Bounded Implementations of Replicated Data Types

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 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.

Burkhardt et al have proposed a declarative framework to specify eventually consistent replicated data types (ECRDTs). Using this, they introduce replication-aware simulations for verifying the correctness of ECRDT implementations. Unfortunately, this approach does not yield an effective strategy for formal verification.

By imposing reasonable restrictions on the underlying network, we recast their declarative framework in terms of standard labelled partial orders. For well-behaved ECRDT specifications, we are able to construct canonical finite-state reference implementations with bounded metadata, which can be used for formal verification of ECRDT implementations via CEGAR. We can also use our reference implementations to design more effective test suites for ECRDT implementations.

1 Introduction

Replicated data types are used by web services that need to 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 [1] 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

* Partially supported by Indo-French CNRS LIA Informel and a grant from Infosys Foundation.
** Supported by TCS Research Fellowship.
*** Partially supported by a grant from Infosys Foundation.

updates across the multiple copies are resolved to guarantee that all the copies eventually agree. *Conflict-free Replicated Data Types (CRDTs)*, introduced in [2, 3], are a sub class 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 [2–5]. Recently, a comprehensive framework has been proposed in [6] 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.

Finite state abstractions have been widely studied in the context of formal verification. Model checking, for instance, uses techniques such as state space enumeration, abstract interpretation and symbolic execution to algorithmically verify if an abstract finite state system satisfies its specification. Finite state models such as automata over distributed words, communicating finite state machines and Petri nets have been successfully used to model and verify concurrent and distributed systems.

In this paper, we focus on a class of CRDTs known as the *Commutative Replicated Data Types (CmRDTs)* whose replicas broadcast every update they receive from a client. The key contributions of our work are as follows

- Demonstrating the use of labelled partial orders as a framework for providing declarative specifications of CmRDTs.
- Generalizing the *gossip problem* introduced in [7] and providing a bounded solution to this problem assuming bounded concurrency.
- Using the bounded solution of the gossip problem to obtain finite state implementations of CmRDTs whose specifications satisfy certain properties.

The paper is organized as follows. In Section 2 we introduce replicated data types. Following this, we show how to use standard labelled partial orders as a framework for declarative specification for CmRDTs. After defining bounded CmRDTs, we generalize the gossip problem and provide a bounded solution in Section 5. We then show how this bounded solution can be used to derive a bounded implementation of CmRDTs. In the next section we show how bounded implementations can be used in the formal verification of CmRDTs. In the final section we summarize our work and discuss interesting challenges.

2 Replicated Data Types

We consider distributed systems consisting of a set $\mathcal{R}$ of N replicas, denoted $[0..N-1]$. 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. Similarly, it updates its local state on receiving an update request from the client.

In this paper, we restrict our attention to a class of replicated data types called *Commutative Replicated Data Types (CmRDTs)*, introduced in [2]. 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 assume that the updates are delivered in causal order—that is, if update u_1 at replica r_1 is initiated before update u_2 at replica r_2, then every replica receives information about u_1 before information about u_2. We shall define this notion formally in the next section. Under this assumption, we note that when a replica receives and applies an update operation, its state would contain the effect of all operations that causally precede the current update. We now define some terminology introduced in [2, 3] to reason about these data types.

A CmRDT $\mathcal{D}$ is a tuple $(\mathcal{V}, \mathcal{Q}, \mathcal{U})$ 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.

The send and receive components of a broadcast that follows an update operation are denoted by **send** and **receive**, respectively. We denote by $\mathcal{M}$ the set $\{\mathbf{send}, \mathbf{receive}\}$.

For an instance of an operation $o \in \mathcal{Q} \cup \mathcal{U} \cup \mathcal{M}$, we use $Rep(o)$, $Op(o)$ and $Args(o)$ to denote the source replica, operation name and arguments, respectively, of the operation.

Definition 1 (Run) *A* run *of a replicated data type is a pair* (α, φ) *where*

- α *is a sequence of operations* $Io_1o_2 \ldots o_n$*, where* I *is a special operation that initializes the states of all the* N *replicas, and each* $o_i \in \mathcal{Q} \cup \mathcal{U} \cup \mathcal{M}$*.*
- φ *is a partial function from* $[0..n]$ *to* $2^{[0..n]}$ *such that*

- $dom(\varphi) = \{i \leq n \mid o_i$ *is a* **send** *operation*$\}$.
- $\forall j \in \varphi(i) : j > i$ *and* o_j *is a* **matching receive** *operation. In particular, the matching receive operation at replica* r *is denoted by* $\varphi_r(i)$.

Note that every update operation in a run $Io_1o_2 \ldots o_n$ will be followed by a send operation where the source replica broadcasts details of this update to all other replicas. Without loss of generality, we assume that this send event is the next event in the run: in other words, if o_i is an update event, then the send event that broadcasts details of o_i to all other replicas is the event o_{i+1}.

For $\alpha = Io_1o_2 \ldots o_n$, we let $\alpha[j]$ denote the operation o_j and $\alpha[:j]$ denote the prefix $Io_1o_2 \ldots o_j$. Note that $\alpha[0]$ (and hence $\alpha[:0]$) is always I. The subsequence of α consisting of all operations with source replica r is denoted α^r. (By convention, every replica r is a source replica for I.)

The state of replica r at the end of the run α is denoted by $S^r(\alpha)$.

Definition 2 (History) *Let* (α, φ) *be a run and* r *be a replica. The* history *of* r *with respect* α*, denoted by* $\mathcal{H}^r(\alpha)$ *is the set of all update operations whose effects are applied at* r*, either directly or indirectly, to arrive at the state* $S^r(\alpha)$*. Formally,* $\mathcal{H}^r(\alpha)$ *is inductively defined as follows:*

- *For* $\alpha = I$, $\forall r \in \mathcal{R} : \mathcal{H}^r(\alpha) = \emptyset$
- *For* $i > 0$, $\mathcal{H}^r(\alpha[:i]) = \begin{cases} \mathcal{H}^r(\alpha[:i-1]) & \text{if } Rep(\alpha[i]) \neq r \text{ or } Op(\alpha[i]) \in \mathcal{Q} \cup \{\textbf{send}\} \\ \mathcal{H}^r(\alpha[:i-1]) \cup \{\alpha[i]\} & \text{if } Rep(\alpha[i]) = r \text{ and } Op(\alpha[i]) \in \mathcal{U} \\ \mathcal{H}^r(\alpha[:i-1]) \cup \mathcal{H}^s(\alpha[:j]) & \text{if } Rep(\alpha[i]) = r, Op(\alpha[i]) = \textbf{receive}, Rep(\alpha[j]) = s, \text{ and } \varphi_r(j) = i \end{cases}$

We define causality and concurrency for pairs of update operations as follows.

