Effective verification of Replicated Data Types using Later Appearance Records (LAR)

Madhavan Mukund, Gautham Shenoy R, and S P Suresh

Chennai Mathematical Institute, India
{madhavan,gautshen,spsuresh}@cmi.ac.in

Abstract. Replicated data types store copies of identical data across multiple servers in a distributed system. For the replicas to satisfy strong eventual consistency, these data types should be designed to guarantee conflict free convergence of all copies in the presence of concurrent updates. This requires maintaining history related metadata that, in principle, is unbounded.

While earlier work such as [2] and [9] has concentrated on declarative frameworks for formally specifying *Conflict-free Replicated Data Types* (CRDTs) and conditions that guarantee the existence of finite-state (*distributed*) reference implementations, there has not been a systematic attempt so far to use the declarative specifications for effective verification of CRDTs.

In this work, we propose a simple *global* reference implementation for CRDTs specified declaratively, and simple conditions under which this is guaranteed to be finite. Our implementation uses the technique of *Later Appearance Record* (LAR). We also outline a methodology for effective verification of CRDT implementations using CEGAR.

1 Introduction

Replicated data types are used by web services that maintain multiple copies of the same data across different servers to provide better availability and fault tolerance. Clients can access and update data at any copy. Replicated data types cover a wide class of data stores that include distributed databases and DNS servers, as well as NoSQL stores such as Redis and memcached. The CAP theorem [4] shows that it is impossible for replicated data types to provide both strong consistency and high availability in the presence of network and node failures. Hence, web services that aim to be highly available in the presence of faults opt for a weaker notion of consistency known as *eventual consistency*. Eventual consistency allows copies to be inconsistent for a finite period of time. However, the web service must ensure that conflicts arising due to concurrent updates across multiple copies are resolved to guarantee that all the copies eventually agree. *Conflict-free Replicated Data Types (CRDTs)*, introduced in [11,12], are a subclass of replicated data types that are eventually consistent and conflict free.

An abstract specification of a data type describes its properties independent of any implementation. Such a specification plays a crucial role in formal

verification of the correctness of any implementation of the data type. Most of the early work on CRDTs described these data types through implementations [1,8,11,12]. Recently, a comprehensive framework has been proposed in [2] to provide declarative specifications for a wide variety of replicated data types, along with a methodology to prove the correctness of an implementation via replication aware simulations. Unfortunately this strategy does not lend itself to effective formal verification of the implementations.

In [9], we describe a bounded reference implementation of a CRDT generated from a declarative specification. This construction produces a distributed implementation where each replica only has a local view of the overall computation, obtained through the messages that it receives. This requires an intricate distributed timestamping protocol [7,10] to reuse timestamps in order to bound the implementation. Moreover, strong assumptions about the underlying operating environment have to be directly incorporated into the reference implementation.

The main aim of generating a reference implementation is to come up with an effective verification procedure for generic CRDT implementations. The key observation of this paper is that a *global* reference implementation is sufficient for this purpose. In a global reference implementation, we can directly keep track of causality between update events without exchanging additional information between replicas. In fact, we show that we can maintain a local sequential history for each replica in terms of a later appearance record (LAR) [5], from which we can faithfully reconstruct the causality relation. This greatly simplifies the construction. Moreover, the LAR-based construction is independent of any assumptions on the environment required to bound the size of the reference implementation.

The paper is organized as follows. In the next section, we define CRDTs and introduce declarative specifications. Section 3 describes how the construction of a reference implementation. In the next section, we describe an effective technique for CRDTs using CEGAR [3]. We conclude with a summary and a discussion of future research directions.

2 CRDTs, traces and Specifications

We consider distributed systems consisting of a set $\mathcal{R}$ of N replicas, denoted $[1..N]$. We use p, q, r, s and their primed variants to range over $\mathcal{R}$. These replicas are interconnected through an asynchronous network. We assume that replicas can crash and recover infinitely often. However, when a replica recovers from a crash it is expected to resume operation from some safe state that it was in before the crash. We are interested in replicated data types that are implemented on top of such distributed systems.

A replicated data type exposes a set of side-effect-free operations known as *queries* for clients to obtain information contained in the data type. It makes available a set of state-modifying operations known as *updates* to allow clients to update the contents of the data type. For example, in a replicated set, *contains* is a query method, while *add* and *delete* are update methods.

At any point, a client can interact with any one of the N replicas. The replica that services a query (respectively, update) request from the client is said to be the *source replica* for that query (respectively, update). The source replica uses its local information to process the query. On receiving an update request from the client, the source replica modifies its local state appropriately.

In this paper, we restrict our attention to a class of replicated data types called *Conflict-free Replicated Data Types (CRDTs)*, introduced in [11]. In these data types, each time a replica receives an update request from a client, it applies the update locally and broadcasts to all the other replicas a message containing the data that they require to apply this update. On receiving this broadcast, each replica performs a local update using the data sent by the source replica. We now define some terminology from [11,12] to reason about these data types.

A CRDT $\mathcal{D}$ is a tuple $(\mathcal{V}, \mathcal{Q}, \mathcal{U}, \mathsf{Ret})$ where:

- $\mathcal{V}$ is the underlying set of values stored in the datatype and is called the *universe* of a replicated datatype. For instance, the universe of a replicated read-write register is the set of integers that the register can hold.
- $\mathcal{Q}$ denotes the set of query methods exposed by the replicated data type.
- $\mathcal{U}$ denotes the set of update methods.
- Ret is the set of all return values for queries.

We assume that $\perp$ is a designated "empty value", belonging to both $\mathcal{V}$ and Ret.

