \( e' \) originated from the node \( t \) (i.e., \( e \xrightarrow{\text{vis}}_t E \subseteq e' \)), if the effect of \( e \) has reached \( t \) before \( e' \) is issued. We also use \( \text{visible}(E, t) \) to represent the set of origin events whose effectors have reached \( t \).

\( \text{ACT} \) says that each node \( t \) may have its own arbitration order \( \text{an}_t \), which is a total order over the origin events on \( E \) visible to \( t \). Each \( \text{an}_t \) must preserve the visibility order on \( E \) (i.e., \( \forall t \text{vis}(\text{E}) \subseteq \text{an}_t \)).

On functional correctness, \( \text{ACT} \) requires that the concrete execution on node \( t \) should correspond to the execution of the abstract events following the arbitration order \( \text{an}_t \) (see \( \text{ExecRelated}_\varphi(t, (E, S), (\Gamma, \text{an}_t)) \)). As defined in Fig. 8, \( \text{ExecRelated} \) says that every state in \( t \)'s concrete execution can be mapped (via \( \varphi \)) 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 \( E' \) of the concrete trace \( E \). We use \( \text{visible}(E', t) \mid \text{ar} \) to represent a serialization of the set visible \( E', t \) following the total order \( \text{ar} \). Then \( \text{exec}(\Gamma, S_{n_t}, E) \) executes the sequence of abstract operations on \( E \), starting from the initial abstract state \( S \). It returns the final state \( S_{n_t}' \) and the return value \( n' \) of the last operation. Similarly, we use \( \text{exec}_{\text{st}}(S, E) \) to represent the final state generated by executing the effectors on \( E \) from the initial state \( S \). 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 \( \text{Coh}(\text{an}_t, \text{an}_{t'}, (\Gamma, \Rightarrow)) \)), as defined in
Fig. 8). Combined with (vis \_ t \leq a_n) 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.** Cvt\_φ(\_E, S) iff
\n\forall E', E'', t', t. E' \leq E \land E'' \leq E \land \text{visible}(E', t) = \text{visible}(E'', t') \implies \phi(\text{exec\_st}(S, E'\_t)) = \phi(\text{exec\_st}(S, E''\_t))

Cvt\_φ(\_E, 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 E (see E' \leq E and E'' \leq E, which says E' and E'' can be different prefixes of 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 Cvt\_φ(Π), requires every event trace E of Π to satisfy Cvt.

**Lemma 5.** If ACC\_φ(Π, (Γ, ≻⊳)) then Cvt\_φ(Π).

Another important property of ACC\_φ(Π, (Γ, ≻⊳)) 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 (Γ, ≻⊳). The abstract version of the client program is defined below:

\[(AProg) \_P := \text{with } (Γ, ≻⊳) \text{ do } C_1 \| \ldots \| C_n\]

It is safe to reason about clients at the abstract level as long as the CRDT implementation Π contextually refines (Γ, ≻⊳).

**Definition 6.** Π \subseteq_φ (Γ, ≻⊳) iff, for all clients C\_1, ..., C\_n and state S \in dom(φ), for all [E] and σ\_E,
\n(\_E, σ\_E) \in T\_φ(Π) \_in \_C\_1 \| \ldots \| C\_n, S) \implies
\n(obs\_φ([E]), σ\_E) \in T\_φ(\text{with } (Γ, ≻⊳) \text{ do } C\_1 \| \ldots \| C\_n, φ(S))

Informally, Π \subseteq_φ (Γ, ≻⊳) says, for any clients and initial states, executing the clients with Π does not generate more observable behaviors than the execution using (Γ, ≻⊳) in the abstract operational semantics (presented below). T\_φ(Π, S) and T\_φ(Π, S) are defined similarly as T(Π, S) (Sec. 3), but they additionally record the final state σ\_E. Also in the extended trace [E] they record all the intermediate object states together with the events. The function obs\_φ([E]) maps the extended trace [E] 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).**

\[\text{ACC}_φ(Π, (Γ, ≻⊳)) \iff Π \subseteq_φ (Γ, ≻⊳).\]

**Abstract operational semantics** describes the execution of programs in the form of with (Γ, ≻⊳) do C\_1 \| \ldots \| C\_n.