Definition 3 (Happened-Before and Concurrency) *Let* $u = \alpha[i]$ *and* $u' = \alpha[j]$ *be update operations at source replicas* r *and* r'*, respectively, in a run* (α, φ)*. We say that* u *has* happened before u'*, denoted* $u \xrightarrow{\text{hb}} u'$*, if* $u \in \mathcal{H}^{r'}(\alpha[:j-1])$.

If neither $u \xrightarrow{\text{hb}} u'$ *nor* $u' \xrightarrow{\text{hb}} u$*, we say that* u *and* u' *are* concurrent. *This is denoted by* $u \parallel u'$.

Strong eventual consistency (SEC) [3] is a stronger variant of *eventual consistency* [8] that is useful for reasoning about the correctness of replicated systems.

Definition 4 (Strong Eventual Consistency) *Let* (α, φ) *be a run and let* r, r' *be a pair of replicas. We say that the replicated data type satisfies strong eventual consistency if* r *and* r' *are query equivalent whenever* $\mathcal{H}^r(\alpha) = \mathcal{H}^{r'}(\alpha)$ *— that is, for any query after* α*,* r *and* r' *return the same values.*

Note that strong eventual consistency does not refer to the order in which updates are applied at a particular replica. As long as the sets of updates applied at two replicas are the same, the observable behaviour of the replicas is identical. *Commutative replicated data types (CmRDTs)* [2,3] are a class of replicated data types that satisfy strong eventual consistency by construction. The definition of CmRDTs ensures that replicas do not need to detect or resolve conflicts. The following characterization of CmRDTs is from [3].

Definition 5 (CmRDT) *A replicated data type is said to be a commutative replicated data type (CmRDT) iff for any pair of update operations u, u' if $u \parallel u'$ then u and u' are commutative. If $u \xrightarrow{\text{hb}} u'$ then at every replica, the effect of u is applied before applying the effect of u'.*

The recent paper [6] provides a comprehensive declarative framework for specification of a large class of replicated data types. The framework is very general and accommodates a wide variety of data stores. For instance, it allows reasoning about data stores with multiple replicated objects, with arbitrary delivery patterns for messages. The variety of features permitted in the declarative framework render it impractical for effective verification of CmRDTs. Retaining the core idea from [6], we provide the specifications of CmRDTs in terms of standard labelled partial orders [9, 10] in the next section.

3 Labelled partial orders models for replicated data types

Let (α, φ) be a run of a replicated data type. We define $\mathcal{E}_\alpha$ to be the set of *events* associated with **send** and **receive** operations in α.

$\mathcal{E}_\alpha = \{e_i \mid 0 \leq i < |\alpha|, \mathit{Op}(\alpha[i]) \in \mathcal{U} \cup \{\textbf{receive}\}\}$. Each $e_i \in \mathcal{E}_\alpha$ corresponds to some operation $\alpha[i]$ in α. We define $\mathit{Rep}(e_i)$, $\mathit{Op}(e_i)$ and $\mathit{Args}(e_i)$ to be $\mathit{Rep}(\alpha[i])$, $\mathit{Rep}(\alpha[i])$ and $\mathit{Args}(\alpha[i])$.

We extend φ to $\mathcal{E}_\alpha$ as follows. For $e_i \in \mathcal{E}_\alpha$, let $\alpha[i]$ be the corresponding event in α. If $\alpha[i] \in \mathcal{U}$, recall that $\alpha[i{+}1]$ is assumed to be the send event where the effect of this update is broadcast to all other replicas. We define $\varphi_\alpha(e_i) = \{e_j \mid j \in \varphi(i{+}1)\}$. Further, we define $\varphi_{r,\alpha}(e_i) = e_j$ if $e_j \in \varphi_\alpha(e_i) \wedge \mathit{Rep}(e_j) = r$.

For a replica r, let $\mathcal{E}^r_\alpha = \{e \in \mathcal{E}_\alpha \mid \mathit{Rep}(e) = r\}$. Since each replica is sequential, all events in $\mathcal{E}^r_\alpha$ are totally ordered. Let $\leq^r_\alpha$ denote this total order on $\mathcal{E}^r_\alpha$. We define $\preceq_\alpha$ to be the smallest partial order on $\mathcal{E}_\alpha$ such that:

- For any replica r, and any pair of events $e, e' \in \mathcal{E}^r_\alpha$, $e \leq^r_\alpha e' \implies e \preceq_\alpha e'$.
- For any event $e \in \mathcal{E}_\alpha$ with $\mathit{Op}(e) = \textbf{receive}$, $\varphi^{-1}_\alpha(e) \preceq_\alpha e$.

We say that a pair of events $e, e' \in \mathcal{E}$ are *concurrent* (denoted by $e \parallel e'$) when neither $e \preceq e'$ nor $e' \preceq e$.

Definition 6 (Trace) *A run (α, φ) gives rise to an associated* labelled partial order *$(\mathcal{E}_\alpha, \varphi_\alpha, \preceq_\alpha)$. We shall use the term* trace *(borrowed from the theory of Mazurkiewicz traces [9]) to refer to this labelled partial order.*

We usually drop the subscript α and assume that the trace that we refer to has an associated run α. Recall that we have assumed that messages are delivered in causal order. This can be formalized in the trace framework as follows:

$$\forall e_i, e_j \in \mathcal{E} : (\ Op(e_i) = Op(e_j) = \textbf{receive} \wedge \\ Rep(e_i) = Rep(e_j)\ \wedge \varphi^{-1}(e_i) \preceq \varphi^{-1}(e_j)\ ) \implies e_i \preceq e_j.$$

Definition 7 (Subtrace) *Let $t = (\mathcal{E}, \varphi, \preceq)$ be a trace. Each subset $\mathcal{E}' \subseteq \mathcal{E}$ defines a* subtrace $t^{\mathcal{E}'} = (\mathcal{E}', \varphi', \preceq')$ *where $\varphi' = \varphi|_{\mathcal{E}'}$ and $\preceq' = \preceq|_{\mathcal{E}'}$.*

Note that a subtrace can have "holes"—we could have three events $e \preceq e' \preceq e''$ in t and a subtrace t' containing $\{e, e''\}$ but not e'.