Definition 1 (Operations) *An operation of a CRDT $\mathcal{D} = (\mathcal{V}, \mathcal{Q}, \mathcal{U}, \mathsf{Ret})$ is a tuple $o = (m, r, args, ret)$ where $m \in \mathcal{Q} \cup \mathcal{U} \cup \{\textbf{receive}\}$ is the action, $r \in \mathcal{R}$ is the source replica, args is a tuple of arguments from $\mathcal{V}$, and $ret \in \mathsf{Ret}$ is the return value, satisfying the following conditions:*

- *if $m \in \mathcal{U}$, $ret = \perp$.*
- *if $m = \textbf{receive}$, $args = ret = \perp$.*

For an operation $o = (m, r, args, ret)$, we define $Op(o) = m$, $Args(o) = args$, $Rep(o) = r$, and $Ret(o) = ret$. We call o a query operation *if $m \in \mathcal{Q}$, an* update operation *if $m \in \mathcal{U}$ and a* receive operation *if $m = \textbf{receive}$.*

We denote the set of operations of $\mathcal{D}$ by $\Sigma(\mathcal{D})$.

Definition 2 (Run) *A* run *of a replicated data type is a pair (ρ, φ) where*

- *ρ is a sequence $o_1 o_2 \dots o_n$ of operations from $\Sigma(\mathcal{D})$.*
- *φ is a partial function from $[1..n]$ to $[1..n]$ such that*
  - *$dom(\varphi) = \{i \leq n \mid o_i \text{ is a receive operation}\}$.*
  - *if $\varphi(i) = j$ then $j < i$, o_j is an update operation and $Rep(o_i) \neq Rep(o_j)$.*

For a sequence $\rho = o_1 o_2 \cdots o_n$, we denote by $\rho[i]$ the operation o_i, and we denote by $\rho[i:j]$ the subsequence $o_i o_{i+1} \cdots o_j$.

Definition 3 *Let (ρ, φ) be a run with $\rho = o_1 \cdots o_n$. An update operation o_i is said to be* delivered *if $(\forall r \in \mathcal{R})\big[r \neq Rep(\rho[i]) \implies (\exists j)[r = Rep(\rho[j]) \wedge \varphi(j) = i]\big]$.*

Definition 4 (Events) *Let (ρ, φ) be a run of a replicated data type. We associate an event with each update and receive operation performed in ρ. Formally, the set $\mathcal{E}_\rho$ is a set of* events *associated with the operations in ρ given by*

$$\mathcal{E}_\rho = \{e_i \mid 1 \leq i \leq |\rho|, Op(\rho[i]) \in \mathcal{U} \cup \{\mathbf{receive}\}\}.$$

Each $e_i \in \mathcal{E}_\rho$ corresponds to the operation $\rho[i]$ in ρ. We define $Rep(e_i)$, $Op(e_i)$ and $Args(e_i)$ to be $Rep(\rho[i])$, $Op(\rho[i])$ and $Args(\rho[i])$.

We extend φ to $\mathcal{E}_\rho$ as follows. For $e_i \in \mathcal{E}_\rho$, let $\rho[i]$ be the corresponding event in ρ. Then, $\varphi(e_i) = e_j$ if $\varphi(\rho[i]) = j$.

Definition 5 (Happened before) *For a run (ρ, φ) and a replica r, we denote by $\mathcal{E}_\rho^r$ the set of r-events $\{e \in \mathcal{E}_\rho \mid Rep(e) = r\}$. The total order $\{(e_i, e_j) \mid e_i, e_j \in \mathcal{E}_\rho^r, i < j\}$ is denoted by $\leq_\rho^r$. We denote by $\leq_\rho^{\mathbf{receive}}$ the relation $\{(\varphi(e), e) \mid e \in \mathcal{E}_\rho, Op(e) = \mathbf{receive}\}$.*

The happened before *relation on (ρ, φ), denoted $\preceq_\rho$, is defined by*

$$\bigcup_{r \in \mathcal{R}} (\leq_\rho^{\mathbf{receive}} \cup \leq_\rho^r)^+$$

For a pair of update events e, e' we say that e has happened before *e' if $e \preceq_\rho e'$. We say that a pair of events $e, e' \in \mathcal{E}$ are* concurrent *(denoted by $e \parallel_\rho e'$) when neither $e \preceq_\rho e'$ nor $e' \preceq_\rho e$ holds.*

The definition of $\preceq_\rho$ is subtle. If a replica r receives information about an update at r', r continues to know about this update even after it performs more local actions. But r does not necessarily know about events at r' prior to this update. Hence, $\preceq_\rho$ is not transitive, though it is always acyclic. If we have a strong delivery criterion like *causal delivery* along with the assumption that each update is broadcast to every replica, then one can show that $\preceq_\rho$ is transitive.

We now define the trace associated with a run.

Definition 6 (Trace) *The* trace *associated with a run (ρ, φ) is the triple $(\mathcal{E}_\rho, \varphi, \preceq_\rho)$. (The term* trace *is borrowed from Mazurkiewicz trace theory [6]). We denote the trace of a run (ρ, φ) by trace(ρ, φ). The set of all traces is denoted by $\mathcal{T}$.*

Given a trace $(\mathcal{E}, \varphi, \preceq)$ and a subset of events $X \subseteq \mathcal{E}$, the subtrace induced by X is given by $(X, \varphi_X, \preceq_X)$, where φ_X and $\preceq_X$ are the obvious restrictions of φ and $\preceq$ to the set X.

