Certified Mergeable Replicated Data Types

Vimala Soundarapandian
IIT Madras
Chennai, India
cs19d013@cse.iitm.ac.in

Adharsh Kamath
NITK Surathkal
Surathkal, India
adharshkamathr@gmail.com

Kartik Nagar
IIT Madras
Chennai, India
nagark@cse.iitm.ac.in

KC Sivaramakrishnan
IIT Madras and Tarides
Chennai, India
kcsrk@cse.iitm.ac.in

Abstract

Replicated data types (RDTs) are data structures that permit concurrent modification of multiple, potentially geo-distributed, replicas without coordination between them. RDTs are designed in such a way that conflicting operations are eventually *deterministically* reconciled ensuring *convergence*. Constructing correct RDTs remains a difficult endeavour due to the complexity of reasoning about independently evolving states of the replicas. With the focus on the correctness of RDTs (and rightly so), existing approaches to RDTs are less efficient compared to their sequential counterparts in terms of the time and space complexity of local operations. This is unfortunate since RDTs are often used in a local-first setting where the local operations far outweigh remote communication.

This paper presents Peepul, a pragmatic approach to building and verifying efficient RDTs. To make reasoning about correctness easier, we cast RDTs in the mould of the distributed version control system, and equip it with a three-way merge function for reconciling conflicting versions. Further, we go beyond just verifying convergence, and provide a methodology to verify arbitrarily complex specifications. We develop a replication-aware simulation relation to relate RDT specifications to their efficient purely functional implementations. We implement Peepul as an F* library that discharges proof obligations to an SMT solver. The verified efficient RDTs are extracted as OCaml code and used in Irmin, a distributed database built on the principles of Git.

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**; • **Software and its engineering** → **Formal software verification**; • **Computer systems organization** → **Availability**.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523735

Keywords: MRDTs, Eventual consistency, Automated verification, Replication-aware simulation

ACM Reference Format:
Vimala Soundarapandian, Adharsh Kamath, Kartik Nagar, and KC Sivaramakrishnan. 2022. Certified Mergeable Replicated Data Types. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523735

1 Introduction

Modern cloud-based software services often replicate data across multiple geographically distributed locations to tolerate partial failures of servers and to minimise latency by bringing data closer to the user. While services like Google Docs allow several users to concurrently edit the document, the conflicts are resolved with the help of a centralised server. On the other hand, services like Github and Gitlab, built on the decentralised version control system Git, avoid the need for a centralised server and permit the different replicas (forks) to synchronize with each other in a peer-to-peer fashion. By avoiding centralised server, *local-first software* [20] such as Git bring in additional benefits of security, privacy and user ownership of data.

While Git is designed for line-based editing of text files and requires manual intervention in the presence of merge conflicts, RDTs generalise this concept to arbitrary general-purpose data structures such as lists and hash maps, and ensure convergence without manual intervention. Convergent Replicated Data Types (CRDTs) [32], which arose from distributed systems research, are complete reimplementations of sequential counterparts aimed at providing convergence without user intervention, and have been deployed in distributed databases such as AntidoteDB [31] and Riak [29].

To resolve conflicting updates, CRDTs generally need to carry their causal contexts as metadata [35]. Managing this causal context is often expensive and complicated. For example, consider the observed-removed set CRDT (OR-set) [32], where, in the case of concurrent addition and removal, the addition wins. A typical OR-set implementation uses two grow-only sets, one for elements added to the set $\mathcal{A}$ and

another for elements that are removed $\mathcal{R}$. An element e is removed from the OR-set by adding it to the set $\mathcal{R}$, and thus creating a *tombstone* for e. The set membership is given by the difference between the two: $\mathcal{A} - \mathcal{R}$, and two concurrent versions can be merged by unioning the individual $\mathcal{A}$ and $\mathcal{R}$ sets. Observe that the tombstones for removed elements cannot be garbage collected as that would require all the replicas to remove the element at the same time, which requires global coordination. This leads to an inefficient implementation. Several techniques have been proposed to minimise this metadata overhead [1, 35], but the fundamental problem remains.

1.1 Mergeable Replicated Data Types

As an alternative to CRDTs, mergeable replicated data types (MRDTs) [18] have been proposed, which extend the idea of distributed version control for arbitrary data types. The causal context necessary for resolving the conflicts is maintained by the MRDT middleware. MRDTs allow ordinary purely functional data structures [27] to be promoted to RDTs by equipping them with a three-way merge function that describes the conflict resolution policy. When conflicting updates need to be reconciled, the causal history is used to determine the lowest common ancestor (lca) for use in the three-way merge function along with the conflicting states. The MRDT middleware garbage collects the causal histories when appropriate [8], and is no longer a concern for the RDT library developer. This branch-consistent view of replication not only makes it easier to develop individual data types but also leads to a natural transactional semantics [6, 9].

An efficient OR-set MRDT that avoids tombstones can be implemented as follows. We represent the OR-set as a list of pairs of the element and a unique id, which is generated per operation. The list may have duplicate elements with different ids. Adding an element appends the element and the id pair to the head of the list ($O(1)$ operation). Removing an element removes all the occurrences of the element from the list ($O(n)$ operation). Given two concurrent versions of the OR-set a and b, and their lowest common ancestor l, the merge is implemented as $(a - l)$ @ $(b - l)$ @ $(l \cap a \cap b)$, where @ stands for list append. Intuitively, we append the lists formed by newly added elements in a and b with the list of elements that are present in all three versions. The unique id associated with the element ensures that in the presence of concurrent addition and removal of the same element, the newly added element with the fresh id, which has not been seen by the concurrent remove, will remain in the merged result. The merge operation can be implemented in $O(n \log n)$ time by sorting the individual lists. In §2.1.2, we show how to make this implementation even more efficient by removing the duplicate elements with different ids from the OR-set.

1.2 Efficiency and Correctness

The key question is how do we guarantee that such efficient implementations still preserve the intent of the OR-Set in a sound manner? Optimisations such as removing duplicate elements are notoriously difficult to get right since the replica states evolve independently. Moreover, individually correct RDTs may fail to preserve convergence when putting together [19]. Kaki et al. [18] (Quark) opine that merge functions should not be written by hand, but automatically derived from a relational representation of the sequential data type. Their idea is to capture the key properties of the algebraic data type as relations over its constituent elements. Then, the merge function devolves to a merge of these relations (sets) expressed as MRDTs. During the merge, the concrete implementations are reified to their relational representations expressed in terms of sets, merged using set semantics, and the final concrete state is reconstructed from the relational set representation.

Unfortunately, mapping complex data types to sets do not lead to efficient implementations. For example, a queue in Kaki et al. is represented by two characteristic relations – a unary relation for membership and a binary relation for ordering. For a queue with n elements, the ordering relation contains n^2 elements. Reifying the queue to its characteristic relations and back to its concrete representation for every merge is inefficient and impractical. This technique does not scale as the structure of the data type gets richer (Red-Black tree, JSON, file systems, etc.). The more complex the data type, the more complex the characteristic relations become, having an impact on the cost of the merge. Further Kaki et al. do not consider the functional correctness of MRDT implementations, but instead only focuses on the correctness of convergence.

1.3 Certified MRDTs

Precisely specifying and verifying the functional correctness of efficient RDT implementations is not straightforward due to the complexity of handling conflicts between divergent versions. This results in a huge gap between the high-level specifications and efficient implementations. In this work, we propose to bridge this gap by using Burckhardt et al.'s *replication-aware simulation relation* [5]. However, Burckhardt et al.'s simulation is only applicable to CRDTs and cannot be directly extended to MRDTs which assume a different system model.

We first propose a system model and operational semantics for MRDTs, and precisely define the problem of convergence and functional correctness for MRDTs. We also introduce a new notion of *convergence modulo observable behaviour*, which allows replicas to converge to different states, as long as their observable behaviour to clients remains the same. This notion allows us to build and verify even more efficient MRDTs.

Further, we go beyond Burckhardt et al.'s work [5] in the use of simulation relations by mechanizing and automating (to an extent) the complete verification process. We instantiate our technique as an F* library named Peepul and mechanically verify the implementation of several efficient purely functional MRDT implementations including an efficient replicated two-list queue. Our replicated queue supports constant time push and pop operations, a linear time merge operation, and does not have any tombstones. To the best of our knowledge, ours is the first formal declarative specification of a distributed queue (§6), and its mechanised proof of correctness.

Being an SMT-solver-aided programming language, F* allows us to discharge many of the proof obligations automatically through the SMT solver. Even though our approach requires the simulation relation as input, we also observe that in most cases, the simulation relation directly follows from the declarative specification.

Our technique also supports composition, and we demonstrate how parametric polymorphism allows composition of not just the MRDT implementations but also their proofs of correctness. From our MRDT implementations in F*, we extract verified OCaml implementations and execute them on top of Irmin, a Git-like distributed database. Our experimental evaluation shows that our efficient MRDT implementations scale significantly better than other RDT implementations.

To summarize, we make the following contributions:

- We propose a store semantics for MRDT implementations and formally define the convergence and functional correctness problem for MRDTs, including a new notion of convergence modulo observable behaviour.
- We propose a technique to verify both convergence and functional correctness of MRDTs by adapting the notion of replication-aware simulation relation [5] to the MRDT setting.
- We mechanize and automate the complete verification process using F*, and apply our technique to several complex MRDT implementations, including a new time and space-efficient OR-set and a queue MRDT.
- We provide experimental results which demonstrate that our efficient MRDT implementations perform much better as compared with previous implementations, and also show the tradeoff between proof automation and verification time in F*.