If X is a predicate that defines the subset $\mathcal{E}' \subseteq \mathcal{E}$ then t^X denotes the subtrace $t^{\mathcal{E}'}$. As special cases, $t^{\mathcal{U}}$ and $t^{\textbf{receive}}$ respectively denote the subtraces consisting of only the update events and **receive** events in t, respectively. We shall denote by $\mathcal{E}^{\mathcal{U}}$ and $\mathcal{E}^{\textbf{receive}}$ their respective event sets. For a pair of traces t and t', the notions $t \subseteq t', t \cup t'$ and $t \cap t'$, are defined in the standard manner. For $t = (\mathcal{E}, \varphi, \preceq)$, we write $e \in t$ to mean that $e \in \mathcal{E}$.

Definition 8 (Downward Closure) *Let $t = (\mathcal{E}, \varphi, \preceq)$ be a trace. A subset $\mathcal{E}' \subseteq \mathcal{E}$ is said to be* downward closed *if $\forall e, e' \in \mathcal{E} : (e \preceq e' \wedge e' \in \mathcal{E}' \implies e \in \mathcal{E}')$*

In particular, for an event e, the downward closure of e is defined to be the set $\downarrow e = \{e' \in \mathcal{E} \mid e' \preceq e\}$.

Clearly, the entire set of events $\mathcal{E}$ is downward closed. Also, if $\mathcal{E}'$ and $\mathcal{E}''$ are downward closed subsets of $\mathcal{E}$, then so are $\mathcal{E}' \cup \mathcal{E}''$ and $\mathcal{E}' \cap \mathcal{E}''$. If $\mathcal{E}'$ is a downward closed set and $\mathcal{E}''$ is the set of maximal events in $\mathcal{E}'$, then $\mathcal{E}' = \bigcup_{e \in \mathcal{E}''} \downarrow e$.

Definition 9 (Ideal) *Let $t = (\mathcal{E}, \varphi, \preceq)$ and $\mathcal{E}' \subseteq \mathcal{E}$. The subtrace $t^{\mathcal{E}'}$ is said to be an* ideal *if $\mathcal{E}'$ is downward closed.*

In particular, if $\mathcal{E}' = \downarrow e$ for some $e \in \mathcal{E}$, then we refer to the ideal $t^{\mathcal{E}'}$ as the prime trace *generated by e and denote it by t_e.*

Definition 10 *Let $t = (\mathcal{E}, \varphi, \preceq)$ be an ideal. Then,*

- *Events(t) denotes $\mathcal{E}$, the set of events in t.*
- *Count$(t) = |Events(t)|$ denotes the number of events in $\mathcal{E}$.*
- *maxSet(t) denotes the set $\{e \in \mathcal{E} \mid \nexists e' \in \mathcal{E} : e \preceq e'\}$ of* maximal events *in t.*
- *For a replica r, the* maximal r event *in t, denoted by $max_r(t)$, is an event e such that $Rep(e) = r$ and $\forall e' \in \mathcal{E} : Rep(e') = r \implies e' \preceq e$. (Note that $max_r(t)$ is always defined since the event corresponding to the initialization operation I is an r event for every replica r.)*
- *The* r-view *of t is the ideal generated by $max_r(t)$ and is denoted by $\partial_r(t)$.*
- *The* latest r' event *that r is aware of in the ideal t, denoted by $latest_{r \to r'}(t)$ is defined to be $max_{r'}(\partial_r(t))$.*

The behaviour of a replicated data type is the set of traces that it generates. Note that this set is downward closed—if a trace t is present in the set then all ideals in t are also present in the set. We shall denote this set by $\mathcal{T}$. We let $\mathcal{T}_p$ denote the set of all prime traces in $\mathcal{T}$. In the trace framework, the definitions of happened-before and concurrency are straightforward.

Definition 11 (Happened Before and Concurrency) *Let $t = (\mathcal{E}, \varphi, \preceq)$ be a trace with events e and e' that are associated with update operations u and u', respectively. Then u is said to have* happened before *u', denoted $u \xrightarrow{\text{hb}} u'$, if $e \preceq e'$, and u and u' are said to be* concurrent *if neither $e \preceq e'$ nor $e' \preceq e$.*

We can now reformulate *strong eventual consistency (SEC)* as follows.

Definition 12 (Strong Eventual Consistency(SEC)) *Let t be a trace and let r, r' be a pair of replicas. We say that the replicated data type satisfies strong eventual consistency if the replicas r and r' are query equivalent whenever $Events(\partial_r(t)^{\mathcal{U}}) = Events(\partial_{r'}(t)^{\mathcal{U}})$.*

Note that strong eventual consistency refers only to the subtraces $\partial_r(t)^{\mathcal{U}}$ and $\partial_{r'}(t)^{\mathcal{U}}$ defined by the updates events in the r and r' views of t. These views may have different sets of **receive** events.

We now show how we can specify CmRDTs in the framework of traces.

3.1 Specifications of CmRDTs

The internal state of a replicated data type is exposed to the client via queries. Hence, a specification framework for a CmRDT should provide mechanisms for uniformly defining the behaviour of any query operation that is applied at replica at any stage of the computation, based on the update operations in the history of the replica at that stage. Note that in the framework of traces, the history of a replica corresponds to the update events in view of the replica, which is a prime trace. Hence, we define the specification of a CmRDT with respect to prime traces.

Definition 13 (Declarative Specification) *Let $\mathcal{D} = (\mathcal{V}, \mathcal{Q}, \mathcal{U})$ be a CmRDT. Let $\mathcal{A} = \bigcup_{q \in \mathcal{Q}} Args(q)$. The specification of $\mathcal{D}$, is a function $Spec_{\mathcal{D}} : \mathcal{Q} \times \mathcal{A} \times \mathcal{T}_p \to \mathcal{V}$ which determines the return value of any query $q \in \mathcal{Q}$ in any prime trace $t \in \mathcal{T}_p$.*

We highlight the key differences between our declarative specification and the one proposed in [6]. Since messages in CmRDTs are causally delivered, the $\preceq$ relation in the trace framework captures both the visibility relation *vis* and the replica order relation *ro* from [6]. However, while $\preceq$ is a partial order, *vis* is only defined to be an acyclic relation over the events, while *ro* is the union of total orders that is recovered by restricting $\preceq$ to events of the same replica. In [6], the *execution context* of an event is defined to be the set of all events that are visible to the event. The specification defines the return value of each query

in every execution context. In the trace framework, the execution context for an event is the ideal generated by that event.