Definition 7 (View) *Let $t = (\mathcal{E}, \varphi, \preceq)$ be a trace. For a replica $r \in \mathcal{R}$, the maximal r-event in t is denoted by $max_r(t)$. The* view *of r in t, denoted $\partial_r(t)$, is the subtrace induced by the subset $\mathcal{E}' = \{e' \in \mathcal{E} \mid e' \preceq max_r(t)\}$.*

Definition 8 (Declarative Specification and Permitted Runs) *Let $\mathcal{D} = (\mathcal{V}, \mathcal{Q}, \mathcal{U}, \mathsf{Ret})$ be a CRDT. A declarative specification of $\mathcal{D}$ is a function $f : \mathcal{T} \times \mathcal{Q} \times \mathcal{V}^* \to \mathsf{Ret}$ that determines the return value of any query $q \in \mathcal{Q}$ with arguments $args \in \mathcal{V}^*$ in a trace t.*

If f is a declarative specification of $\mathcal{D}$, the set of permitted runs of $\mathcal{D}$, denoted $Runs(\mathcal{D}, f)$, consists of all $\mathcal{D}$-runs (ρ, φ) such that for all query operations $\rho[i] = (q, r, args, ret)$, $ret = f(\partial_r(trace(\rho[i], \varphi)), q, args)$.

If a CRDT is specified declaratively, all responses to queries are determined by the trace generated by a run, and not the specific interleaving of operations in the run. Even this is an overkill—typically, the response to a query is determined not by the entire trace but by the subtrace generated by a set of relevant events whose size is bounded, independent of the length of the trace. Further, this set can usually be computed easily. We now formalize this intuition.

Definition 9 (Computable specification) *Let $\mathcal{D}$ be a CRDT and f be a declarative specification of $\mathcal{D}$. f is said to be computable if there exist computable functions $g : \mathcal{T} \times \mathcal{Q} \times \mathcal{V}^* \to \mathcal{T}$ and $h : \mathcal{T} \times \mathcal{Q} \times \mathcal{V}^* \to \mathsf{Ret}$ such that:*

- *$g(t, q, args)$ is a subtrace of t containing only update events.*
- *$f(t, q, args) = h(g(t, q, args), q, args)$.*
- *If $g(t, q, args) \subseteq t' \subseteq t$ then $g(t', q, args) = g(t, q, args)$.*
- *If t and t' are isomorphic, $h(t, q, args) = h(t', q, args)$.*

In such a situation, we say that f is computable via g and h.

The subtrace $g(t, q, args)$ can be thought of as the relevant information *needed to compute $f(t, q, args)$. The function h computes the desired value of f using the subtrace identified by g. The third condition captures a monotonicity constraint: information that has become irrelevant now will never reappear as relevant information later.*

Example 10. OR-Set [1,8,11] is a CRDT implementation of sets. The operations are given by $\mathcal{D}_{OR-Set} = (\mathcal{V}, \{\mathbf{contains}\}, \{\mathbf{add}, \mathbf{delete}\}, \{True, False\})$.

The main issue is resolving concurrent **add** *and* **delete** *operations. In OR sets,* **add** *wins in such a situation, so* **contains** *returns true.*

The declarative specification f capturing this behaviour, given via computable functions g and h, is defined as follows:

- *$(\forall x \in \mathcal{V})(\forall t \in \mathcal{T})$ $g(t, \mathbf{contains}, x)$ is the set of maximal events in the subtrace t_x of t where $t_x = \{e \mid Op(e) \in \{\mathbf{add}, \mathbf{delete}\} \wedge Args(e) = x\}$.*
- *$(\forall x \in \mathcal{V})(\forall t \in \mathcal{T})$ $h(t, \mathbf{contains}, x)$ is True iff there is a maximal event e of t with $Op(e) = \mathbf{add}$ and $Args(e) = x$.*

Definition 11 (Bounded specification) *If a specification function f is computable via g and h and there is a bound K such that $|g(t, q, args)| \leq K$ for all t, q and $args$, we say that f is a bounded specification (with bound K).*

Example 12. The specification of OR-Sets provided in Example 10 is bounded with a bound $N = |\mathcal{R}|$ since $g(t, \mathbf{contains}, x)$ contains the maximal x-events and there can be at most one maximal x-event in $g(t, \mathbf{contains}, x)$ per replica.

3 CRDT Implementation

Recall that a run is a pair (ρ, φ) where ρ is a sequence of operations of $\mathcal{D}$ and φ is a function that identifies the update (at a remote replica) corresponding to each receive operation in ρ. When we consider an implementation of a CRDT, its runs will typically be just sequences of operations. The function φ is not provided along with the run, but it is reasonable to assume that the implementation has enough extra information to identify the update operation corresponding to each receive event. One way to model this abstractly is to timestamp each operation by a natural number and assign the same timestamp to a receive and its matching update. Since we are interested in finite-state CRDT implementations also, we would like to use a bounded linearly ordered set ID of identifiers as timestamps. It is simplest to assume that $ID \subseteq \mathbb{N}$.

For a time-stamped operation $o' = (o, id) \in \Sigma(\mathcal{D}) \times ID$, we define $Id(o') = id$ and $\psi(o') = \psi(o)$ for $\psi() \in \{Rep(), Op(), Ret(), Args()\}$.

We say that a timestamped run $\rho' \in (\Sigma(\mathcal{D}) \times ID)^*$ is *well-formed* if timestamps are assigned sensibly, as follows.

- for every receive operation $\rho'[j]$, there is $i < j$ such that $Id(\rho'[i]) = Id(\rho'[j])$, $Op(\rho'[i]) \in \mathcal{U}$ and for all $k \in [i+1..j-1]$,

$$Op(\rho'[k]) = \textbf{receive} \implies Rep(\rho'[k]) \neq Rep(\rho'[j]) \vee Id(\rho'[k]) \neq Id(\rho'[j]).$$

- For $i < j$, if $\rho'[i]$ and $\rho'[j]$ are update operations and $Id(\rho'[i]) = Id(\rho'[j])$, then for every replica $r \neq Rep(\rho'[i])$, there is a $k \in [i+1..j-1]$ such that $Op(\rho'[k]) = \textbf{receive}$, $Rep(\rho'[k]) = r$ and $Id(\rho'[k]) = Id(\rho'[i])$.