The rest of the paper is organized as follows. §2 presents the implementation model and the declarative specification framework for MRDTs. §3 presents the formal semantics of the git-like replicated store on which MRDT implementations run. In §4, we present a new verification strategy for MRDTs based on the notion of replication-aware simulation. §5 highlights the compositionality of our technique in verifying complex verified MRDTs by reusing the proofs of simpler ones. §6 presents the formally verified efficient replicated queue. §7 presents the experimental evaluation and §8 presents the related work.

2 Implementing and Specifying MRDTs

In this section, we present the formal model for describing MRDT implementations and their specifications.

2.1 Implementation

Our model of replicated datastore is similar to a distributed version control system like Git [11], with replication centred around versioned states in branches and explicit merges. A typical replicated datastore will have a key-value interface with the capability to store arbitrary objects as values [16, 29]. Since our goal is to verify correct implementations of individual replicated objects, our formalism models a store with a single object.

A replicated datastore consists of an object which is replicated across multiple ***branches*** $b_1, b_2, \ldots \in branchID$. Clients interact with the store by performing ***operations*** on the object at a specified branch, modifying its local state. The different branches may concurrently update their local states and progress independently. We also allow the dynamic creation of a new branch by copying the state of an existing branch. A branch at any time can get updates from any other branch by performing a ***merge*** with that branch, updating its local copy to reflect the merge. Conflicts might arise when the same object is modified in two or more branches, and these are resolved in a datatype-specific way.

An object has a type $\tau \in Type$, whose ***type signature*** (Op_τ, Val_τ) determines the set of supported operations Op_τ and the set of their return values Val_τ. A special value $\perp \in Val_\tau$ is used for operations that return no value.

Definition 2.1. A ***mergeable replicated data type (MRDT) implementation*** for a data type τ is a tuple $D_\tau = (\Sigma, \sigma_0, do, merge)$ where:

- Σ is the set of all possible states at a branch,
- $\sigma_0 \in \Sigma$ is the initial state,
- $do : Op_\tau \times \Sigma \times Timestamp \to \Sigma \times Val_\tau$ implements every data type operation,
- $merge : \Sigma \times \Sigma \times \Sigma \to \Sigma$ implements the three-way merge strategy.

An MRDT implementation $\mathcal{D}_\tau$ provides two *methods*: *do* and *merge* that the datastore will invoke appropriately. We assume that these methods execute atomically. A client request to perform an operation $o \in Op_\tau$ at a branch triggers the call $do(o, \sigma, t)$. This takes the current state $\sigma \in \Sigma$ of the object at the branch where the request is issued and a timestamp $t \in Timestamp$ provided by the datastore, and produces the updated object state and the return value of the operation.

The datastore guarantees that the timestamps are unique across all of the branches, and for any two operations a and b,

with timestamps t_a and t_b, if a happens-before b, then $t_a < t_b$. The data type implementation can use the timestamp provided to implement the conflict-resolution strategy but is also free to ignore it. For simplicity of presentation, we assume that the timestamps are positive integers, $Timestamp = \mathbb{N}$. The datastore may choose to implement the timestamp using Lamport clocks [21], along with the unique branch id to provide uniqueness of timestamps.

A branch a may get updates from another branch b by performing a merge, which modifies the state of the object in branch a. In this case, the datastore will invoke $merge(\sigma_{lca}, \sigma_a, \sigma_b)$ where σ_a and σ_b are the current states of branch a and b respectively, and σ_{lca} is the lowest common ancestor (LCA) of the two branches. The LCA of the two branches is the most recent state from which the two branches diverged. We assume that the execution of the store will begin with a single branch, from which new branches may be dynamically created. Hence, for any two branches, the LCA will always exist.

2.1.1 OR-set. We illustrate MRDT implementations using the example of an OR-set. Recall from §1 that the OR-set favours the addition in the case where there is a concurrent addition and removal of the same element on different branches.

1: $\Sigma = \mathcal{P}(\mathbb{N} \times \mathbb{N})$
2: $\sigma_0 = \{\}$
3: $do(rd, \sigma, \_) = (\sigma, \{a \mid (a, \_) \in \sigma\})$
4: $do(add(a), \sigma, t) = (\sigma \cup \{(a, t)\}, \bot)$
5: $do(remove(a), \sigma, t) = (\{e \in \sigma \mid fst(e) \neq a\}, \bot)$
6: $merge(\sigma_{lca}, \sigma_a, \sigma_b) =$
$(\sigma_{lca} \cap \sigma_a \cap \sigma_b) \cup (\sigma_a - \sigma_{lca}) \cup (\sigma_b - \sigma_{lca})$

Figure 1. OR-set data type implementation

Let us assume that the elements in the OR-set are natural numbers. Its type signature would be $(Op_{orset}, Val_{orset}) = (\{\text{add}(a), \text{remove}(a) \mid a \in \mathbb{N}\} \cup \{\text{rd}\}, \{\mathcal{P}(\mathbb{N}), \bot\})$. Figure 1 shows an MRDT implementation of the OR-set data type. The state of the object is a set of pairs of the element and the timestamp. The operations and the merge remain the same as described in §1.1. Note that we use fst and snd functions to obtain the first and second elements respectively from a tuple. This implementation may have duplicate entries of the same element with different timestamps.

2.1.2 Space-efficient OR-set (OR-set-space). One possibility to make this OR-set implementation more space-efficient is by removing the duplicate entries from the set. A duplicate element will appear in the set if the client calls add(e) for an element e which is already in the set. Can we reimplement add such that we leave the set as is if the set already has e? Unfortunately, this breaks the intent of the OR-set. In particular, if there were a concurrent removal of e on a different branch, then e will be removed when the branches are merged. The key insight is that the effect of the duplicate add has to be recorded to not lose additions.

1: $\Sigma = \mathcal{P}(\mathbb{N} \times \mathbb{N})$
2: $\sigma_0 = \{\}$
3: $do(rd, \sigma, \_) = (\sigma, \{a \mid (a, \_) \in \sigma\})$
4: $do(add(a), \sigma, t) =$ if $(a, \_) \in \sigma$ then $(\sigma[a \mapsto t], \bot)$
5: else $(\sigma \cup \{(a, t)\}, \bot)$
6: $do(remove(a), \sigma, t) = (\{e \in \sigma \mid fst(e) \neq a\}, \bot)$
7: $merge(\sigma_{lca}, \sigma_a, \sigma_b) =$
8: $\{e \mid e \in (\sigma_{lca} \cap \sigma_a \cap \sigma_b)\} \cup$
9: $\{e \mid e \in (\sigma_a - \sigma_{lca}) \wedge (fst(e), \_) \notin (\sigma_b - \sigma_{lca})\} \cup$
10: $\{e \mid e \in (\sigma_b - \sigma_{lca}) \wedge (fst(e), \_) \notin (\sigma_a - \sigma_{lca})\} \cup$
11: $\{e \mid e \in (\sigma_a - \sigma_{lca}) \wedge$
12: $(\forall t.\ (fst(e), t) \in (\sigma_b - \sigma_{lca}) \Rightarrow snd(e) > t)\} \cup$
13: $\{e \mid e \in (\sigma_b - \sigma_{lca}) \wedge$
14: $(\forall t.\ (fst(e), t) \in (\sigma_a - \sigma_{lca}) \Rightarrow snd(e) > t)\}$

Figure 2. Space-efficient OR-set (OR-set-space) implementation

Figure 2 provides the implementation of the space-efficient OR-set. The read and the remove operations remain the same as the earlier implementation. If the element being added is not already present in the set, then the element is added to the set along with the timestamp. Otherwise, the timestamp of the existing entry is updated to the new timestamp. Given that our timestamps are unique, the new operation's timestamp will be distinct from the old timestamp. This prevents a concurrent remove from deleting this new addition.

Another possibility of duplicates is that the same element may concurrently be added on two different branches. The implementation of the merge function now has to take care of this possibility and not include duplicates. An element in the merged set was either in the LCA and the two concurrent states (line 8), or was only added in one of the branches (lines 9 and 10), or was added in both the branches in which case we pick the entry with the larger timestamp (lines 11–14).

2.2 Specification

Given that there are several candidates for implementing an MRDT, we need a way to specify the behaviour of an MRDT so that we may ask the question of whether the given implementation satisfies the specification. We now present a declarative framework for specifying MRDTs which closely follows the framework presented by Burckhardt et al. [5]. We define our specifications on an *abstract state*, which capture the state of the distributed store. It consists of *events* in execution of the distributed store, along with a *visibility* relation among them.

Definition 2.2. An ***abstract state*** for a data type $\tau = (Op_\tau, Val_\tau)$ is a tuple $I = \langle E, oper, rval, time, vis \rangle$, where

- $E \subseteq Event$ is a set of events,
- $oper : E \to Op_\tau$ associates the data type operation with each event,
- $rval : E \to Val_\tau$ associates the return value with each event,
- $time : E \to Timestamp$ associates the timestamp at which an event was performed,
- $vis \subseteq E \times E$ is an irreflexive, asymmetric and transitive ***visibility relation***.

Given $e \xrightarrow{vis} f$, e is said to causally precede f. In our setting, it may be the case that the operation of f follows the operation of e on the same branch, or the operations of f and e were performed on different branches b_f and b_e, but before the operation of f, the branch b_e on which the operation of e was performed was merged info b_f.

We specify a data type τ by a function $\mathcal{F}_\tau$ which determines the return value of an operation o based on prior operations applied to that object. $\mathcal{F}_\tau$ also takes as a parameter the abstract state that is visible to the operation. Note that the abstract state contains all the information that is necessary to specify the return value of o.