We now provide declarative specifications for a few CmRDTs in the framework of traces.

PN counters A PN counter maintains a counter by keeping track of the number of increments and decrements it receives. A query should return the latest count that a replica is aware of.

- $\mathcal{U} = \{Inc, Dec\}$
- $\mathcal{Q} = \{Fetch\}$ and $arity(Fetch) = 0$
- **Specification:** Let t be a trace. Let $I = \{e \in t \mid Op(e) = Inc\}$ and $D = \{e \in t \mid Op(e) = Dec\}$. In any prime trace t,

$$Spec_{Counter}(Fetch, \bot, t) = Count(t^I) - Count(t^D)$$

MV registers An MV register is a read-write register that on a read, returns the values of the latest (possibly concurrent) writes.

- $\mathcal{U} = \{Write\}$
- $\mathcal{Q} = \{Read\}$ and $arity(Read) = 0$
- **Specification:** Let $W = \{e \in t \mid Op(e) = Write\}$. In any prime trace t,

$$Spec_{MVReg}(Read, \bot, t) = \bigcup Args(maxSet(t^W))$$

OR sets An OR set is a distributed set that follows the "add-wins" semantics for concurrent adds and deletes of the same element.

- $\mathcal{U} = \{Add, Delete\}$
- $\mathcal{Q} = \{Contains\}$ and $arity(Contains) = 1$
- **Specification:** For an element $x \in \mathcal{V}$, define a predicate $U_x = \{e \in t \mid x \in Args(e)\}$. In any prime trace t,

$$Spec_{ORSet}(Contains, (x), t) = True \iff \exists e \in maxSet(t^{U_x}) : Op(e) = Add$$

In the next section, we discuss how declarative specifications in terms of labelled partial orders can be used to obtain bounded implementations for a class of *well-behaved* CmRDTs.

4 Bounded CmRDTs

In this section we discuss sufficient conditions for replicated data types to have a bounded implementation.

Definition 14 *We say that an implementation of a CmRDT is bounded if the information maintained by every replica and the contents of each message propagating an update are bounded, regardless of the length of the computation.*

Finite state implementations have played an important role in formal verification of reactive systems. We shall see later that they can be used for verification of replicated data types as well. We use the example of OR-Sets to discuss the challenges involved in arriving at a bounded implementation.

Observe that if the universe $\mathcal{V}$ of an OR-Set is unbounded, we cannot hope to achieve a bounded implementation since we need labels of unbounded size to name the elements in $\mathcal{V}$, even if the size of the actual set is bounded. Hence, we assume that $\mathcal{V}$ is bounded, in order to achieve a bounded implementation.

If the size of $\mathcal{V}$ is bounded by K, then the number of unique queries is also bounded by K. In the OR-Set specification, for an element $x \in \mathcal{V}$, the query **contains**(x) requires only the maximal x-events present in the view of the replica. The number of such maximal x-events is bounded by N. This implies that the number of events required to answer a query at any point in time is bounded by KN. Thus, in a reference implementation, it suffices to keep track of only this finite fragment of the partial order at any replica. Replicas can purge from their view events that are no longer relevant for answering any queries.

However, this requires the causal order between the relevant events to be correctly maintained by the replicas. Typically, vector clocks have been used to track causality among the events of a distributed system. However, vector clocks grow monotonically as the computation progresses. Hence, any implementation that uses vector clocks cannot be bounded.

Earlier work such as [7, 11] has implemented bounded timestamping in distributed systems by solving what is known as the *gossip problem*. In the next section, we present a generalized version of the gossip problem and provide a bounded solution to it. We then use the bounded solution to arrive at a bounded implementation for a class of well behaved replicated data types.

5 Generalized Gossip Problem

Consider a distributed system with N replicas. Whenever a replica interacts with a client, it does some local processing and broadcasts a message to all the other replicas. This is similar to the behaviour of CmRDTs described earlier. Suppose now that every replica keeps track of the latest event it knows about every other replica in the system. During a broadcast, along with the message, each replica r also sends across its knowledge about the latest event of every other replica r' in the system. A recipient r'' needs to correctly compute for every replica r' whether its knowledge of the latest r' event is more up to date than the knowledge of r' that it has received from r. This is known as the *gossip problem*, and has been studied in [7, 11].

In the generalized gossip problem, instead of maintaining just the latest events, we assume that every replica keeps track of a bounded subtrace of its view which we refer to as the primary information.

Definition 15 (Information graphs) *An information graph G of a trace $t = (\mathcal{E}, \preceq, \varphi)$ is a subtrace t^E where $E \subseteq_{\text{fin}} \mathcal{E}^{\mathcal{U}}$. We denote the set of all information graphs of t by $\mathcal{G}(t)$. Let $\mathcal{G} = \bigcup_{t \in \mathcal{T}} \mathcal{G}(t)$.*

Definition 16 (Primary information) *A* primary information function *is a function $f : \mathcal{T}_p \to \mathcal{G}$ that assigns an information graph to each prime trace so that the following conditions are satisfied.*

- $f(t) \in \mathcal{G}(t)$.
- *For any trace $t = \downarrow e$ and $e' \in maxSet(t \setminus \{e\})$, $f(t) \cap \downarrow e' \subseteq f(\downarrow e')$.*

A primary information function f is said to be bounded *if $\exists M\ \forall t : |f(t)| \leq M$.*

Our goal is to ensure that $f(\downarrow e)$ can be computed from $f(\downarrow e_1), \ldots, f(\downarrow e_n)$ and e, where $\{e_1, \ldots, e_n\} = maxSet(\downarrow e \setminus \{e\})$. We refer to this as the **(generalized) gossip problem** for a primary information function f. Observe that even if $f(t)$ is a bounded set for every t, we still need an unbounded set of labels to unambiguously identify the events.

We say that the gossip problem for f has a bounded solution if we can update f using a bounded set of labels to identify the events. Clearly we must ensure that no two distinct events in the primary information have the same labels. Hence, each replica needs to identify which of its events are in the primary information of other replicas. To capture this, we define secondary information.