Clients are executed following the interleaving semantics. On each node, we always keep the initial object state 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 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 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 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 Coh(ar\_t, ar\_n, (Γ, ≻⊳)) 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 Π \subseteq_φ (Γ, ≻⊳) can ensure Cvt\_φ(Π), the convergence of the concrete object. With the Abstraction Theorem (Thm 7), we can derive Lem. 5 again: ACC\_φ(Π, (Γ, ≻⊳)) can ensure Cvt\_φ(Π) 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 (Γ, ≻⊳) 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 \(\vdash \{P\} \text{with } (Γ, ≻⊳) \text{ do } C\_1 \| \ldots \| C\_n \{Q\}\), where P and Q are traditional Hoare-logic state assertions over both client and object states. To enable thread-local reasoning, we borrow ideas from shared-memory state assertions and base our logic on rely-guarantee reasoning [11]. Each C\_i is verified in the form of R, G; Γ, ≻⊳ \vdash \{P\} C\_i \{Q\}, where R and G
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 \( s = a \). 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 \( \text{addAfter}(a, c) \) from the thread \( t_2 \), it may not have received \( \text{addAfter}(a, b) \) from the thread \( t_1 \), though \( \text{addAfter}(a, c) \) is issued only after \( t_2 \) receives \( \text{addAfter}(a, b) \). As a result, it is possible that \( y \) reads \( \text{acdb} \). But, when \( t_3 \) finally receives \( \text{addAfter}(a, b) \), it must insert \( \text{addAfter}(a, b) \) before \( \text{addAfter}(a, c) \) (in the abstract semantics) to restore the causality (required by the coherence check). It is impossible for \( y \) to read \( \text{abcd} \).

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 \([a_i]^P\) and \([\alpha]^P\) describe singleton action sets containing only the action \( a \). The former says the action \( a \) (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 \( a \). 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 \( c \in v \), its assertion must contain \([\text{addAfter}(a, c)]_{t_3}\), but only \([\text{addAfter}(a, b)]_{t_1}\).

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, \([\text{addAfter}(a, b)]_{t_1} \sqcup [\text{remove}(e)]_{t_2}\) says \( \text{addAfter}(a, b) \) and \( \text{remove}(e) \) can be ordered either way. It can also describe a set of conflicting but concurrently issued actions, so that we do not need to enumerate all the possible execution traces. For instance, when the program \( (\text{addAfter}(a, b); || \text{addAfter}(a, c)) \) terminates, we have \([\text{addAfter}(a, b)]_{t_1} \sqcup [\text{addAfter}(a, c)]_{t_3}\).

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

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

\[
([a_i]^P \sqcup [a'_i]^P) \Rightarrow [a_i]^P \times [a'_i]^P \lor [a'_i]^P \times [a_i]^P
\]

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

\[
(s \wedge \text{emp}) \sqcup ([\text{addAfter}(a, b)]_{t_1} \times [\text{addAfter}(a, c)]_{t_3}) \Rightarrow s = \text{acb}
\]

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

\[
(s \wedge \text{emp}) \sqcup ([\text{addAfter}(a, b)]_{t_1} \times [\text{addAfter}(a, c)]_{t_3}) \Rightarrow s = \text{ac} \lor s = \text{acb}
\]

The assertion \( p \triangleright q \) 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 \( Q_e \) all the actions must have arrived at node \( t \)). For instance, the following holds:

\[
(s \wedge \text{emp}) \sqcup ([\text{addAfter}(a, b)]_{t_1} \times [\text{addAfter}(a, c)]_{t_3}) \Rightarrow s = \text{acb}
\]
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 \( \text{Emp} \) says there is no action issued. The assertion \( p \sim [α]^i \) says \( t \) 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 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 \( R = (p’ \sim [α]_{i}) \), 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 \( [α'] \) 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 \( [α'] \) 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 (\( \sim \)) 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, \( p \overset{def}{=} [\text{addAfter}(a, b)]_{1} \) is stabilized to the following \( p_{1} \) under \( R_{1} \), for the RGBA object:

\[
R_{1} \overset{def}{=} \text{addAfter}(a, b)_{1} \sim [\text{addAfter}(a, c)]_{2} \quad (7.1)
\]

\[ p_{1} \overset{def}{=} p \land ([\text{addAfter}(a, b)]_{1} \times [\text{addAfter}(a, c)]_{2}) \]

In the inference rules (see the \text{CALL-R} and \text{LOCAL} rules in Fig. 11), we use the stability check \( \text{Sta}(p, R, \sim) \). It is passed by stabilized assertions only. For (7.1), \( \text{Sta}(p_{1}, R_{1}, \sim) \) holds.