Definition 2.3. A ***replicated data type specification*** for a type τ is a function $\mathcal{F}_\tau$ that given an operation $o \in Op_\tau$ and an abstract state I for τ, specifies a return value $\mathcal{F}_\tau(o, I) \in Val_\tau$.

OR-set specification. As an illustration of the specification language, let us consider the OR-set. For the OR-set, both add and remove operations always return $\bot$. We can formally specify the 'add-wins' conflict resolution strategy as follows:

$$\mathcal{F}_{orset}(\mathsf{rd}, \langle E, oper, rval, time, vis\rangle) = \{a \mid \exists e \in E.\ oper(e) = \mathsf{add}(a) \wedge \neg(\exists f \in E.\ oper(f) = \mathsf{remove}(a) \wedge e \xrightarrow{vis} f)\}$$

In words, the read operation returns all those elements for which there exists an add operation of the element which is not visible to a remove operation of the same element. Hence, if an add and remove operation are concurrent, then the add would win. Notice that the specification, while precisely encoding the required semantics, is far removed from the MRDT implementations of the OR-set that we saw earlier. Providing a framework for bridging this gap in an automated and mechanized manner is one of the principal contributions of this work.

3 Store Semantics and MRDT Correctness

In this section, we formally define the semantics of a replicated datastore $\mathbb{S}$ consisting of a single object with data type implementation $\mathcal{D}_\tau$. Note that the store semantics can be easily generalized to multiple objects (with possibly different data types) since the store treats each object independently. We then define formally what it means for data type implementations to satisfy their specifications. We also introduce a novel notion of convergence across all the branches called *convergence modulo observable behaviour* that differs from the standard notions of eventual consistency. This property allows us to have more efficient but verified merges.

The ***semantics*** of the store is a set of all its executions. In order to easily relate the specifications which are in terms of abstract states to the implementation, we maintain both the concrete state (as given by the data type implementation) and the abstract state at every branch in our store semantics. Formally, the semantics of the store are parametrised by a data type τ and its implementation $D_\tau = (\Sigma, \sigma_0, do, merge)$. They are represented by a labelled transition system $\mathcal{M}_{D_\tau} = (\Phi, \to)$. Assume that $\mathcal{B}$ is the set of all possible branches. Each state in Φ is a tuple (ϕ, δ, t) where,

- $\phi : \mathcal{B} \rightharpoonup \Sigma$ is a partial function that maps branches to their concrete states,
- $\delta : \mathcal{B} \rightharpoonup I$ is a partial function that maps branches to their abstract states,
- $t \in Timestamp$ maintains the current timestamp to be supplied to operations.

The initial state of the labelled transition system consists of only one branch $b_\bot$, and is represented by $C_\bot = (\phi_\bot, \delta_\bot, 0)$ where $\phi_\bot = [b_\bot \mapsto \sigma_0]$ and $\delta_\bot = [b_\bot \mapsto I_0]$.

Here, σ_0 is the initial state as given by the implementation D_τ, while I_0 is the empty abstract state, whose event set is empty. In order to describe the transition rules, we first introduce abstract operations $do^\#$, $merge^\#$ and $lca^\#$ which perform a data type operation, merge operation and find the lowest common ancestor respectively on abstract states:

$$do^\#\langle I, e, op, a, t\rangle = \langle I.E \cup \{e\}, I.oper[e \mapsto op], I.rval[e \mapsto a], I.time[e \mapsto t], I.vis \cup \{(f, e) \mid f \in I.E)\}\rangle$$

$$merge^\#(I_a, I_b) = I_m \text{ where}$$
$$I_m.E = I_a.E \cup I_b.E$$
$$\mathsf{prop} \in \{oper, rval, time\}$$
$$I_m.\mathsf{prop}(e) = \begin{cases} I_a(e) & \text{if } e \in I_a.E \\ I_b(e) & \text{if } e \in I_b.E \end{cases}$$
$$I_m.vis = I_a.vis \cup I_b.vis$$

$$lca^\#(I_a, I_b) = \langle I_a.E \cap I_b.E, I_a.oper\mid_{E_l}, I_a.rval\mid_{E_l}, I_a.time\mid_{E_l}, I_a.vis\mid_{E_l}\rangle$$

In terms of abstract states, $do^\#$ simply adds the new event e to the set of events, appropriately setting the various event properties and visibility relation. $merge^\#$ of two abstract states simply takes a union of the events in the two states. Similarly, the $lca^\#$ of two abstract states would be the intersection of events in the two states.

Figure 3 describes the transition function $\to$. The first rule describes the creation of a new branch b_2 from the current branch b_1. Both the concrete and abstract states of the new

$$\frac{b_1 \in dom(\phi) \qquad b_2 \notin dom(\phi) \qquad \phi' = \phi[b_2 \mapsto \phi(b_1)] \qquad \delta' = \delta[b_2 \mapsto \delta(b_1)]}{(\phi, \delta, t) \xrightarrow{CREATEBRANCH(b_1,b_2)} (\phi', \delta', t)}$$

$$\frac{b \in dom(\phi) \qquad \mathcal{D}_\tau.do(o, \phi(b), t) = (\sigma', a) \qquad e : \{oper = o, time = t, rval = a\} \qquad do^\#(\delta(b), e, o, a, t) = I' \qquad \phi' = \phi[b \mapsto \sigma'] \qquad \delta' = \delta[b \mapsto I']}{(\phi, \delta, t) \xrightarrow{DO(o,b)} (\phi', \delta', t+1)}$$

$$\frac{b_1 \in dom(\phi) \qquad b_2 \in dom(\phi) \qquad lca \in dom(\phi) \qquad \delta(lca) = lca^\#(\delta(b_1), \delta(b_2)) \qquad \mathcal{D}_\tau.merge(\phi(lca), \phi(b_1), \phi(b_2)) = \sigma_{merge} \qquad merge^\#(\delta(b_1), \delta(b_2)) = I_{merge} \qquad \phi' = \phi[b_1 \mapsto \sigma_{merge}] \qquad \delta' = \delta[b_1 \mapsto I_{merge}]}{(\phi, \delta, t) \xrightarrow{MERGE(b_1,b_2)} (\phi', \delta', t)}$$

Figure 3. Semantics of the replicated datastore

branch will be the same as that of b_1. The second rule describes a branch b performing an operation o which triggers a call to the *do* method of the corresponding data type implementation. The return value is recorded using the function *rval*. A similar update is also performed on the abstract state of branch b using $do^\#$. The third rule describes the merging of branch b_2 into branch b_1 which triggers a call to the *merge* method of the data type implementation. We assume that the store provides another branch *lca* whose abstract and concrete states correspond to the lowest common ancestor of the two branches.

Definition 3.1. An ***execution*** χ of $\mathcal{M}_{D_\tau}$ is a finite but unbounded sequence of transitions starting from the initial state $C_\perp$.

$$\chi = (\phi_\perp, \delta_\perp, 0) \xrightarrow{e_1} (\phi_1, \delta_1, t_1) \xrightarrow{e_2} \ldots \xrightarrow{e_n} (\phi_n, \delta_n, t_n) \quad (1)$$

Definition 3.2. An execution χ ***satisfies*** the specification $\mathcal{F}_\tau$ for the data type τ, written as $\chi \models \mathcal{F}_\tau$, if for every DO transition $(\phi_i, \delta_i, t_i) \xrightarrow{DO(o,b)} (\phi_{i+1}, \delta_{i+1}, t_i + 1)$ in χ, such that $\mathcal{D}_\tau.do(o, \phi_i(b), t_i) = (\sigma, a)$, then $a = \mathcal{F}_\tau(o, \delta_i(b))$.

That is for every operation o, the return value a computed by the implementation on the concrete state must be equal to the return value of the specification function $\mathcal{F}_\tau$ computed on the abstract state. Next, we define the notion of convergence (i.e. strong eventual consistency) in our setting:

Definition 3.3. An execution χ (as in equation 1) is ***convergent***, if for every state (ϕ_i, δ_i) and

$$\forall b_1, b_2 \in dom(\phi_i).\delta_i(b_1) = \delta_i(b_2) \implies \phi_i(b_1) = \phi_i(b_2)$$

That is, two branches with the same abstract states–which corresponds to having seen the same set of events–must also have the same concrete state. We note that even though eventual consistency requires two branches to converge to the same state, from the point of view of a client that uses the data store, this state is never directly visible. That is, a client only notices the operations and their return values. Based on this insight, we define the notion of observational equivalence between states, and a new notion of convergence modulo observable behaviour that requires branches to converge to observationally equivalent states.

Definition 3.4. Two states σ_1 and σ_2 are ***observationally equivalent***, written as $\sigma_1 \sim \sigma_2$, if the return value of every operation supported by the data type applied on the two states is the same. Formally,

$$\forall \sigma_1, \sigma_2 \in \Sigma.\ \forall o \in Op_\tau.\ \forall t_1, t_2 \in Timestamp.\ \exists a \in Val_\tau.\ \mathcal{D}_\tau.do(o, \sigma_1, t_1) = (\_, a) \wedge \mathcal{D}_\tau.do(o, \sigma_2, t_2) = (\_, a) \implies \sigma_1 \sim \sigma_2$$

Definition 3.5. An execution χ (as in equation 1) is ***convergent modulo observable behavior***, if for every state (ϕ_i, δ_i) and

$$\forall b_1, b_2 \in dom(\phi_i).\delta_i(b_1) = \delta_i(b_2) \implies \phi_i(b_1) \sim \phi_i(b_2) \quad (2)$$

The idea behind convergence modulo observable behaviour is that the state of the object at different replicas may not converge to the same (structurally equal) representation, but the object has the same observable behaviour in terms of its operations. For example, in the OR-set implementation, if the set is implemented internally as a binary search tree (BST), then branches can independently decide to perform balancing operations on the BST to improve the complexity of the subsequent read operations. This would mean that the actual state of the BSTs at different branches may eventually not be structurally equal, but they would still contain the same set of elements, resulting in the same observable behaviour. Note that the standard notion of eventual consistency implies convergence modulo observable behaviour.