Definition 17 (Secondary information) *A function $F : \mathcal{R} \times \mathcal{T}_p \to 2^{(\mathcal{E}^0)}$ ($\mathcal{E}^0 = \bigcup_{t \in \mathcal{T}_p} Events(t)$) is a* secondary information function *for a primary information function f if, for each prime trace $t = \downarrow e$:*

1. *$Events(f(\partial_r(t))) \cap \mathcal{E}^{\mathcal{U}} \subseteq F(r,t) \subseteq Events(t) \cap \mathcal{E}^r$.*
2. *$F(r,t) = F(r, \downarrow max_r(t))$ if e is not an r-event.*
3. *$F(r,t)$ is computable from e, $\bigcup_{e' \in maxSet(\downarrow e \setminus \{e\})} F(Rep(e'), \downarrow e')$ and $f(t)$, if e is an r-event.*
4. *If e and e' are r-events such that $e' \in \downarrow e \setminus F(r, \downarrow e)$, then for any r' and t', $e \in \partial_{r'}(t') \implies e' \notin f(\partial_{r'}(t'))$.*
5. *For an r-event $e' \in t \setminus F(r,t)$, if $e' \in f(\downarrow e'')$, then $\varphi_{r''}(e'') \in \partial_r(t)$ for all r''.*

A secondary information function F is said to be bounded *if $\exists M\ \forall r\ \forall t : |F(r,t)| \leq M$.*

The first three conditions are straightforward: they say that F is locally updatable and that it subsumes f. The fourth condition states, in essence, that r can reuse the label of an event e' that has left $F(r,t)$ for a new event e, since at the point when another replica r' receives e, e' would have moved out of its primary information, so there is no ambiguity in the labelling. The last condition is subtle and crucial for the next proof. It states that if r has generated a label for an event e' and potentially reused it for a later event after e' leaves $F(r,t)$, then there is no update event e'' by another replica r' that could potentially send this reused label later to an agent. In other words, such sends have been delivered to all replicas before r reuses the label.

Theorem 18 (Bounded solution for gossip). *Let f be a primary information function such that the gossip problem for f has a solution. It has a bounded solution if there is a bounded secondary information function F for f.*

Proof. Let the gossip problem for f have a solution, and let the bound on F be M. Clearly $M+1$ labels suffice to label each event in $F(r,t)$ uniquely, for any t. We fix a label set $\mathcal{L}$ of size $M+1$. For any t, we label events in $F(r,t)$ with pairs (r,ℓ), such that $\ell \in \mathcal{L}$. We also maintain a marking function that identifies events in $f(t)$, and the ordering $\prec$ restricted to $f(t)$. As a trace progresses, we can reassign an unused label (there is at least one) to each new event in the set F. Since both f and F after each event e are computable from the values of f and F at the maximal r events in t, the marking and ordering can also be updated.

Finally, replica r can reuse labels for events in $F(r,t)$ without fear of confusion because of condition 4 in the definition of secondary information. If e' and e are two r update events with $e' \prec e$ that are assigned the same label (r,ℓ), then clearly $e' \in t \setminus F(r,t)$. When r' receives the event e (let us say this receive event is e_1 and $t' = {\downarrow}e_1$), $e' \notin f(\partial_{r'}(t')) = f(t')$. Thus r' can decide if the received event is new or old by referring to the f at the receive event.

Let e' and e be r-update events such that $t = {\downarrow}e$ and $e' \in t \setminus F(r,t)$. Suppose r uses the same label (r,ℓ) for e and e'. A replica r'' can receive an event with label (r,ℓ) from a replica $r' \neq r$. This means that there is an r'-event e'' such that $e' \in f({\downarrow}e'')$. (Note that replicas communicate their primary information to others.) But condition 5 ensures that $\varphi_{r''}(e'') \in \partial_r(t)$, so this send happens before the label (r,ℓ) is reused.

Thus there is no ambiguity caused either by sends by the same replica or by sends by different replicas. This shows that a bounded secondary information function implies a bounded labelling solution. □

5.1 Bounded Solution

In the gossip problem considered in [7,11], the primary information f at t is the set $\{max_{r'}(\partial_r(t)) \mid r, r' \in \mathcal{R}\}$ along with some more information that ensures that the gossip problem for f is solvable. A bounded solution for f is provided in [11] when f itself is bounded, but under additional restrictions on the traces of the system. A notion of acknowledgements is introduced, and all traces of the system are required to have at most B unacknowledged messages.

In [11], replicas piggyback acknowledgements to previously received messages in their subsequent broadcasts. Hence, the bound on unacknowledged messages requires that replicas communicate with each other at regular intervals. Such a solution would not work in CmRDTs, because replicas broadcast messages only when they interact with a client. Hence, we need a stronger guarantee from the underlying messaging system. One such condition is defined below.

Definition 19 (B-concurrency) *A trace t is B-concurrent if for every $e \in t$, $|\{e' \in t \mid e' \parallel e\}| \leq B$. A system is B-concurrent if all its traces are B-concurrent.*

The following theorem is the main result in this section. It implies (in conjunction with Theorem 18) that for a B-concurrent system and a bounded primary

information function f, if the gossip problem for f has a solution, then it has a bounded solution.

Theorem 20. *If f is a bounded primary function defined on a B-concurrent system, there is a bounded secondary information function for f.*

Proof. Let the bound on f be M. To define the function F, we need the notion of *recent updates*. An r-update event e is recent (in a trace t) if $e \in f(\partial_r(t))$ or there are at most B r-updates after e in $\partial_r(t)$. We define the function F as follows: $F(r,t) = \{e \in \partial_r(t) \mid e \text{ is a recent } r\text{-update event}\}$.

Clearly $|F(r,t)| \leq M + B + 1$ for any r and t, and conditions 1, 2 and 3 in the definition of secondary information function easily hold.

Condition 4 is relatively simple. $e' \in (\downarrow e \setminus F(r, \partial_r(\downarrow e)))$ implies that $e' \notin f(\partial_r(\downarrow e))$. Hence for any t' such that $e \in \partial_{r'}(t')$, by definition of a primary information function, it is the case that $e' \notin f(\partial_{r'}(t'))$.

Condition 5 is proved as follows. If $e' \in t \backslash F(r,t)$, then there are $B+1$ r-update events after e'. Suppose there is e'' with $e' \in f(\downarrow e'')$ such that $\varphi_{r''}(e'') \notin \partial_r(t)$, for some replica r''. Then there are more than B events concurrent with e'', which contradicts B-concurrency.