Inference rules. Figure 11 presents the key inference rules. The \text{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 \( \mathcal{P} \land \text{emp} \)). At the end, we derive the state assertion \( Q_{3} \) by receiving all the actions in \( q_{1} \) (see \( q_{1} \Rightarrow Q_{3} \)). 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 \text{CALL} rule, we first compute the return value \( n’ \) of the call, using \( p \overset{μ}{\rightarrow} n’ \), where \( μ ∈ \text{AbsState} → \text{Val} \) is the return value generator of \( Γ(f, n) \). \( p \overset{μ}{\rightarrow} n’ \) 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 assignment rule in Hoare logic. Finally we add the newly generated action \( α \) to the action set in \( q \), and use the resulting assertion \( (q, ⊳) \times [α]_{i} \) as the postcondition. The invocation of \( α \) following \( q \) (i.e., \( q \sim [α]_{i} \)) 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 \text{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 \( (p \overset{μ}{\rightarrow} n’) \) does not hold, we can first apply a standard disjunction rule to branch on \( p \), and apply the \text{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 \( \text{Emp} \) here. To allow a weaker \( R \), we can apply the \text{CSQ} rule to stabilize the postcondition by weakening \( (q, ⊳) \times [α]_{i} \). Then we apply \text{CALL-R} rule, which requires the pre- and post-conditions be stable with respect to \( R \) and satisfy \text{cmnt-closed}. Here \text{cmnt-closed}(p) iff \( p \) is preserved after receiving one or more actions that are already issued in \( p \).

The \text{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 \( t_{5} \) 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 \( a_{k} \) unconditionally. \( G_{t_{2}} \) says that \( t_{2} \) calls \( a_{c} \) after it receives \( a_{b} \). Similarly, \( G_{t_{3}} \) says that \( t_{3} \) calls \( a_{d} \) after it receives \( a_{c} \). Here we write \( \phi(\alpha) \) for \( [α]_{i} \cup \text{true} \).

By the \text{PAR} rule, we only need to verify each thread independently. For thread \( t_{5} \), we first stabilize \( p_{a} \) under \( R_{1} \), resulting in the assertion in Fig. 12. After finding \( c \in v, \) we
we add the extra requirements.

We get the assertion (4). It has the branch

\[ \rightarrow \] variant assertion.

Our logic can be easily ex-

\[ \tau \vdash \{ p \} \varphi \{ q \} \] if \( \tau \vdash \{ p \} \varphi \{ q \} \) is the Hoare triple that uses 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 ex-

\[ \varphi (\Pi, (\Gamma, \Rightarrow)) \Rightarrow \text{ACC}_{\varphi} (\Pi, (\Gamma, \Rightarrow)) \]

Examples. Using Theorem 8, we have verified seven CRDT algorithms [26], 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

\[ \Pi \Rightarrow \text{ACC}_{\varphi} (\Pi, (\Gamma, \Rightarrow)) \]

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 \( \Pi \) in \( C_1 || \ldots || C_n \):

- Provide the specifications for CRDTs. The operation specification \( \Gamma \) is the same as the one for sequential data types. It is also easy to come up with the conflict relation \( \Rightarrow \), which is between all the non-commutative abstract operations in \( \Gamma \).
- 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 prove the proofs following the logic rules.
- Apply the proof method for CRDT implementations. All one needs to do is to provide \( \Rightarrow \) and \( \varphi \), and prove the set of proof obligations. The proof obligations are all first-order formulae. They do not universally quant-

\[ \Rightarrow \in \mathcal{P}(\text{Effector} \times \text{Effector}) \] (the time-stamp order)

\[ \varphi \in \text{LocalState} \rightarrow \mathcal{P}(\text{Effector}) \] (the view function)

The time-stamp order \( \Rightarrow \) is a partial order between effec-

\[ \delta \Rightarrow \delta' \iff \exists a, i, b, a', i', b' \; \delta = \text{AddAft}(a, i, b) \]

\[ \delta = \text{AddAft}(a', i', b') \wedge i < i' \]

\[ \vee \delta' = \text{Rmv}(a) \vee \delta' = \text{Rmv}(b) \]

Here \( \Rightarrow \) 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 \( \Rightarrow \) is specified at the implementation level. One should not confuse it with the won-by order \( \triangleright \) over abstract operations, which we introduce in Sec. 2.4 and Sec. 9.