Definition 3.6. A data type implementation $\mathcal{D}_\tau$ is ***correct***, if every execution χ of $\mathcal{M}_{D_\tau}$ satisfies the specification $\mathcal{F}_\tau$ and is convergent modulo observable behavior.

4 Proving Data Type Implementations Correct

In the previous section, we have defined what it means for an MRDT implementation to be correct concerning the specification. In this section, we show how to prove the correctness of an MRDT implementation with the help of replication-aware simulation relations.

4.1 Replication-Aware Simulation

For proving the correctness of a data type implementation $\mathcal{D}_\tau$, we use ***replication-aware simulation relations*** $\mathcal{R}_{sim}$. While similar to the simulation relations used in Burckhardt et al. [5], in this work, we apply them to MRDTs rather than CRDTs. Further, we also mechanize and automate simulation-based proofs by deriving simple sufficient conditions which can easily be discharged by tools such as F*. Finally, we apply our proof technique to a wide range of MRDTs, with substantially complex specifications (e.g. queue MRDT described in §6).

The $\mathcal{R}_{sim}$ relation essentially associates the concrete state of the object at a branch b with the abstract state at the branch. This abstract state would consist of all events which were applied to the branch. Verifying the correctness of an MRDT through simulation relations involves two steps: (i) first, we show that the simulation relation holds at every transition in every execution of the replicated store, and (ii) the simulation relation meets the requirements of the data type specification and is sufficient for convergence. The first step is essentially an inductive argument, for which we require the simulation relation between the abstract and concrete states to hold for every data type operation instance and merge instance. These two steps are depicted pictorially in figures 4 and 5, respectively.

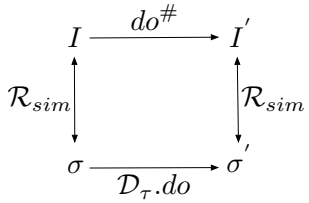

Figure 4. Verifying operations

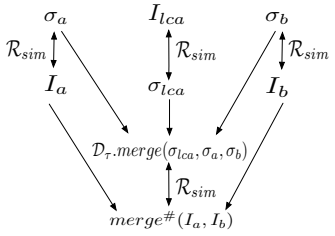

Figure 5. Verifying 3-way merge

Figure 4 considers the application of a data type operation (through the *do* function) at a branch. Assuming that the simulation relation $\mathcal{R}_{sim}$ holds between the abstract state I and the concrete state σ at the branch, we would have to show that $\mathcal{R}_{sim}$ continues to hold after the application of the operation through the concrete *do* function of the implementation and the abstract $do^\#$ function on the abstract state.

Figure 5 considers the application of a merge operation between branches a and b. In this case, assuming $\mathcal{R}_{sim}$ between the abstract and concrete states at the two branches and for the LCA, we would then show that $\mathcal{R}_{sim}$ continues to hold between the concrete and abstract states obtained after merge. Note that since the concrete merge operation also uses the concrete LCA state σ_{lca}, we also assume that $\mathcal{R}_{sim}$ holds between the concrete and abstract LCA states.

These conditions consider the effect of concrete and abstract operations locally and thus enable automated verification. To discharge these conditions, we also consider two store properties, Ψ_{ts} and Ψ_{lca} that hold across all executions (shown in Table 1). Ψ_{ts} pertains to the nature of the timestamps associated with each operation, while Ψ_{lca} characterizes the lowest common ancestor used for merge. These properties hold naturally due to the semantics of the replicated store. These properties play an important role in discharging the conditions required for the validity of the simulation relation.

In particular, $\Psi_{ts}(I)$ asserts that in the abstract state I, causally related events have increasing timestamps, and no two events have the same timestamp. $\Psi_{lca}(I_l, I_a, I_b)$ will be instantiated with the LCA of two abstract states I_a and I_b (i.e. $I_l = lca^\#(I_a, I_b)$), and asserts that the visibility relation between events which are present in both I_a and I_b (and hence also in I_l) will be the same in all three abstract states. These properties follow naturally from the definition of LCA and are also maintained by the store semantics.

Table 2 shows the conditions required for proving the validity of the simulation relation $\mathcal{R}_{sim}$. In particular, Φ_{do} and Φ_{merge} exactly encode the scenarios depicted in the figures 4 and 5. Note that for Φ_{do}, we assume Ψ_{ts} for the input abstract state on which the operation will be performed. Similarly, for Φ_{merge}, we assume Ψ_{ts} for all events in the merged abstract state (thus ensuring Ψ_{ts} also holds for events in the original branches) and Ψ_{lca} for the LCA of the abstract states.

Once we show that the simulation relation is maintained at every transition in every execution inductively, we also have to show that it is strong enough to imply the data type specification as well as guarantee convergence. For this, we define two more conditions Φ_{spec} and Φ_{con} (also in table 2). Φ_{spec} says that if abstract state I and concrete state σ are related by $\mathcal{R}_{sim}$, then the return value of operation o performed on σ should match the value of the specification function $\mathcal{F}_\tau$ on the abstract state. Since the $\mathcal{R}_{sim}$ relation is maintained at every transition, if Φ_{spec} is valid, then the implementation will satisfy the specification. Finally, for convergence, we require that if two concrete states are related to the same abstract state, then they should be observationally equivalent. This corresponds to our proposed notion of convergence modulo observable behaviour.

Table 1. Store properties

$\Psi_{ts}(I)$	$\forall e, e' \in I.E.\ e \xrightarrow{I.vis} e' \Rightarrow I.time(e) < I.time(e')$ $\wedge \forall e, e' \in I.E.\ I.time(e) = I.time(e') \Rightarrow e = e'$
$\Psi_{lca}(I_l, I_a, I_b)$	$I_l.E = I_a.E \cap I_b.E$ $\wedge\ I_l.vis = I_a.vis_{\|I_l.E} = I_b.vis_{\|I_l.E}$

Table 2. Sufficient conditions for showing validity of simulation relation

$\Phi_{do}(\mathcal{R}_{sim})$	$\forall I, \sigma, e, op, a, t.\ \mathcal{R}_{sim}(I, \sigma) \wedge do^{\#}(I, e, op, a, t) = I'$ $\wedge\ \mathcal{D}_{\tau}.do(op, \sigma, t) = (\sigma', a) \wedge \Psi_{ts}(I) \implies \mathcal{R}_{sim}(I', \sigma')$
$\Phi_{merge}(\mathcal{R}_{sim})$	$\forall I_a, I_b, \sigma_a, \sigma_b, \sigma_{lca}.\ \mathcal{R}_{sim}(I_a, \sigma_a) \wedge \mathcal{R}_{sim}(I_b, \sigma_b)$ $\wedge\ \mathcal{R}_{sim}(lca^{\#}(I_a, I_b), \sigma_{lca}) \wedge \Psi_{ts}(merge^{\#}(I_a, I_b)) \wedge \Psi_{lca}(lca^{\#}(I_a, I_b), I_a, I_b)$ $\implies \mathcal{R}_{sim}(merge^{\#}(I_a, I_b), \mathcal{D}_{\tau}.merge(\sigma_{lca}, \sigma_a, \sigma_b))$
$\Phi_{spec}(\mathcal{R}_{sim})$	$\forall I, \sigma, e, op, a, t.\ \mathcal{R}_{sim}(I, \sigma) \wedge\ \mathcal{D}_{\tau}.do(op, \sigma, t) = (\sigma', a)$ $\wedge\ \Psi_{ts}(I) \implies a = \mathcal{F}_{\tau}(op, I)$
$\Phi_{con}(\mathcal{R}_{sim})$	$\forall I, \sigma_a, \sigma_b.\ \mathcal{R}_{sim}(I, \sigma_a) \wedge \mathcal{R}_{sim}(I, \sigma_b) \implies \sigma_a \sim \sigma_b$

Definition 4.1. Given a MRDT implementation $\mathcal{D}_{\tau}$ of data type τ, a replication-aware simulation relation $\mathcal{R}_{sim} \subseteq \mathcal{I}_{\tau} \times \Sigma$ is valid if $\Phi_{do}(\mathcal{R}_{sim}) \wedge \Phi_{merge}(\mathcal{R}_{sim}) \wedge \Phi_{spec}(\mathcal{R}_{sim}) \wedge \Phi_{con}(\mathcal{R}_{sim})$.

Theorem 4.2 (Soundness)**.** *Given a MRDT implementation $\mathcal{D}_{\tau}$ of data type τ, if there exists a valid replication-aware simulation $\mathcal{R}_{sim}$, then the data type implementation $\mathcal{D}_{\tau}$ is correct* [1].

4.2 Verifying OR-sets Using Simulation Relations

Let us look at the simulation relations for verifying OR-set implementations in §2.1 against the specification $\mathcal{F}_{orset}$ in §2.2.