The first condition captures the fact that timestamps unambiguously match receive events to update operations. The second condition prevents a timestamp from being reused before it has been received by all replicas.

The run associated with a well-formed timestamped run $\rho' = ((o_1, \ell_1), (o_2, \ell_2), \ldots, (o_m, \ell_m))$ is a pair (ρ, φ) such that $\rho = o_1 o_2 \cdots o_m$ and for any $i \leq |\rho'|$, if o_i is a receive operation, $\varphi(i) = \max\{j < i \mid \ell_j = \ell_i \text{ and } Op(o_j) \in \mathcal{U}\}$.

In what follows, we consider only well-formed timestamped runs.

Lemma 13. *For every run (ρ, φ) of $\mathcal{D}$, we can identify a set ID such that there is a well-formed timestamped run $\rho' \in (\Sigma(\mathcal{D}) \times ID)^*$ whose associated run is (ρ, φ).*

Proof. All query operations can be labelled with a fixed identifier (say 0, for concreteness). Each update operation $\rho[i]$ is labelled with the smallest identifier in ID that does not label any undelivered update operation in $\rho[1 : i-1]$. Every receive operation $\rho[i]$ is labelled by the same identifier that labels $\rho[j]$, where $\varphi(i) = j$.

Definition 14 (CRDT Implementation and its runs) *An implementation of a CRDT $\mathcal{D}$ is a tuple $\mathcal{D}_I = (S, s^0, ID, \rightarrow)$ where:*

– S *is set the global states.*
– $s^0 \in S$ *is the initial state.*
– $ID \subseteq \mathbb{N}$ *is the set of identifiers, which serve as timestamps.*
– $\rightarrow \subseteq S \times (\Sigma(\mathcal{D}) \times ID) \times S$ *is the transition relation.*

A timestamped run $\rho' = o'_1 \cdots o'_n$ *is accepted by* $\mathcal{D}_I$ *if there exists a sequence of states* $s_0 s_1 \cdots s_n$ *such that* $s_0 = s^0$*, and for every* $i \leq n$*,* $s_{i-1} \xrightarrow{o'_i} s_i$*.* (ρ, φ) *is a run of* $\mathcal{D}_I$ *if it is the run associated with a well-formed timestamped run* ρ' *accepted by* $\mathcal{D}_I$*. We denote the set of all runs of* $\mathcal{D}_I$ *by* $Runs(\mathcal{D}_I)$*.*

Definition 15 (Correctness of a CRDT Implementation) *Let* $\mathcal{D}$ *be a CRDT with declarative specification* f*. An implementation of CRDT* $\mathcal{D}_I$ *is correct if* $Runs(\mathcal{D}_I) \subseteq Runs(\mathcal{D}, f)$*.*

We now present a canonical implementation of a CRDT $\mathcal{D} = (\mathcal{V}, \mathcal{U}, \mathcal{Q}, \mathsf{Ret})$ with a declarative specification f. The canonical implementation, denoted $\mathcal{D}_{ref}$, satisfies the property that $Runs(\mathcal{D}_{ref}) = Runs(\mathcal{D}, f)$.

3.1 Reference Implementation

Before we describe the reference implementation, we present the ingredients needed. The aim is to maintain as little information as possible to respond to each query. The key observation is that the reference implementation is *global*—it can pool together information stored at all replicas without paying the cost of synchronization. If we have a declarative specification f of $\mathcal{D}$ that is computable via g and h, then each replica needs to maintain $\bigcup_{q,args} g(t, q, args)$, where t is the view of r at any point in time. The important ingredient in g is the precedence relation between events, and hence the reference implementation needs to store enough information to recover this. The implementation also needs to intelligently discard information that will no longer prove useful.

The most direct implementation would store (as part of the "state" of each replica) the relevant suffix of the trace—the upward closure of the events in $\bigcup_{q,args} g(t, q, args)$. But we choose a more compact representation called *Later Appearance Records* (LARs), from which the information needed to answer queries can be recovered. An LAR is a set of sequences rather than a partial order, and hence easier to manipulate.

Let $\mathcal{L}$ be a (potentially infinite) set of labels, equipped with a total order $\leq$. We use labels to distinguish between multiple occurrences of the same update method at the same replica with the same arguments. Operations equipped with labels are called nodes.

Definition 16 (Node) *A* node *is a tuple* $(u, r, args, l) \in \mathcal{U} \times \mathcal{R} \times \mathcal{V}^* \times \mathcal{L}$*. For* $v = (u, r, args, l)$*, we define* $Op(v) = u$*,* $Rep(v) = r$*,* $Args(v) = args$ *and* $Label(v) = l$*. The set of all nodes is denoted by* $\mathcal{N}$*.*

Definition 17 (Later Appearance Record) *A Later Appearance Record (LAR) is a sequence of distinct nodes. For a node v and an LAR A, we write $v \in A$ to denote that v appears in the sequence of nodes in A.*

For nodes $v_1, v_2 \in A$, $v_1 \leq_A v_2$ if v_1 occurs earlier than v_2 in A. If A is an LAR and V is a set of nodes then $A - V$ is the subsequence of A consisting of nodes not in V. The set of all LARs is denoted by $\mathcal{A}$.

Each replica uses the LAR to record the order in which it has seen updates, originating locally as well as remotely. In an actual implementation, updates are generated at replicas, and information about them is passed to other replicas by the network, whose behaviour is not under the control of the implementation. But it is assumed that when a replica receives information about an update, it can determine which update is being mentioned. The network might sometimes provide additional guarantees about message delivery (such as *causal delivery* or *FIFO delivery*), and we can sometimes make use of these facts to simplify the implementation. Here we present the general case, without any assumptions about the network.