The view function \( \varphi \) maps each local state \( S \) to a set of effectors that must have been applied before reaching \( S \). 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, \( \varphi \) is instantiated as follows:

\[ \varphi (S) \triangleq \{ \delta \mid \exists a, i, b, (a, i, b) \in S(N) \wedge \delta = \text{AddAft}(a, i, b) \vee \exists a \in S(T) \wedge \delta = \text{Rmv}(a) \} \]

Our proof method, CRDT-TS\( \varphi \)(\( \Pi, (\Gamma, \Rightarrow) \), \( \varphi \), \( \varphi \)) is a con-

- Commutative effectors: the effectors generated by \( \Pi \) are all commutative.
- Same return value: the corresponding operations in \( \Pi \) and \( \Gamma \) have the same return value if executed at \( \varphi \)-related states.
- State correspondence: starting from \( \varphi \)-related states \( S \) and \( S_a \), executing a valid effector \( \delta \) (generated from \( \Pi \)) and the corresponding abstract operation should lead to \( \varphi \)-related states. \( \delta \) is valid if \( \Rightarrow \) does not order it before any \( \delta' \) visible from \( S \), i.e., \( \delta' \in \varphi (S) \).
- Some simple well-formedness checks for \( \Rightarrow \) and \( \varphi \) to ensure the user-specified \( \Rightarrow \) and \( \varphi \) make sense.

Figure 12. Verification of the client with RGA.
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 (↑) and canceled-by (↓) relations. Like ≻ (see Fig. 7), they are also binary relation over actions. The full specification is now a quadruple (Γ, ≻, ▷, ◀).

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

↑ and ▷ can only relate conflicting operations, that is, ◀∉↑ and ▷∈↑. Also ▷ should be valid in that α′ indeed nullifies the effects of α if α ▷ α′. Like ≻, we also overload ↑ and ▷ over operations and events.

We generalize ACC with the extended specification, and define XACC(Π, (Γ, ≻, ▷, ◀)). It requires every trace E of Π to satisfy XACT if causalDelivery(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 e1 happens before another origin event e2, then for any node t the effect of e1 reaches t earlier than that of e2.

Definition 9. XACT_q(Ε, 𝗦, (Γ, ≻, ▷, ◀)) if ∀ar_1, ..., ar_n, 
∀t. totalOrder_visible(Ε,t)(ar_n) ∧ (∀vis ∃t. E_0 ⊆ E ⊆ an)
∧ PresvCancel(ar_n, E, (Γ, ▷)) ∧ ExecRelated_q(t, (Ε, 𝗦), (Γ, ar_n))
∧ ∀t′ ≠ t. RCoH_{t,t′}(ε′, ar_n, ε, (Γ, ▷, ◀))

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_n and ar_{n-1} 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} ⊆ nc-vis(E′_t, (Γ, ▷)) ∩ nc-vis(E′_{t′}, (Γ, ▷))). Moreover, the ordering must follow the won-by order ↑ 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_n and ar_{n-1}.

XACT also requires PresvCancel(an, t, E, (Γ, ▷)). It says, if e_1 is canceled by e_2 and is also visible to e_2 on certain node, the arbitration order ar_1 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 refinement Π ⊆_q (Γ, ≻, ▷, ◀). It is similar to Π ⊆_p (Γ, ≻) (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 add/modify 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 ≻ relation. Their correctness definition computes the dependent cuts of an execution of CRDT, which is similar to our visible(E, 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 $\triangleright$ to relate operations that need to be causally dependent. We use the same symbol $\triangleright$ to relate non-commutative 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. 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 ($\triangleright$) 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 II (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