OR-set. Following is a candidate valid simulation relation for the unoptimized OR-set from §2.1.1:

$$\begin{aligned} \mathcal{R}_{sim}(I, \sigma) \iff (\forall a, t.\ (a, t) \in \sigma \iff \\ (\exists e \in I.E \wedge I.\ oper(e) = add(a) \wedge I.time(e) = t \wedge \\ \neg(\exists f \in I.E \wedge I.\ oper(f) = remove(a) \wedge e \xrightarrow{vis} f))) \end{aligned} \tag{3}$$

The simulation relation says that for every pair of an element and a timestamp in the concrete state, there should be an add event in the abstract state which adds the element with the same timestamp, and there should not be a remove event of the same element which witnesses that add event. This simulation relation is maintained by all the set operations as well as by the merge operation, and it also matches the OR-set specification and guarantees convergence. We use F* to automatically discharge all the proof obligations of Table 2.

[1]The proof of the soundness theorem can be found in the extended version [34] of the paper.

Space-efficient OR-set. Following is a candidate valid simulation relation for the space-efficient OR-set (OR-set-space) from §2.1.2:

$$\begin{aligned} \mathcal{R}_{sim}((E, oper, rval, time, vis), \sigma) \iff \\ (\forall a, t.\ (a, t) \in \sigma \implies (\exists e \in E.\ oper(e) = add(a) \wedge time(e) = t) \\ \wedge \neg(\exists r \in E.\ oper(r) = remove(a) \wedge e \xrightarrow{vis} r) \\ \wedge (\forall e' \in E.(\ oper(e') = add(a) \wedge \neg(\exists r \in E.oper(r) = remove(a) \\ \wedge e' \xrightarrow{vis} r)) \implies t \geq time(e'))) \wedge \\ (\forall e \in E. \forall a \in \mathbb{N}.\ oper(e) = add(a) \\ \wedge \neg(\exists r \in E.\ oper(r) = remove(a) \wedge e \xrightarrow{vis} r) \implies (a, \_) \in \sigma) \end{aligned} \tag{4}$$

The simulation relation, in this case, captures all the constraints of the one for OR-set with duplicates but has additional constraints on the timestamp of the elements in the concrete state. In particular, for an element in the concrete state, the timestamp associated with that element will be the greatest timestamp of all the add events of the same element in the abstract state, which has not been witnessed by a remove event. Finally, we also need to capture the constraint in the abstract to concrete direction. If there is an add event not seen by a remove event on the same element, then the element is a member of the concrete state. As before, the proof obligations of Table 2 are through F*.

5 Composing MRDTs

A key benefit of our technique is that compound data types can be constructed by the composition of simpler data types through parametric polymorphism. The proofs of correctness of the compound data types can be constructed from the proofs of the underlying data types.

5.1 IRC-Style Chat

To illustrate the benefits of compositionality, we consider a decentralised IRC-like chat application with multiple channels. Each channel maintains the list of messages in reverse chronological order so that the most recent message may be displayed first. For simplicity, we assume that the channels are append-only; while new messages can be posted to the channels, old messages cannot be deleted. We also assume that while new channels may be created, existing channels may not be deleted.

$\mathcal{F}_{chat}(rd(ch), \langle E, oper, rval, time, vis\rangle) = log$ where

1: $(\forall t, m.\ (t,m) \in log \iff \exists e \in E.\ oper(e) = send(ch,m) \land time(e) = t) \land$

2: $(\forall t_1, m_1, t_2, m_2.\ ord\ (t_1,m_1)\ (t_2,m_2)\ log \iff \exists e_1, e_2 \in E.\ oper(e_1) = send(ch,m_1) \land time(e_1) = t_1 \land oper(e_2) = send(ch,m_2) \land time(e_2) = t_2 \land t_1 > t_2)$

Figure 6. The specification of IRC-style chat.

The chat application supports sending a message to a channel and reading messages from a channel: $Op_{chat} = \{send(ch,m) \mid ch \in string \land m \in string)\} \cup \{rd(ch) \mid ch \in string\}$. The specification of this chat application is given in Figure 6. For this we define a predicate *ord* such that $ord(t_1,m_1)\ (t_2,m_2)\ l$ holds iff $t_1 \neq t_2$ and (t_1,m_1) occurs before (t_2,m_2) in list l. The specification essentially says the log of messages contains all (and only those) messages that were sent, and messages are ordered in reverse chronological order.

Rather than implement this chat application from scratch, we may quite reasonably build it using existing MRDTs. We may use an MRDT *map* to store the association between the channel names and the list of messages. Given that the conversations take place in a decentralized manner, the list of messages in each channel should also be mergeable. For this purpose, we use a mergeable *log*, an MRDT list that totally orders the messages based on the message timestamp, to store the messages in each of the channels. As mentioned earlier, for simplicity we will assume that the map and the log are grow-only.

5.2 Mergeable Log

The mergeable log MRDT supports operations to append messages to the log and to read the log: $Op_{log} = \{rd\} \cup \{append(m) \mid m \in string\}$. The log maintains messages in reverse chronological order. Figure 7 presents the specification, implementation and simulation relation of the mergeable log. The *sort* function sorts the list in reverse chronological order based on the timestamps associated with the messages.

$\mathcal{F}_{log}(rd, \langle E, oper, rval, time, vis\rangle) = lst$ where

1: $(\forall t, m.\ (t,m) \in lst \iff \exists e \in E.\ oper(e) = append(m) \land time(e) = t) \land$

2: $(\forall t_1, m_1, t_2, m_2.\ ord\ (t_1,m_1)\ (t_2,m_2)\ lst \iff \exists e_1, e_2 \in E.\ oper(e_1) = append(m_1) \land time(e_1) = t_1 \land oper(e_2) = append(m_2) \land time(e_2) = t_2 \land t_1 > t_2)$

$\mathcal{D}_{log} = (\Sigma, \sigma_0, do, merge_{log})$ where

1: $\Sigma_{log} = \mathcal{P}(\mathbb{N} \times string)$

2: $\sigma_0 = \{\}$

3: $do(append(m), \sigma, t) = ((t,m) :: \sigma, \bot)$

4: $do(rd, \sigma, t) = (\sigma, \sigma)$

5: $merge_{log}(\sigma_{lca}, \sigma_a, \sigma_b) = sort((\sigma_a - \sigma_{lca})\ @\ (\sigma_b - \sigma_{lca}))\ @\ \sigma_{lca}$

$\mathcal{R}_{sim-log}(I, \sigma) \iff$

1: $(\forall t, m.\ (t,m) \in \sigma \iff \exists e \in I.E.\ oper(e) = append(m) \land time(e) = t) \land$

2: $(\forall t_1, m_1, t_2, m_2.\ ord\ (t_1,m_1)\ (t_2,m_2)\ \sigma \iff \exists e_1, e_2 \in I.E.\ oper(e_1) = append(m_1) \land time(e_1) = t_1 \land oper(e_2) = append(m_2) \land time(e_2) = t_2 \land t_1 > t_2)$

Figure 7. The specification, implementation and the simulation relation of mergeable log.

$\mathcal{F}_{\alpha-map}(get(k, o_\alpha), I) =$
  let $I_\alpha = project(k, I)$ in $\mathcal{F}_\alpha(o_\alpha, I_\alpha)$

$\mathcal{D}_{\alpha-map} = (\Sigma, \sigma_0, do, merge_{\alpha-map})$ where

1: $\Sigma_{\alpha-map} = \mathcal{P}(string \times \Sigma_\alpha)$

2: $\sigma_0 = \{\}$

3: $\delta(\sigma, k) = \begin{cases} \sigma(k), & \text{if } k \in dom(\sigma) \\ \sigma_{0_\alpha}, & \text{otherwise} \end{cases}$

4: $do(set(k, o_\alpha), \sigma, t) =$
  let $(v, r) = do_\alpha(o_\alpha, \delta(\sigma, k), t)$ in $(\sigma[k \mapsto v], r)$

5: $do(get(k, o_\alpha), \sigma, t) =$
  let $(\_, r) = do_\alpha(o_\alpha, \delta(\sigma, k), t)$ in (σ, r)

6: $merge_{\alpha-map}(\sigma_{lca}, \sigma_a, \sigma_b) =$
  $\{(k, v) \mid (k \in dom(\sigma_{lca}) \cup dom(\sigma_a) \cup dom(\sigma_b)) \land v = merge_\alpha(\delta(\sigma_{lca}, k), \delta(\sigma_a, k), \delta(\sigma_b, k))$

$\mathcal{R}_{sim-\alpha-map}(I, \sigma) \iff \forall k.$

1: $(k \in dom(\sigma) \iff \exists e \in I.E.\ oper(e) = set(k, \_)) \land$

2: $\mathcal{R}_{sim-\alpha}\ (project(k, I),\ \delta(\sigma, k))$

Figure 8. The specification, implementation and simulation relation of α-map.

5.3 Generic Map

We introduce a generic map MRDT, α-map, which associates string keys with a value, where the value stored in the map is itself an MRDT. This α-map is parameterised on an

MRDT α and its implementation $\mathcal{D}_\alpha$, and supports *get* and *set* operations: $Op_{\alpha-map} = \{get(k, o_\alpha) \mid k \in string \wedge o_\alpha \in Op_\alpha\} \cup \{set(k, o_\alpha) \mid k \in string \wedge o_\alpha \in Op_\alpha\}$, where Op_α denotes the set of operations on the underlying value MRDT.