When information about an update has been passed to all other replicas, we would like to be able to discard this information from every replica. For this, it becomes important to record the set of replicas to which information about an update has been communicated. This is modelled using a *network node*. Recall that a node is an update operation along with an identifying label. A network node attaches to a node a timestamp as well information about the state of replicas that have received the update.

Definition 18 (Network node) *A network node is a member of $\mathcal{N} \times ID \times 2^{\mathcal{R}}$. The set of all network nodes is denoted by $\mathcal{N}_{net}$. For a network node $v_{net} = (v, id, R)$ we define $Node(v_{net})$ to mean v, $Id(v_{net})$ to mean id and define $Rep(v_{net})$, $Id(v_{net})$, $Args(v_{net})$ and $Label(v_{net})$ to be the corresponding functions applied on v. We use $Delivered(v_{net})$ to denote R.*

A configuration consists of the LAR of each replica along with the network nodes pertaining to undelivered updates. The aim is to try to purge nodes from LARs whenever possible. A *consistent configuration* is one where these purges have been done safely. Specifically, replica r does not purge a node pertaining to a local update so long as it is present in the LAR of some other replica. Also, if information about a local update has not yet been communicated to all other replicas, r does not purge the corresponding node.

Definition 19 (Configuration) *A configuration C is a member of $\mathcal{A}^{\mathcal{R}} \times 2^{\mathcal{N}_{net}}$. For any configuration $C = ((A_1, A_2, \ldots, A_N), V_{net})$, we denote by $C[r]$ the LAR A_r. We shall denote by C_{net} the set of network nodes V_{net}.*

We say that a configuration C is consistent iff

- *$\forall r, r'$ if there exists $v \in C[r]$ such that $Rep(v) = r'$ then $v \in C[r']$.*
- *$\forall v_{net} \in C_{net}$ if $r \in Delivered(v_{net})$ then $Node(v_{net}) \in C[r]$.*

The trivial configuration denoted by C^0 is one where $\forall r \in \mathcal{R} : C^0[r]$ is the empty LAR and $C^0_{net} = \emptyset$. We denote the set of all consistent configurations by $\mathcal{C}$.

Using the LARs of all the replicas, we can reconstruct the happened before relation for all events that are mentioned in a configuration. Suppose r sees two updates u' and u'' originating at r' and r''. Since updates are seen at the originating replica first before being seen by others, the relation between u' and u'' can be determined by their relative order of appearances in the LARs of r' and r''. Here we crucially use the fact that our implementation is *global*.

Definition 20 (Precedence and Concurrency) *Let C be a consistent configuration. Let r be a replica and $v_i, v_j \in C[r]$ with $Rep(v_i) = r'$ and $Rep(v_j) = r''$. We say that v_i* precedes *v_j in C, denoted by $v_i \leq_C v_j$, if $(v_i \in C[r''] \wedge v_i \leq_{C[r'']} v_j) \wedge (v_j \in C[r'] \implies v_i \leq_{C[r']} v_j)$. (In other words, both r' and r'' locally see v_i before v_j.)*

If neither $v_i \leq_C v_j$ nor $v_j \leq_C v_i$ for any $v_i, v_j \in C[r]$, then we say that v_i and v_j are concurrent in C, denoted by $v_i \parallel_C v_j$.

For a consistent configuration C and replica r, the view of r in C, denoted by $\partial_r(C)$, is the trace $(C[r], \leq_C)$.

If a node in a trace t contains information about an update that is in $g(t, q, args)$ for a query $q(args)$, then that node cannot be purged—otherwise the response to that query would be inaccurate. This is formalized below.

Definition 21 (Relevant node) *Let f be a specification of $\mathcal{D}$ computable via g and h. We say that a node v in a consistent configuration C is relevant with respect to f if there exists a replica r, query $q \in \mathcal{Q}$ and $args \in \mathcal{V}^*$, such that $v \in g(\partial_r(C), q, args)$.*

3.2 Details of the reference implementation

The reference implementation is formally presented below. Each replica maintains an LAR to which it appends information pertaining to each local update. On receiving information about a remote update, it again appends this to the LAR, and also seeks to purge from all LARs nodes that have ceased to become relevant and have been seen by all replicas. This enables the reuse of labels. Since at any trace t the relevant nodes subsume all subtraces of the form $g(t, q, args)$, it follows that the implementation never purges information that is needed to answer a query.

Let f be a specification of a CRDT $\mathcal{D}$ computable via g and h. Its reference implementation is defined to be $\mathcal{D}_{ref} = (\mathcal{C}, C_0, ID, \rightarrow_{ref})$ where $ID = \mathbb{N}$ and $\rightarrow_{ref}$ is defined as follows.

Let $C, C' \in \mathcal{C}$ and let $o = ((m, r, args, ret), id) \in \Sigma(D) \times ID$. Then $C \xrightarrow{o}_{ref} C'$ iff one of the following holds:

- $m \in \mathcal{Q}$ and $ret = f(\partial_r(C), m, args)$ and $C' = C$.

– $m \in \mathcal{U}$, $\forall v_{net} \in C_{net} : Id(v_{net}) \neq id$, and C' is defined as follows:
  - $\forall r' \in \mathcal{R} : r' \neq r \implies C'[r'] = C[r']$.
  - $C'[r] = C[r].v$, with $v = (m, r, args, l)$ where l is a label such that $\forall v' \in C[r] : Label(v') \neq l$.
  - $C'_{net} = C_{net} \cup \{(v, id, \{r\})\}$.
– $m =$ **receive** and there exists a node v and $R \subseteq \mathcal{R}$ such that $(v, id, R) \in C_{net}$ and $r \notin R$, and C' is defined as follows:
  Let C'' be a configuration given by
  - $\forall r' \neq r : C''[r'] = C[r']$.
  - $C''[r] = C[r].v$.
  - $C''_{net} = C_{net} \cup \{(v, id, R \cup \{r\})\} \setminus \{(v, id, R)\}$.