Thus whenever f is a primary information function with a bound M in a B-concurrent system, there is a secondary information function F for f with bound $M + B + 1$. □

5.2 Bounding CmRDTs using Generalized Gossip Problem

Definition 21 *Let $\mathcal{D} = (\mathcal{V}, \mathcal{Q}, \mathcal{U})$ be a CmRDT. We say that a function $f_{\mathcal{D}} : \mathcal{T}_p \to \mathcal{T}$ is a* specification-subtrace function *iff:*

$$\forall q \in \mathcal{Q}\ \forall t \in \mathcal{T}_p\ \forall args \in \mathcal{V}^{arity(q)} :\ Spec_{\mathcal{D}}(q, args, t) = Spec_{\mathcal{D}}(q, args, f_{\mathcal{D}}(t))$$

Thus, a specification-subtrace function picks for every prime trace a subtrace that is sufficient to answer every query of the CmRDT, as per its specification. The identity map is a trivial specification-subtrace function for any CmRDT.

We now provide a sufficient condition for a CmRDT to have a bounded implementation.

Theorem 22. *A CmRDT $\mathcal{D} = (\mathcal{V}, \mathcal{Q}, \mathcal{U})$ has a bounded implementation in a distributed system whose underlying network guarantees B-concurrent traces if there exists a locally computable specification-subtrace function f_D that is a bounded primary information function*

Proof. From Theorem 20, we know that the generalized gossip problem has a bounded solution in a distributed system whose traces are B-concurrent. The bounded solution for the gossip problem with primary information function $f_{\mathcal{D}}$ maintains at every replica r in any trace t, $f_{\mathcal{D}}(\partial_r(t))$. Since $f_{\mathcal{D}}$ is also a specification-subtrace function for $\mathcal{D}$, by definition, the information maintained is sufficient to answer every query correctly as per the specification of $\mathcal{D}$.

Thus, whenever a replica gets an update request $u(args)$ from the client, it is sufficient if it annotates the new event e with $u(args)$ and invokes the bounded solution to the generalized gossip problem. Also, it is sufficient to implement each query operation $q \in \mathcal{Q}$ as per the specification of $Spec_{\mathcal{D}}(q, args, f_{\mathcal{D}}(\partial_r(t)))$. This provides a bounded implementation for $\mathcal{D}$. □

Let us revisit the case of OR-Sets for which $|\mathcal{V}| \leq K$.

Definition 23 *Let $t \in \mathcal{T}_p$ be any prime-ideal of an OR-Set. Let $\mathcal{E}(x,t) = \{e \in t \mid Args(e) = x \wedge Op(e) \in \mathcal{U}\}$. Let*

$$\mathcal{E}(t) = \{\bigcup_{r \in \mathcal{R}} max_r(t^{\mathcal{U}})\} \cup \bigcup_{x \in \mathcal{V}} maxSet(\partial_r(t)^{\mathcal{E}(x,t)}).$$

We define $f_{ORSet}(t) = t^{\mathcal{E}(t)}$.

From the definition of f_{ORSet} and the specification of OR-Sets presented earlier in Section 3, the following result is evident:

Lemma 24. *f_{ORSet} is a specification-subtrace function for OR-Sets.*

We show that f_{ORSet} is a bounded primary information function.

Lemma 25. *f_{ORSet} is a computable bounded primary information function.*

Proof. In the following proof, we drop the subscript *ORSet* from f_{ORSet} and use f, for ease of presentation.

Let $t \in \mathcal{T}_p$ be any prime ideal.

From the definition, $f(t)$ is a subtrace of t consisting of only the maximal $\mathcal{U}$ events corresponding to each element $x \in \mathcal{V}$ along with the maximal r-update events for each replica r. Thus, $f(t) \in \mathcal{G}(t)$.

Let $t = \downarrow e$ and $e' \in maxSet(t \setminus \{e\})$. Let $t' = \downarrow e'$. Now, an event $e'' \in f(t)$ iff $\exists r \in \mathcal{R}$ such that e'' is an r-maximal update event in t or $\exists x \in \mathcal{V}$ such that e'' is a maximal x-event in t. For an $e'' \in t'$, since $t' \subseteq t$, it is clear that if e'' is a maximal x-event in t then, e'' is also a maximal x-event in t'. Similarly if $e'' \in t'$ and e'' is an r-maximal update event in t then e'' is an r-maximal update event in t'. Thus, $e'' \in f(t')$. Hence, $f(t) \cap t' \subseteq f(t')$.

Thus we have shown that f is a primary information function.

For each replica r, there is one maximal r-update event in any trace t. For each element x, there can be at most N maximal x-events in any prime ideal t. Since $|\mathcal{V}| \leq K$, there can be at most KN events corresponding to the maximal update events. Thus, in any prime ideal t, $|f(t)| \leq KN$, so f is bounded.

Finally to show that f admits a solution to the generalized gossip problem, we establish that $f(t)$ can be locally computed from e, and $\bigcup_{e' \in maxSet(t \setminus \{e\})} f(\downarrow e')$.

We introduce the following notation which will be used below: For any prime trace t'', we write $f(t'') = (E'', \rightarrow'')$ to denote that the set of events in $f(t'')$ is E'' and the trace order restricted to E'' in $f(t'')$ is $\rightarrow''$.

Now, let $Rep(e) = r$. We consider the following cases:

Case e is an x-update event: Let $\{e'\} = maxSet(t \setminus \{e\})$. Let $t' = \downarrow e'$, with $f(t') = (E', \rightarrow')$. Let $E_m = \{e\} \cup E'$ and $\rightarrow_m = \rightarrow' \cup \{(e'', e) \mid e'' \in E'\}$, with $t_m = (E_m, \rightarrow_m)$. It is easy to see that if $e'' \in f(t)$ then $e'' \in E_m$. Moreover, since $f(t') \in \mathcal{G}(t')$ and $t' \subseteq t$, the ordering on events of $f(t')$ is given by $\rightarrow'$ which is consistent with their ordering in t. Finally e is a maximal event in t and hence is above other event in E_m. Both these are captured in $\rightarrow_m$. Hence we can compute $f(t)$ by picking the subtrace of t_m containing the maximal r' events for every $r' \in \mathcal{R}$ and maximal y events for every $y \in \mathcal{V}$. Thus $f(t)$ can be computed from $f(t')$ and e.