Figure 8 shows the specification, implementation and simulation relation of α-map. The implementations for *get* and *set* operations both fetch the current value associated with the key k (and the initial state of $\mathcal{D}_\alpha$ if the key is not present in the map) and apply the given operation o_α from the implementation $\mathcal{D}_\alpha$ on this value. While *set* updates the binding in the map, *get* does not do so and simply returns the value returned by o_α. The merge operation merges the values for each key using the merge function of α. The specification and simulation relation of α-map use the specification and simulation relation of the underlying MRDT α, by projecting the events associated with each key to an abstract execution of α. We now provide the details of this projection function.

5.4 Projection Function

$project\ k\ I_{\alpha-map} = I_\alpha$ where

1: $I_{\alpha-map} = (\Sigma_m, oper_m, rval_m, time_m, vis_m)$ and

2: $I_\alpha = (\Sigma_\alpha, oper_\alpha, rval_\alpha, time_\alpha, vis_\alpha)$ and

3: $(\forall e, k, o.\ e \in \Sigma_m \wedge\ oper_m(e) = set(k, o) \iff \exists e' \in \Sigma_\alpha.\ oper_\alpha(e') = o\ \wedge rval_m(e) = rval_\alpha(e') \wedge time_m(e) = time_\alpha(e')) \wedge$

4: $(\forall e_1, e_2.\ e_1 \in \Sigma_m \wedge e_2 \in \Sigma_m \wedge\ oper_m(e_1) = set(k, \_) \wedge oper_m(e_2) = set(k, \_)\ \wedge e_1 \xrightarrow{vis_m} e_2 \iff \exists e_1^{'}, e_2^{'} \in \Sigma_\alpha.\ time_\alpha(e_1^{'}) = time_m(e_1) \wedge time_\alpha(e_2^{'}) = time_m(e_2)\ \wedge e_1^{'} \xrightarrow{vis_\alpha} e_2^{'})$

Figure 9. Projection function for mapping α-map execution to α execution.

Figure 9 gives the projection function which when given an abstract execution $I_{\alpha-map}$ of α-map, projects all the *set*-events associated with a particular key k to define an abstract execution I_α. There is a one-to-one correspondence between *set*-events to k in $I_{\alpha-map}$ and events in I_α, with the corresponding events in I_α preserving the operation type, return values, timestamps and the visibility relation. The project function as used in the specification of $\mathcal{F}_{\alpha-map}$ ensures that the return value of *get*-events obey the specification $\mathcal{F}_\alpha$ as applied to the projected α-execution.

Similarly, the simulation relation of α-map requires the simulation relation of α to hold for every key, between the value associated with the key and the corresponding projected execution for the key. We can now verify the correctness of the generic α-map MRDT by relying on the correctness of α. That is if $\mathcal{R}_{sim-\alpha}$ is a valid simulation relation for the implementation $\mathcal{D}_\alpha$, then $\mathcal{R}_{sim-\alpha-map}$ is a valid simulation relation for $\mathcal{D}_{\alpha-map}$. This allows us to build the proof of correctness of α-map using the proof of correctness of α.

For our chat application, we instantiate α-map with the mergeable log $\mathcal{D}_{log}$. The chat application itself is a wrapper

$\mathcal{F}_{chat}(rd(ch), I) = \mathcal{F}_{log-map}(get(ch, rd), I)$

$\mathcal{D}_{chat} = \mathcal{D}_{log-map}$ where

1: $do(send(ch, m), \sigma, t) = do(set(ch, append(m)), \sigma, t)$

2: $do(rd(m), \sigma, t) = do(get(k, rd), \sigma, t)$

Figure 10. Implementation of IRC-style chat.

around the log-map MRDT as shown in Fig. 10. To verify the correctness of $\mathcal{D}_{chat}$, we only need to separately verify $\mathcal{D}_{\alpha-map}$ and $\mathcal{D}_{log}$. Note that one can instantiate α with any verified MRDT implementation to obtain a verified α-map MRDT.

6 Case study: A Verified Queue MRDT

Okasaki [27] describes a purely functional queue with amortized time complexity of $O(1)$ for enqueue and dequeue operations. This queue is made up of two lists that hold the front and rear parts of the queue. Elements are enqueued to the rear queue and dequeued from the front queue (both are $O(1)$ operations). If the front queue is found to be empty at dequeue, then the rear queue is reversed and made to be the front queue ($O(n)$ operation). Since each element is part of exactly one reverse operation, the enqueue and the dequeue have an amortized time complexity of $O(1)$. In this section, we show how to convert this efficient sequential queue into an MRDT by providing additional semantics to handle concurrent operations.

For simplicity of specification, we tag each enqueued element with the unique timestamp of the enqueue operation, which ensures that all the elements in the queue are unique. The queue supports two operations: $Op_{queue} = \{dequeue\} \cup \{enqueue(a) \mid a \in \mathbb{V}\}$, where $\mathbb{V}$ is some value domain. Unlike a sequential queue, we follow an *at-least-once* dequeue semantics – an element inserted into the queue may be consumed by concurrent dequeues on different branches. At-least-once semantics is common for distributed queueing services such as Amazon Simple Queue Service(SQS) [2] and RabbitMQ [28]. At a merge, concurrent enqueues are ordered according to their timestamps.

6.1 Merge Function of the Replicated Queue

To illustrate the three-way merge function, consider the execution presented in figure 11. For simplicity, we assume that the timestamps are the same as the values enqueued. Starting from the LCA, each branch performs a sequence of dequeue and enqueue operations. The resulting versions are then merged. Observe that in the merged result, the elements 1 and 2 which were dequeued (with 1 dequeued on both the branches!) are not present. Elements 3, 4 and 5 which are present in all three versions are present in the

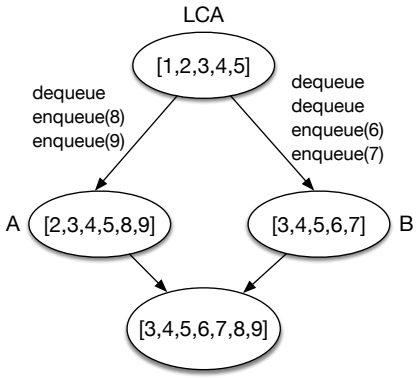

Figure 11. Three-way merge for queues

merged result. Newly inserted elements appear at the suffix, sorted according to their timestamps.

The merge function first converts each of the queues to a list and finds the longest common contiguous subsequence between the three versions (*[3,4,5]*). The newly enqueued elements are suffixes of this common subsequence – *[8,9]* and *[6,7]* in queues A and B, respectively. The final merged result is obtained by appending the common subsequence to the suffixes merged according to their timestamps. Each of these operations has a time complexity of $O(n)$ where n is the length of the longest list. Hence, the merge function is also an $O(n)$ operation [2].

6.2 Specification of the Replicated Queue

We now provide the specification for the queue MRDT, which is based on the declarative queue specification in Nagar et al. [24]. In particular, compared to the sequential queue, the only constraint that we relax is allowing multiple dequeues of the same element.

In order to describe the specification, we first introduce a number of axioms which declaratively specify different aspects of queue behaviour. Consider the match_I predicate defined for a pair of events e_1, e_2 in an abstract execution I:

$$\mathsf{match}_I(e_1, e_2) \Leftrightarrow I.oper(e_1) = enqueue(a) \\ \wedge I.oper(e_2) = dequeue \wedge a = I.rval(e_2)$$

Let EMPTY be the value returned by a dequeue when the queue is empty. We define the following axioms:

- $AddRem(I) : \forall e \in I.E.\ I.oper(e) = dequeue \wedge I.rval(e) \neq \texttt{EMPTY} \implies \exists e' \in I.E.\ \mathsf{match}_I(e', e)$
- $Empty(I) : \forall e_1, e_2, e_3 \in I.E.\ I.oper(e_1) = dequeue \wedge I.rval(e_1) = \texttt{EMPTY} \wedge I.oper(e_2) = enqueue(a) \wedge e_2 \xrightarrow{I.vis} e_1 \implies \exists e_3 \in I.E.\ \mathsf{match}_I(e_2, e_3) \wedge e_3 \xrightarrow{I.vis} e_1$
- $FIFO_1(I) : \forall e_1, e_2, e_3 \in I.E.\ I.oper(e_1) = enqueue(a) \wedge \mathsf{match}_I(e_2, e_3) \wedge e_1 \xrightarrow{I.vis} e_2 \implies \exists e_4 \in I.E.\ \mathsf{match}_I(e_1, e_4)$
- $FIFO_2(I) : \forall e_1, e_2, e_3, e_4 \in I.E.\ \neg(\mathsf{match}_I(e_1, e_4) \wedge \mathsf{match}_I(e_2, e_3) \wedge e_1 \xrightarrow{I.vis} e_2 \wedge e_3 \xrightarrow{I.vis} e_4)$

These axioms essentially encode queue semantics. *AddRem* says that for every dequeue event which does not return EMPTY, there must exist a matching enqueue event. *Empty* says that if a dequeue event returns EMPTY, there should not be an unmatched enqueue visible to it. Finally, $FIFO_1$ and $FIFO_2$ encode the first-in-first-out nature of the queue. These axioms ensure that if an enqueue event e_1 was visible to another enqueue event e_2, then the element inserted by e_1 will be dequeued first. Notice that a sequential queue would also have an injectivity axiom, which disallows multiple dequeues to be matched to an enqueue, but we do not enforce this requirement for the replicated queue.

To define $\mathcal{F}_{Queue}$, we first note that enqueue operation always returns $\perp$. For an abstract state I, $\mathcal{F}_{Queue}(dequeue, I)$ returns a such that if we add the new event e for the dequeue to the abstract state I, then the resulting abstract state $do^{\#}(I, e, dequeue, a, t)$ must satisfy all the queue axioms.