  If $R \cup \{r\} \neq \mathcal{R}$ then $C' = C''$ else
  - $\forall r' \in \mathcal{R} : C'[r'] = C''[r'] - V$, where

$$V = \{v \in \bigcap_{r' \in \mathcal{R}} C''[r'] \mid v \text{ is not relevant in } C''\}.$$

  - $C'_{net} = C''_{net} \setminus \{(v, id, R \cup \{r\})\}$.

3.3 Correctness of the reference implementation

Lemma 22. *Every reachable configuration C of $\mathcal{D}_{ref}$ is consistent.*

Proof. The initial configuration is trivially consistent, and each transition purges only those nodes that are no longer relevant and are delivered to every replica. This proves the lemma.

Lemma 23. *Suppose $\rho' \in (\Sigma(\mathcal{D}) \times ID)^*$ is accepted by $\mathcal{D}_{ref}$ and that $C_0 \xrightarrow{\rho'}_{ref} C$. Let (ρ, φ) be the run associated with ρ' and $t = trace(\rho, \varphi)$. Then, for all r, q and $args$, $g(\partial_r(t), q, args)$ is isomorphic to $g(\partial_r(C), q, args)$.*

Proof. The proof is by induction on the length of ρ'. The case when $\rho' = \varepsilon$ is trivial. So let $\rho' = \sigma'.o$. Let C' be a configuration such that $C_0 \xrightarrow{\sigma'}_{ref} C' \xrightarrow{o}_{ref} C$. Let (σ, φ) be the run corresponding to σ' and let $t' = trace(\sigma, \varphi)$. We assume by the induction hypothesis that for all r, q and $args$, $g(\partial_r(t'), q, args)$ is isomorphic to $g(\partial_r(C'), q, args)$. There are three cases to be considered.

o **is a query operation:** In this case $C = C'$ and $t = t'$, so the lemma follows.

o **is an update operation:** Suppose $Rep(o) = r$. For $r' \neq r$, it is clear from the transition rules that $C[r'] = C'[r']$. It is also the case that $\partial_{r'}(t) = \partial_{r'}(t')$, so the lemma still holds for queries at replicas other than r.
On the other hand, $C[r] = C'[r].v$ where v is a node with a fresh id, corresponding to o. Since v is the latest node in $C[r]$ and $v \notin C[r']$ for any other r', it is clear that $v' \leq_C v$ iff $v' \in C[r]$. But $v' \in C[r]$ iff v' corresponds to an update received by r or originating in r. Thus $\partial_r(C) = \partial_r(C') \cup \{v\}$, with v as the largest element. It is easy to see that the maximal r-event in the trace t is greater than all other events in $\partial_r(t')$. Thus $g(\partial_r(C), q, args)$ is isomorphic to $g(\partial_r(t), q, args)$.

o **is a receive operation:** Suppose $Rep(o) = r$. We add a node at the end of $C[r]$, but also purge all the LARs of some irrelevant nodes (those that are received by every replica). Since irrelevant nodes do not feature in $g(\partial_{r'}(t), q, args)$ for any r' and $q(args)$, all we need to show is that the order among relevant nodes is captured correctly. But the order between update events does not change at the point of time of a receive. It can be checked that $\leq_C = \leq_{C'}$, and thus the lemma follows.

Lemma 24. *Suppose a well-formed timestamped run $\rho' \in (\Sigma(\mathcal{D}) \times ID)^*$ is accepted by $\mathcal{D}_{ref}$. Let (ρ, φ) be the run associated with ρ'. Then $(\rho, \varphi) \in Runs(\mathcal{D}, f)$.*

Proof. Suppose $C_0 \xrightarrow{\rho'}_{ref} C$. Let $t = trace(\rho, \varphi)$. Since $g(\partial_r(C), q, args)$ is isomorphic to $g(t, q, args)$ and since h returns the same values on isomorphic traces, it easily follows that for all query operations $\rho[i] = (q, r, args, ret)$, $ret = f(\partial_r(trace(\rho[i], \varphi)), q, args)$. Thus $(\rho, \varphi) \in Runs(\mathcal{D}, f)$.

Lemma 25. *Suppose $(\rho, \varphi) \in Runs(\mathcal{D}, f)$. Let $\rho' \in (\Sigma(\mathcal{D}) \times ID)^*$ be a well-formed timestamped run whose associated run is (ρ, φ). Then ρ' is accepted by $\mathcal{D}_{ref}$.*

Proof. We prove the lemma for $\rho'[1 : i]$, by induction on i. The base case, when $i = 0$ is trivial. So let $i > 0$. Suppose $\rho'[1 : i - 1]$ is accepted by $\mathcal{D}_{ref}$ by an execution ending in configuration C. Let (σ, φ) and (σ', φ) be the runs associated with $\rho'[1 : i - 1]$ and $\rho'[1 : i]$ respectively. Let $t = trace(\sigma, \varphi)$ and $t' = trace(\sigma', \varphi)$. Let $o = \rho'[i] = ((m, r, args, ret), id)$. There are three cases to consider.

$m \in \mathcal{Q}$**:** In this case $t = t'$. We know that $ret = f(\partial_r(t'), m, args) = f(\partial_r(t), m, args)$. But we also know that $g(\partial_r(C), m, args)$ is isomorphic to $g(\partial_r(t), m, args)$. Thus it follows that $ret = f(\partial_r(C), m, args)$. Hence $C \xrightarrow{o} C$ and $\rho'[1 : i]$ is accepted by $\mathcal{D}_{ref}$.