Case e is a receive event: Let $\{e', e''\} = maxSet(t \setminus \{e\})$. Let $Rep(e') = r$ and $Rep(e'') = r'$ Let $t = \downarrow e$, $t' = \downarrow e'$ and $t'' = \downarrow e''$. We let $f(t') = (E', \rightarrow')$ and $f(t'') = (E'', \rightarrow'')$.

By causal delivery, every event $e''' \in t'' \setminus \{e''\}$ is already in t'. So if such an $e''' \notin f(t')$ then $e''' \notin f(t)$. Also, any event $e''' \in (f(t') \cap t'') \setminus f(t'')$ is not going to feature in $f(t)$, from the definition of primary information. Such an event can be identified correctly in $f(t')$ since it will be the case that $(e''', max_{r'}(t')) \in \rightarrow'$. Thus, the events required to compute $f(t)$ are $E_m = \{e''\} \cup E' \setminus \{e''' \mid e''' = max_{r'}(t') \vee e''' \rightarrow' max_{r'}(t')\}$. Let $\rightarrow_m = (E_m \times E_m) \cap (\rightarrow' \cup \rightarrow'')$. Then, $f(t)$ can be computed from $t_m = (E_m, \rightarrow_m)$ by picking the subtrace from t_m consisting of the maximal r'' events from every replica r'' and maximal x-events for every element $x \in \mathcal{V}$. Thus $f(t)$ can be computed from $f(t')$, $f(t'')$ and e. □

Hence, from Lemmas 24 and 25 we can conclude that an OR Set with a bounded universe has a bounded implementation in a distributed system whose underlying network guarantees B-concurrent traces.

We present the bounded implementation for OR-Sets via a bounded solution for the generalized gossip problem in Algorithm 1 (page 18). The algorithm stores the relevant information view of a replica in (V, P). It uses a free set of labels F which can be used to label a new update event at the replica and a retired set of labels R (line 4) which keeps track of the labels that are no longer relevant but might still be active in the system.

Lines 9–19 describe the GENERATENODE() method that generates a new node corresponding to the latest update event (Line 38). This method picks an unused label from F (line 12) and increments the retired-duration of the labels in R (line 13). The labels which have been in the retired set for a duration of B-sends can be recycled (Lines 15, 16). The node consists of the replica id, the label and the private information that the client would have passed on to the replica (Line 18). We shall later see that this private information will be the name of the update function as well as the argument to it.

Helpers GETREP() and GETLABEL() return the replica id and the label associated with a node, respectively (Lines 21–29). Helper GETMAXIMAL() returns the maximal events in the partial order (V', P') (Lines 31–35).

The SEND() method (Lines 37–41) creates a new node and computes the new partial order with this new node as the maximal node (Lines 38–40). It locally recomputes the value of the primary information (Line 41) and then broadcasts

the updated partial order to all the other replicas (Line 42). The RECEIVE() method recomputes the relevant information based on the new information that the replica has received.

RECOMPUTERELEVANT() method merges the relevant information at the replica along with the updated partial order that it has received and recomputes the relevant information from this new subtrace (Line 49). It then records the nodes that were present in the relevant information prior to recomputing but are no longer present (Line 53 and 54). It retires the labels corresponding to these nodes (Line 55). Finally it updates its primary information with the one that was computed in Line 50 (Line 56).

The second part of the algorithm describes how OR Sets can be implemented using the bounded solution to the gossip problem. Whenever a replica receives an update request from the client, it invokes the SEND() method with the update name and the argument as the private information (Lines 18–22). Lines 9–20 implement the specification-subtrace function for OR-Set that retains only the maximal events corresponding to each element of the universe along with the maximal events corresponding to each replica in the system. Lines 28–34 implement the EXISTS query method that returns True if there is at least one maximal node corresponding to the x that was added due to an ADD update operation.

6 Applications to verification

A bounded reference implementation can be used for verification. We outline two scenarios in which such an implementation is useful.

CEGAR Counterexample Guided Abstraction Refinement, or CEGAR, is an iterative technique to verify reachability properties of software systems [12]. In the CEGAR approach, one uses abstraction techniques from program analysis and other domains to build a finite-state abstraction of a given implementation. This abstraction is designed to over-approximate the behaviour of the original system.

The finite-state approximation is run through a model-checker to verify if the safety property is met. If no unsafe state is reachable, it means that the original system is safe since the abstracted system over-approximates the actual behaviour. On the other hand, if the model-checker asserts that an unsafe state is reachable, the counterexample generated by the model-checker is executed on the original system. If the counterexample is valid, a bug has been found. If the counterexample is infeasible, the abstraction was too coarse and a refinement of the abstraction is calculated. This process is iterated until a safe abstraction is reached or a valid bug is detected.

As we have noted, implementations of replicated data types need to keep track of metadata about past operations in order to reconcile conflicts. These are typically done using unbounded objects such as counters or vector clocks. To apply CEGAR to such an implementation, we can derive a finite state abstraction

and run it synchronously with our bounded reference implementation. We can characterize each reachable state of the abstraction as legal or illegal depending on whether or not it is query equivalent to the reference implementation. We can then follow the usual CEGAR methodology outlined above.

Testing of distributed systems While verification approaches such as CEGAR can be used in a white box setting where we have access to internal details of the implemenation under test, in a black box scenario we have to rely on testing.

Effective testing of distributed systems is a challenging task. The first problem is that we cannot typically test the system globally, so we have to apply tests locally using notations such as TTCN [13]. Even when such a methodology is available, there are two criteria that are difficult to establish for test suites: coverage and redundancy. In both cases, the main source of complexity is the presence of concurrency. In a concurrent system, it is very difficult to estimate if a test suite covers a reasonable set of reachable global states because of many different linearizations possible. Secondly, it is not obvious to what extent tests are overlapping, again because of reordering of independent events.

For the coverage problem, we can construct test suites that cover different portions of the state space of the reference implementation. If this coverage is widespread, we can have more confidence in the coverage of the implementation under test. For redundancy, once again we can use the underlying independence relation to identify when two tests overlap by checking how much the corresponding traces overlap as partial orders. In the replicated data type scenario, we may in fact want to generate redundant test cases that differ only in the order of concurrent events in order to validate eventual consistency.

7 Conclusion and Summary

The theory of replicated data types is still at a formative stage. In early work [2,3] eventually consistent replicated data types have been specified operationally, in terms of a proposed implementation. This often leaves the actual behaviour of the data type unclear under different combinations of concurrent updates.