Notice how the queue axioms are substantially different from the way the MRDT queue is implemented. The simulation relation that we use to bridge this gap and relate the implementation with the abstract state is very straightforward: we simply say that for every element present in the concrete state of the queue, there must be an enqueue event without a matching dequeue. We also assert the other direction and enforce the queue axioms on the abstract state. We were able to successfully discharge the conditions for validity of the simulation relation using F*.

7 Evaluation

In this section, we evaluate the instantiation of the formalism developed thus far in Peepul, an F* library of certified efficient MRDTs. We first discuss the verification effort followed by the performance evaluation of efficient MRDTs compared to existing work. These results were obtained on a 2-socket Intel®Xeon®Gold 5120 x86-64 [15] server running Ubuntu 18.04 with 64GB of main memory.

7.1 Verification in F*

F*'s core is a functional programming language inspired by ML, with support for program verification using refinement types and monadic effects. Though F* has support for built-in effects, Peepul library only uses the pure fragment of the language. Given that we can extract OCaml code from our verified implementations in F*, we can directly utilise our MRDTs on top of Irmin [16], a Git-like distributed database, whose execution model fits the MRDT system model.

As part of the Peepul library, we have implemented and verified 9 MRDTs – increment-only counter, PN counter, enable-wins flag, last-writer-wins register, grows-only set, grows-only map, mergeable log, observed-remove set and

[2] The functional queue simulation relation and implementation can be found in the extended version [34] of the paper.

functional queue. Our specifications capture both the functional correctness of local operations as well as the semantics of the concurrent conflicting operations.

F*'s support for type classes provides a modular way to implement and verify MRDTs. The Peepul library defines an MRDT type class that captures the sufficient conditions to be proved for each MRDT as given in Table 2. This library contains 124 lines of F* code. Each MRDT is a specific instance of the type class which satisfy the conditions. It is useful to note that our MRDTs sets, maps and queues are polymorphic in their contents and may be plugged with other MRDTs to construct more complex MRDTs as seen in §5.

Table 3 tabulates the verification effort for each MRDT in the Peepul library. We include three versions of OR-sets:

- **OR-set:** the unoptimized one from §2.1.1 which uses a list for storing the elements and contains duplicates.
- **OR-set-space:** the space-optimized one from §2.1.2 which also uses a list but does not have duplicates.
- **OR-set-spacetime:** a space- and time-optimized one which uses a binary search tree for storing the elements and has no duplicates. The merge function produces a height balanced binary tree.

The lines of code represent the number of lines for implementing the data structure without counting the lines for refinements, lemmas, theorems and proofs. This is approximately the number of lines of code there will be if the data structures were implemented in OCaml. Everything else that has to do with verification is included in the lines of proofs. It is useful to note that the lines of proof for simple MRDTs such as counter and last-writer-wins (LWW) register is high compared to the lines of code since we also specify and prove their full functional correctness.

For many of the proofs, F* can automatically verify the properties either without any lemmas or just a few, thanks to F* discharging the proof obligations to the SMT solver. Most of the proofs are a few tens of lines of code except queues. In queues, the implementation is far removed from the specification, and hence, additional lemmas were necessary to bridge this gap.

F* allows the user to provide additional lemmas that help the solver arrive at the proof faster. We illustrate this for the enable-wins flag, G-set and OR-set by adding additional lemmas. Correspondingly, we observe that the verification time reduces significantly. Thanks to F*, the developers of new MRDTs in Peepul can strike a balance between verification times and manual verification effort.

In this work, we have not used F* support for tactics and interactive proofs. We believe that some of the time-consuming calls to the SMT solver may be profitably replaced by a few interactive proofs. On the whole, the choice of F* for Peepul reduces manual effort and most of the proofs are checked within a few seconds.

7.2 Performance Evaluation

In this section, we evaluate the runtime performance of efficient MRDTs in Peepul.

7.2.1 Peepul vs Quark. We first compare the performance of Peepul MRDTs against the MRDTs presented in Kaki et al. [18] (Quark). Recall that Quark lifts sequential data types to MRDTs by equipping them with a merge function, which converts the concrete state of the MRDT to a relational (set-based) representation that captures the characteristic relations of the data type. The actual merge is implemented as a merge of these sets for each of the characteristic relations. After the merging of the relational representations, the final result is obtained by a concretization function. Compared to this, Peepul merges are implemented directly on the concrete representations.

To highlight the impact of the efficient merge function in Peepul, we evaluate the performance of merge in queues. Both Peepul and Quark uses the same sequential queue representation, and the only difference is the merge function between the two. For this experiment, we start with an empty queue and perform a series of randomly generated operations with a 75:25 split between enqueues and dequeues. We use this version as the LCA and subsequently perform two different sets of operations to get the two divergent versions. We then merge these versions to measure the time taken for the merge.

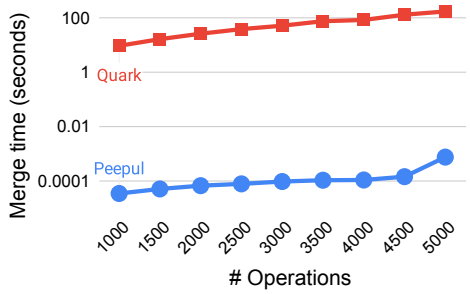

Figure 12. Merge performance of Peepul and Quark queues.

The results are reported in figure 12. For a queue, Quark needs to reify the ordering relation as a set that will contain n^2 elements for a queue of size n. In addition, there is also the cost of abstracting and concretising the queue to and from relational representation. As a result, the merge function takes 10 seconds for 1000 operations, increasing to 178 seconds for 5000 operations. On the other hand, Peepul's linear-time merge took less than a millisecond in all of the cases. This shows that Quark merge is unacceptably slow even reasonably sized queues, while Peepul remains fast and practical.

We also compare the performance of OR-set in Peepul and Quark. Since the merge function in Quark is based on automatic relational reification, Quark does not allow duplicate elements to be removed from the OR-set. To highlight

Table 3. Peepul verification effort.

MRDTs verified	#Lines code	#Lines proof	#Lemmas	Verif. time (s)
Increment-only counter	6	43	2	3.494
PN counter	8	43	2	23.211
Enable-wins flag	20	58	3	1074
		81	6	171
		89	7	104
LWW register	5	44	1	4.21
G-set	10	23	0	4.71
		28	1	2.462
		33	2	1.993
G-map	48	26	0	26.089
Mergeable log	39	95	2	36.562
OR-set (§2.1.1)	30	36	0	43.85
		41	1	21.656
		46	2	8.829
OR-set-space (§2.1.2)	59	108	7	1716
OR-set-spacetime	97	266	7	1854
Queue	32	1123	75	4753

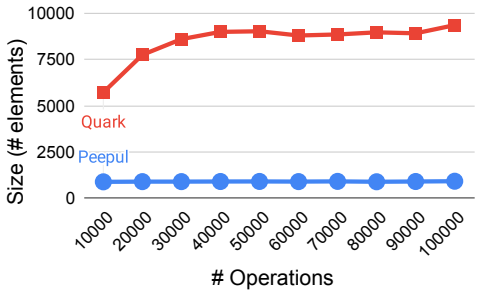

Figure 13. Performance of Peepul and Quark OR-sets.

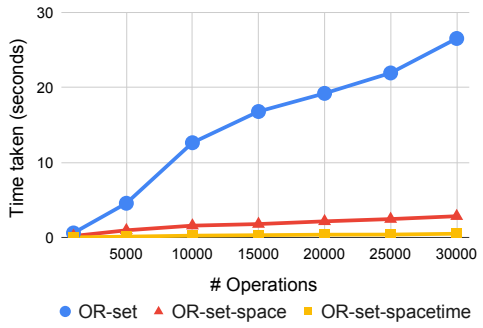

Figure 14. Running time of OR-sets.

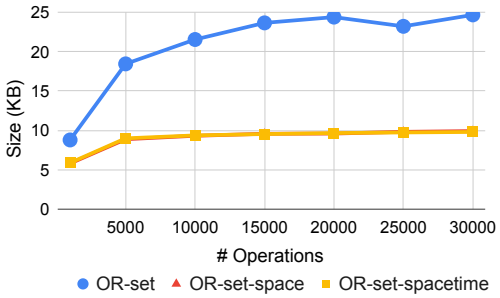

Figure 15. Space consumption of OR-sets. The OR-set-space line is hidden by the OR-set-spacetime line.

the impact of duplicate elements, we perform an experiment similar to the queue one except that we pick a 50:50 split between add and remove operations. The values added are randomly picked in the range (0:1000). For Peepul, we pick the space-optimized OR-set (OR-set-space). We report the number of elements in the final set including duplicates.

The results are presented in figure 13. Due to the duplicates, the size of the Quark set increases with an increasing number of operations; the growth is not linear due to the stochastic interplay between add and remove. For Peepul, the set size always remains below 1000 which is the range of the values picked. The results show that MRDTs in Peepul are much more efficient than in Quark.

7.2.2 Peepul OR-set Performance. We also compare the overall performance of the three OR-set implementations

in Peepul. Our workload consists of 70% lookups, 20% adds and 10% remove operations starting from an initially empty set on two different branches. We trigger a merge every 500 operations. We measure the overall execution time for the entire workload and the maximum size of the set during the execution.

The results are reported in figures 14 and 15. The results show that OR-set-spacetime is the fastest, and is around 5× faster than OR-set-space due to the fast reads and writes thanks to the binary search tree in OR-set-spacetime. Both OR-set-space and OR-set-spacetime consume a similar amount of memory. The unoptimized OR-set is both slower and consumes more memory than the other variants due to the duplicates. The results show that Peepul enables the construction of efficient certified MRDTs that have significant performance benefits compared to unoptimised ones.