$m \in \mathcal{U}$**:** Since $\rho'[1 : i]$ is well-formed, it has to be the case that either id is not used in $\rho'[1 : i-1]$, or if it is used in an update operation $\rho'[j]$, every replica has received that update in $\rho'[j+1 : i-1]$. Thus, there is no node $v_{net} \in C_{net}$ with $Id(v_{net}) = id$. So, o is enabled at C and $\rho[1 : i]$ is accepted by $\mathcal{D}_{ref}$.

$m =$ **receive:** Since $\rho'[1 : i]$ is well-formed, it has to be the case that there is an earlier update at some other replica with the same identifier that has not yet been communicated to r. Thus there exists a node v and $R \subseteq \mathcal{R}$ such that $(v, id, R) \in C_{net}$ and $r \notin R$. It follows that o is enabled at C and $\rho'[1 : i]$ is accepted by $\mathcal{D}_{ref}$.

From the previous two lemmas we can conclude the following:

Theorem 26. $Runs(\mathcal{D}_{ref}) = Runs(\mathcal{D}, f)$

3.4 Bounding the reference implementation

For effective verification, we need to ensure that the set of traces of the CRDT has a finite representation. The reference implementation constructed in the previous section is not necessarily finite-state. The unboundedness arises due to several reasons.

- If the size of the universe is not bounded, the number of nodes, and hence the number of configurations, will not be bounded.
- If there is no bound on the number of undelivered messages, then the number of network states would be unbounded, and therefore the size of C_{net} of any configuration C is unbounded.
- If the specification of the CRDT itself is not finite, then the number of relevant nodes in the configuration is unbounded, even when the universe $\mathcal{V}$ is finite.

With some reasonable assumptions, we can ensure that the reference implementation is finite-state.

1. **Universe Size:** We assume that the size of the universe is bounded by a parameter m. This is a reasonable assumption since most CRDT implementations treat the elements of the universe in a uniform manner. Hence for the purpose of verification, it suffices to consider a universe whose size is bounded.
2. **Delivery Constraints:** We assume that the number of undelivered messages in the network is bounded by the parameter b. Again, this is a reasonable assumption since most practical implementations of strong eventual consistency also requires that messages are reliably delivered to all the replicas. We can pick a sufficiently large b that correctly characterizes the network guarantee of the actual implementation.
3. **Bounded Specification:** We assume that the specification function f computable via g and h comes with a bound K. Let k be the maximum arity of any $q \in \mathcal{Q}$. If the universe if bounded, the number of query instances is bounded by $|\mathcal{Q}| \times m^k$. Since the specification function has a bound K, the size of the relevant nodes in a configuration is bounded by $\ell = K \times |\mathcal{Q}| \times m^k$. For example, in case of OR-sets, to answer the query **contains**(x) it sufficies to keep track of the maximal x-events. Since the number of replicas $\mathcal{R}$ is bounded by N the number of maximal x-events is bounded by N. Hence if the universe is bounded by m then the number of relevant nodes in a configuration is no more than $m \cdot N$.

We now prove that, with these assumptions, the size of the reference implementation is bounded. Each configuration of $\mathcal{D}_{ref}$ consists of an LAR for each replica, and a set of network nodes. As is clear from the transition rules, the only network nodes we retain are those that are still undelivered to some replicas. Thus, if there is a bound on the number of undelivered messages, there is

also a bound on the number of network nodes present in each configuration. But the set of network nodes that occur in all configurations might still be unbounded. To bound this, we need to bound the set of all nodes and the set ID. The size of the set ID can be bounded by b, the number of undelivered messages, as explained below.

Let C be a reachable configuration of $\mathcal{D}_{ref}$ and o an update operation enabled at C. Now it has to be the case that only if there are at most $b-1$ network nodes in C_{net} (otherwise, there would be more than b undelivered messages in the run upto and including o). Thus as long ID has b elements, the reference implementation can always attach a fresh timestamp to o. (Formally this means that we can map any timestamped run of $\mathcal{D}_{ref}$ to an equivalent run which uses at most b timestamps.)

We now turn to bounding the set of all nodes. The only unbounded component in this is the set $\mathcal{L}$ of labels.

Lemma 27. *If the number of undelivered messages is bounded by b and the number of relevant events is bounded by ℓ then it is sufficient to have a label set $\mathcal{L}$ of size $b+\ell$.*

Proof. Let $\rho = \rho'.o$ be any run of the reference implementation such that the number of undelivered messages in ρ is bounded by b. Let o be an update operation at replica r. Let C' be the configuration of the reference implementation at the end of ρ'.

Note that the number of undelivered update operations in ρ' is strictly less than b; otherwise, ρ would have more than b undelivered messages. It follows that the number of undelivered nodes in C' is at most $b-1$. (A node v is undelivered in C' if $(v, R) \in C'_{net}$ for some $R \subseteq \mathcal{R}$.) A node v is present in some LAR $C'[r']$ if v is undelivered or v is relevant. Thus the number of distinct nodes in C' is at most $b+\ell-1$. Thus if $|\mathcal{L}| = b+\ell$, there is at least one free label in $\mathcal{L}$ to label the new node $C[r] \setminus C'[r]$. Thus, it is sufficient to have a label set $\mathcal{L}$ of size $b+\ell$.

From the above, we can conclude that the number of nodes in $\mathcal{N}$ is bounded by $|\mathcal{U}| \times N \times m^{k'} \times (b+\ell)$ (where, as before, k' is the maximum arity of any $u \in \mathcal{U}$).

Since the set ID is also bounded (by b, as already explained), the set of network nodes is bounded (by $|\mathcal{N}| \times |ID| \times 2^N$).