This deficiency has been addressed in [6], which introduces a theory of declarative specifications for replicated data types. However, the model proposed in [6] is very general and hence ineffective for actual verification.

Most practical distributed systems provide strong guarantees on the underlying message subsystem, such as causal delivery. In fact, causal delivery is assumed to hold for the implementations described in [2, 3]. If we assume causal delivery, we have shown that we can drastically simplify the declarative framework proposed in [6] and work with standard labelled partial orders.

Borrowing ideas from Mazurkiewicz trace theory, we have formulated a generalization of the gossip problem and shown that this can be used to derive bounded implementations for replicated data types, provided we have an additional guarantee of bounded concurrency. Though bounded concurrency seems

like a very strong property, it is automatically achieved if we combine causal message delivery with bounded message delays. The only complication that can arise is from a replica crashing. However, if we assume that when a replica wakes up from a crash, it first processes all pending receive actions before initiating any sends, we retain bounded concurrency. Note that causal delivery is also infeasible if we do not make similar assumptions about how a crashed process recovers.

Our main contribution is a systematic approach to construct bounded reference implementations for replicated data types. We have argued that this kind of implementation is useful for both verification and testing.

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. Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2) (2002) 51–59
2. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA (January 2011) `http://hal.inria.fr/inria-00555588/PDF/techreport.pdf`.
3. Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: SSS. (2011) 386–400
4. Bieniusa, A., Zawirski, M., Preguiça, N.M., Shapiro, M., Baquero, C., Balegas, V., Duarte, S.: An optimized conflict-free replicated set. CoRR **abs/1210.3368** (2012)
5. Mukund, M., Shenoy, G.R., Suresh, S.P.: Optimized or-sets without ordering constraints. In Chatterjee, M., Cao, J.N., Kothapalli, K., Rajsbaum, S., eds.: ICDCN. Volume 8314 of Lecture Notes in Computer Science., Springer (2014) 227–241
6. Burkhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. (2014) 271–284
7. Mukund, M., Sohoni, M.A.: Keeping track of the latest gossip in a distributed system. Distributed Computing **10**(3) (1997) 137–148
8. Vogels, W.: Eventually consistent. ACM Queue **6**(6) (2008) 14–19
9. Mazurkiewicz, A.: Trace theory. In: Petri Nets: Applications and Relationships to Other Models of Concurrency. Springer (1987) 278–324
10. Pratt, V.: Modeling concurrency with partial orders. International Journal of Parallel Programming **15**(1) (1986) 33–71
11. Mukund, M., Narayan Kumar, K., Sohoni, M.A.: Bounded time-stamping in message-passing systems. Theor. Comput. Sci. **290**(1) (2003) 221–239
12. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5) (2003) 752–794
13. Willcock, C., Deiß, T., Tobies, S., Keil, S., Engler, F., Schulz, S.: An Introduction to TTCN-3. Wiley (2005)

Algorithm 1 Bounded Reference Implementation

Bounded Reference Implementation

```
1  V: Set of nodes
2  P: Partial order on V
3  F: Set of available labels.
4  R: Set of pair of retired labels and
   a modulo B+1 counter.
5  // E captures the relation ≺
6  // Each replica stores a copy of
   V, E, F and R
7        initially ∅, ∅, L, ∅
8
9  helper generateNode(priv):
10 Returns v ∈ N
11       Let l ∈ F
12       Let F := F \ {l}
13       Let R' := {(l', c' + 1) | (l', c') ∈ R}
14       Let F' := {(l', c') ∈ R' | c' = B}
15       R := R' \ F'
16       F := F ∪ {l' | (l', c') ∈ F'}
17       Let rep := myID()
18       Let v' := (l, rep, priv)
19       return v'
20
21 helper getRep(v):
22 Returns the replica associated with a Node.
23       Let v = (l, r, p)
24       return r
25
26 helper getLabel(v):
27 Returns the replica associated with a Node.
28       Let v = (l, r, p)
29       return l
30
31 helper getMaximal(V', P'):
32 Returns the set of maximal nodes in the
   partial order
33       Let V^max := {v ∈ V' |
34       ¬∃v' ∈ V' : (v, v') ∈ P'}
35       return V^max
36
37 generic send(priv):
38       Let v' := generateNode(priv)
39       Let V' := V ∪ {v'}
40       Let P' := P ∪ {(v, v') | v ∈ V}
41       recomputeRelevant(V', P')
42       Broadcast (V', P') to all other replicas
43
44 generic receive(V', P'):
45       recomputeRelevant(V', P')
46 helper recomputeRelevant(V', P'):
47 // Recomputes the relevant information
48 // in the partial order (V ∪ V', E ∪ P')
49 // and retires the irrelevant labels.
50       Let (V'', P'') := f(V ∪ V', P ∪ P')
51       Let {v'} := getMaximal(V', P')
52       Let rep := myId()
53       Let L := {getLabel(v) |
54       v ∈ V ∪ {v'} \ V'', getRep(v) = rep}.
55       Let R := R ∪ {(l, 0) | l ∈ L}
56       (V, P) := (V'', P'')
```

OR Set

```
1  helper getOp(v):
2       Let v = (l, r, (opname, x))
3       return opname
4
5  helper getArgs(v):
6       Let v = (l, r, (opname, x))
7       return x
8
9  helper f_ORSet(V', P'):
10   For x ∈ V:
11       Let V_x := {v ∈ V' | getArgs(v) = x}
12       Let P_x := P'' ∩ (V_x × V_x)
13       Let V_x^max := getMaximal(V_x, P_x)
14   For r ∈ R:
15       Let V_r := {v ∈ V' | getRep(v) = r}
16       Let P_r := P'' ∩ (V_r × V_r)
17       Let V_r^max := getMaximal(V_r, P_r)
18 Let V'' := ⋃_{x∈V} V_x^max ∪ ⋃_{r∈R} V_r^max
19 Let P'' := P' ∩ (V'' × V'')
20 return (V'', P'')
21
22 update add(x):
23       send((add, x))
24
25 update delete(x):
26       send((delete, x))
27
28 query exists(x):
29       Let V_x^add := {v ∈ V |
           getArgs(v) = x ∧ getOpv = add}
30       If V_x^add ≠ ∅:
31       Then
32            return True
33       Else:
34            return False
```