8 Related Work

Reconciling concurrent updates is an important problem in distributed systems. Some of the works proposing new designs and implementations of RDTs [4, 30, 32] neither provide their formal specification nor verify them. Due to the concurrently evolving state of the replicas, informally reasoning about the correctness of even simple RDTs is tedious and error-prone. In this work, our focus is on mechanically verifying efficient implementations of MRDTs.

Several works focus on the specification and verification of CRDTs [3, 5, 12, 22, 23, 25, 36]. CRDTs typically assume a system model which involves several replicas communicating over a network with asynchronous message passing. Correspondingly, the specification and verification techniques for CRDTs will have to take into account the properties of message passing such as message ordering and delivery guarantees. On the other hand, MRDTs are described over a Git-like distributed store with branching and merging, which in turn may be implemented over asynchronous message passing. We believe that, by lifting the level of abstraction, MRDTs are easy to specify, implement and verify compared to CRDTs.

In terms of mechanised verification of RDTs, prior work has used both automated and interactive verification. Zeller et al. [36] verify state-based CRDTs with the help of interactive theorem prover Isabelle/HOL. Gomes et al. [12] develop a foundational framework for proving the correctness of operation-based CRDTs. In particular, they construct a formal network model that may delay, drop or reorder messages sent between replicas. Under these assumptions, they verify several op-based CRDTs using Isabelle/HOL. Nair et al. [25] presents an SMT-based verification tool to specify state-based CRDTs and verify invariants over its state. Nagar et al. [23] also utilise SMT-solver to automatically verify the convergence of CRDTs under different weak consistency policies. Liu et al. [22] present an extension of the SMT-solver-aided Liquid Haskell to allow refinement types on type classes and use to implement a framework to verify operation-based CRDTs. Similar to Liu et al., Peepul also uses an SMT-solver-aided programming language F*. We find that the SMT-solver-aided programming language offers a useful trade-off between manual verification effort and verification time.

Our verification framework for MRDTs builds on the concept of replication-aware simulation introduced by Burckhardt et al. [5]. Burckhardt et al. present precise specifications for RDTs and (non-mechanized) proof of correctness for a few CRDT implementations. Burckhardt et al.'s specifications are presented over the CRDT system model with explicit message passing between replicas. In this work, we lift these specifications to a higher level by abstracting out the guarantees provided by the low-level store (Ψ_{ts} and Ψ_{lca}). Further, we also observe that the simulation relation $\mathcal{R}_{sim}$ cannot be used as an inductive invariant on its own, and instead, a conjunction of $\mathcal{R}_{sim}$ with Ψ_{ts} and Ψ_{lca} is required (see conditions Φ_{do} and Φ_{merge} in Table 2). To enable mechanised verification, we identify the relationship between $\mathcal{R}_{sim}$ and the functional correctness and convergence of MRDTs. This leads to a formal specification framework that is suitable for mechanized and automated verification. We demonstrate this by successfully verifying several complex MRDT implementations in F* including the first, formally verified replicated queue.

MRDTs were first introduced by Farnier et al. [10] for Irmin [16], a distributed database built on the principles of Git. Quark [18] automatically derives merge functions for MRDTs using the invertible relational specification. However, their merge semantics focused only on convergence, and not the functional correctness of the data type. Our evaluation (§7.2.1) shows that merge through automatically derived invertible relational specification is prohibitively expensive for data types with a rich structure such as queues. Tardis [6] also uses a branch-and-merge approach to weak consistency, but does not focus on verifying the correctness of the RDTs.

Not all application logic can be expressed only using eventually consistent and convergent RDTs. For example, a replicated bank account that guarantees a non-negative balance requires coordination between concurrent withdraw operations. Several previous works have explored RDTs that utilize on-demand coordination based on application invariants [7, 13, 14, 17, 26, 33]. We leave the challenge of extending Peepul to support on-demand coordination to future work.

Acknowledgments

We thank our shepherd, Constantin Enea, and the anonymous reviewers for their reviewing effort and high-quality reviews. We also thank Aseem Rastogi and the F* Zulip community for helping us with F* related queries.

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111 (Jan 2018), 162–173. https://doi.org/10.1016/j.jpdc.2017.08.003

[2] Amazon. 2006. Simple Queue Service by Amazon. https://aws.amazon.com/sqs/

[3] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. 2016. Specification and Complexity of Collaborative Text Editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (Chicago, Illinois, USA) *(PODC '16)*. Association for Computing Machinery, New York, NY, USA, 259–268. https://doi.org/10.1145/2933057.2933090

[4] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. An optimized conflict-free replicated set. arXiv:1210.3368 [cs.DC]

[5] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 271–284. https://doi.org/10.1145/2535838.2535848

[6] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. 2016. TARDiS: A Branch-and-Merge Approach To Weak Consistency. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1615–1628. https://doi.org/10.1145/2882903.2882951

[7] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. 2021. ECROs: Building Global Scale Systems from Sequential Code. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 107 (oct 2021), 30 pages. https://doi.org/10.1145/3485484

[8] Shashank Shekhar Dubey. 2021. *Banyan: Coordination-free Distributed Transactions over Mergeable Types.* Ph. D. Dissertation. Indian Institute of Technology, Madras, India. https://thesis.iitm.ac.in/thesis?type=FinalThesis&rollno=CS17S025

[9] Shashank Shekhar Dubey, K. C. Sivaramakrishnan, Thomas Gazagnaire, and Anil Madhavapeddy. 2020. Banyan: Coordination-Free Distributed Transactions over Mergeable Types. In *Programming Languages and Systems*, Bruno C. d. S. Oliveira (Ed.). Springer International Publishing, Cham, 231–250.

[10] Benjamin Farinier, Thomas Gazagnaire, and Anil Madhavapeddy. 2015. Mergeable persistent data structures. In *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, David Baelde and Jade Alglave (Eds.). JFLA, Le Val d'Ajol, France, 1–13. https://hal.inria.fr/hal-01099136

[11] Git. 2021. Git: A distributed version control system. https://git-scm.com/

[12] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 109 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133933

[13] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

[14] Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication Coordination Analysis and Synthesis. *Proc. ACM Program. Lang.* 3, POPL, Article 74 (jan 2019), 32 pages. https://doi.org/10.1145/3290387

[15] Intel 2020. *Intel® Xeon® Gold 5120 Processor Specification.* Intel. https://ark.intel.com/content/www/us/en/ark/products/120474/intel-xeon-gold-5120-processor-19-25m-cache-2-20-ghz.html

[16] Irmin. 2021. Irmin: A distributed database built on the principles of Git. https://irmin.org/

[17] Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. 2018. Safe Replication through Bounded Concurrency Verification. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 164 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276534

[18] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable Replicated Data Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 154 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360580

[19] Martin Kleppmann. 2020. *CRDT composition failure.* University of Cambridge. https://twitter.com/martinkl/status/1327020435419041792

[20] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 154–178. https://doi.org/10.1145/3359591.3359737

[21] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[22] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. 2020. Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 216 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428284

[23] Kartik Nagar and Suresh Jagannathan. 2019. Automated Parameterized Verification of CRDTs. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, New York City, NY, USA, 459–477. https://doi.org/10.1007/978-3-030-25543-5_26

[24] Kartik Nagar, Prasita Mukherjee, and Suresh Jagannathan. 2020. Semantics, Specification, and Bounded Verification of Concurrent Libraries in Replicated Systems. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, Los Angeles, CA, USA, 251–274. https://doi.org/10.1007/978-3-030-53288-8_13

[25] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the Safety of Highly-Available Distributed Objects. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 544–571.

[26] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. 2016. The CISE Tool: Proving Weakly-Consistent Applications Correct. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data* (London, United Kingdom) *(PaPoC '16)*. Association for Computing Machinery, New York, NY, USA, Article 2, 3 pages. https://doi.org/10.1145/2911151.2911160

[27] Chris Okasaki. 1999. *Purely Functional Data Structures.* Cambridge University Press, USA.

[28] RabbitMQ. 2007. Message Brokering Service by RabbitMQ. https://www.rabbitmq.com/queues.html

[29] Riak. 2021. Resilient NoSQL Databases. https://riak.com/

[30] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *J. Parallel Distrib. Comput.* 71, 3 (March 2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

[31] Marc Shapiro, Annette Bieniusa, Nuno Preguiça, Valter Balegas, and Christopher Meiklejohn. 2018. Just-Right Consistency: reconciling availability and safety. arXiv:1801.06340 [cs.DC]

[32] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

[33] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

[34] Vimala Soundarapandian, Adharsh Kamath, Kartik Nagar, and KC Sivaramakrishnan. 2022. Certified Mergeable Replicated Data Types (Extended version). https://doi.org/10.48550/ARXIV.2203.14518

[35] Weihai Yu and Sigbjørn Rostad. 2020. A Low-Cost Set CRDT Based on Causal Lengths. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data* (Heraklion, Greece) *(PaPoC '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/3380787.3393678

[36] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal Specification and Verification of CRDTs. In *34th Formal Techniques for Networked and Distributed Systems (FORTE) (Formal Techniques for Distributed Objects, Components, and Systems, Vol. LNCS-8461)*, Erika Ábrahám and Catuscia Palamidessi (Eds.). Springer, Berlin, Germany, 33–48. https://doi.org/10.1007/978-3-662-43613-4_3 Part 1: Specification Languages and Type Systems.