From Lemma 27 it is clear that the number of distinct nodes in any configuration cannot exceed $b+\ell$. Since the number of undelivered messages are bounded by b, the number of network nodes is bounded by b. Thus, the set of all configurations $\mathcal{C}$ is bounded as follows:

$$|\mathcal{C}| \leq |\mathcal{N}|^{(b+\ell)} \times |\mathcal{N}_{net}|^b.$$

Theorem 28. *If the number of undelivered messages and the size of the universe are bounded and we have a bounded specification for the CRDT, then the reference implementation is bounded.*

4 Effective verification using Bounded Reference Implementation via CEGAR

Verifying CRDT implementations is a challenging task. For instance, consider an implementation that uses a bounded set of timestamps as we have proposed, except that the size of this set is too small. Under certain circumstances, a replica may be forced to reuse a timestamp even when a previous update with the same timestamp has not been delivered. To detect such an error, we have to explore a run that exceeds the bound in the implementation. Unfortunately, we typically do not have access to the internal details of the implementation, so this bound is not known in advance. This results in an unbounded verification task.

Alternatively, we have seen that by making reasonable restrictions on the universe of the datatype and the behaviour of the underlying messasge delivery system, we can generate a bounded reference implementation. Once we have such a bounded reference implementation, we can use Counter Example Guided Abstract Refinement (CEGAR) [3] to effectively verify a given CRDT implementation with respect to the assumptions made on the environment.

More formally, given a implementation of a CRDT with bounded specification, let us assume suitable bounds on the size of the universe, m, and the number of undelivered messages, b. We fix the bounded set of timestamps ID accordingly. We assume the existence of an abstraction function that provides a finite state abstraction $\mathcal{D}_I = (S_I, s^0, ID, \rightarrow_I)$ of the implementation, whose runs are in $(\Sigma(D) \times ID)^*$.

We then construct the synchronous product $\mathcal{M}_{sync} =$
$((S_I \times \mathcal{C}) \cup \{s_{err}\}, (s^0, C^0), ID, \rightarrow_{sync})$, where $\rightarrow_{sync}$ is defined as follows:

- The action $o \in \Sigma(\mathcal{D}) \times ID$ is enabled at the product state (s, C) iff o is enabled at s in $\mathcal{D}_I$. If o is enabled then we define
  - $(s, C) \xrightarrow{o}_{sync} s_{err}$, if o is not enabled at C in $\mathcal{D}_{ref}$
  - $(s, C) \xrightarrow{o}_{sync} (s', C')$, if $s \xrightarrow{o}_I s'$ and $C \xrightarrow{o}_{ref} C'$.
- $\forall o \in \Sigma_{\mathcal{D}} : o$ is not enabled at s_{err}

Lemma 29. *If ρ is a run of $\mathcal{M}_{sync}$ resulting in the state s_{err} starting from the initial state (s^0, C^0), then $\rho \in Runs(\mathcal{D}_I) \setminus Runs(\mathcal{D}, f)$.*

Thus any run ρ leading to the state s_{err} in the synchronous product is a potential counter example. As usual, we can use the finite abstraction to try trace an actual run in original implementation corresponding to ρ. If we succeed in finding such a run, we have found a bug in the original implementation. If the abstract counterexample turns out to be infeasible, then we refine our abstraction using the feedback obtained from our failure to construct a valid run. We repeat this process until a bug is found or we are satisfied with the level of abstraction to which we have verified the system.

5 Conclusion

In this paper, we have shown how to construct a reference implementation for a CRDT that is described using a bounded declarative specification. By imposing reasonable constraints on the universe of the datatype and the underlying message delivery subsystem, the reference implementation can be made finite-state. This can be exploited to verify any given implementation using CEGAR.

The key observation in this paper is that a global reference implementation suffices for verification. This greatly simplifies the construction compared to the distributed reference implementation described in [9], which requires an intricate distributed timestamping procedure due to the local nature of the information available at each replica.

The other interesting feature of our reference implementation is that the basic construction using LARs is independent of the assumptions that we make on the set of data values and the nature of message delivery in order to bound the set of timestamps used. Thus, the reference implementation relies only on the declarative specificaton of the CRDT. We can then separately reason about the size of this implementation under various constraints on the operating environment.

In future work, we would like to explore further benefits of declarative specifications for replicated data types. In particular, one challenging problem is to develop a theory in which we can compose such specifications to derive complex replicated data types by combining simpler ones.

References

[1] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012.

[2] S. Burkhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 271–284, 2014.

[3] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[4] S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[5] Y. Gurevich and L. Harrington. Trees, automata, and games. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 60–65, New York, NY, USA, 1982. ACM.

[6] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 278–324. Springer, 1987.

[7] M. Mukund, K. Narayan Kumar, and M. A. Sohoni. Bounded time-stamping in message-passing systems. *Theor. Comput. Sci.*, 290(1):221–239, 2003.

[8] M. Mukund, G. Shenoy R, and S. P. Suresh. Optimized or-sets without ordering constraints. In M. Chatterjee, J.-N. Cao, K. Kothapalli, and S. Rajsbaum, editors, *ICDCN*, volume 8314 of *Lecture Notes in Computer Science*, pages 227–241. Springer, 2014.

[9] M. Mukund, G. Shenoy R, and S. P. Suresh. Bounded implementations of replicated data types. In *Proceedings of VMCAI 2015*, volume 8931 of *LNCS*, pages 355–372, 2015.

[10] M. Mukund and M. A. Sohoni. Keeping track of the latest gossip in a distributed system. *Distributed Computing*, 10(3):137–148, 1997.

[11] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011. `http://hal.inria.fr/inria-00555588/PDF/techreport.pdf`.

[12] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, pages 386–400, 2011.