Marrying Replicated and Functional Data Structures

Vimala Soundarapandian
IIT Madras, India

Adharsh Kamath
NITK Surathkal, India

Kartik Nagar
IIT Madras, India

KC Sivaramakrishnan
IIT Madras, India

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 time- and space-complexity. This is unfortunate since RDTs are often used in an local-first setting where the local operations far outweigh remote communication.

In this paper, we present Peepul, a pragmatic approach to building and verifying efficient RDTs. To make reasoning about correctness easier, we cast RDTs in the mould of 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 based technique to relate RDT specifications to their efficient purely functional implementations. We have developed 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 Git-like distributed database.

1 Introduction

Modern cloud-based software services often replicate data across multiple geographically distributed locations in order to tolerate against partial failures of servers and 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* [10] such as Git bring in additional benefits of security, privacy and user ownership of data.

PaPoC '22, April 5, 2022, Rennes, France.
.

While Git is designed for line-based editing of text files and require 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) [15], 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 data bases such as AntidoteDB [14] and Riak [13]. In order to resolve conflicting updates, CRDTs generally need to carry their causal contexts as metadata [16]. Managing this causal context is often expensive and complicated.

For example, consider the observed-removed set CRDT (OR-set) [15], 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, 16], but the fundamental problem still remains.

1.1 Mergeable Replicated Data Types

As an alternative to CRDTs, mergeable replicated data types (MRDTs) [8] 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 [12] 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 [4], 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 [3, 5].

For example, 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. 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 more efficient by removing the duplicate elements with different ids from the OR-set.

The key question is how do we guarantee that this implementation, preserves the *intent* of the OR-set? The 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 put together [9]. In [8], 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 does not lead to efficient implementations. Further [8] also does not consider functional correctness of RDTs, but instead only focuses on the convergence problem.

Precisely specifying the *intent* of data types while migrating from the sequential to the replicated world is not straightforward, as this essentially boils down to the question of handling conflicts between concurrent operations. This can be (and has been) done in many diverse ways: prioritizing one type of operation over another (e.g. OR-set), using some form of timestamps (e.g. LWW Registers), changing the semantics of an operation (e.g. lists), etc. We believe that a declarative, event-based form of specifications [2] that exposes the underlying concurrency gives more freedom to the RDT developer and also allows precise specifications. In this work, we provide the first formal declarative specification (to our best knowledge) of the queue RDT which follows 'at-least one dequeue' semantics for enqueued elements, and this relaxation is immediately obvious by looking at the specification.

2 Implementing and Specifying MRDTs

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

2.1 Implementation

We now describe our formal model for MRDT implementations, which include operations of the data type, and the protocol used to exchange updates on this object. Our model of replicated datastore is similar to distributed version systems like Git [6], with replication centered 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 [7, 13]. 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 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 an data type 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,
- $do : Op_\tau \times \Sigma \times Timestamp \rightarrow \Sigma \times Val_\tau$ implements every data type operation,
- $merge : \Sigma \times \Sigma \times \Sigma \rightarrow \Sigma$ 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, $Timestamp = \mathbb{N}$. The datastore may choose to implement the timestamp using Lamport clocks [11], 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 two branches is the most recent state from which the two branches diverged. We assume that 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.

Figure 1 shows an MRDT implementation of the OR-set data type discussed in §1.1.

```
1: $\Sigma = \mathcal{P}(\mathbb{N} \times \mathbb{N})$
2: $\sigma_0 = \{\}$
3: $do(rd, \sigma, t) = (\sigma, \{a \mid (a, t) \in \sigma\})$
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 $(Op_{orset}, Val_{orset}) = (\{\mathsf{add}(a), \mathsf{remove}(a) \mid a \in \mathbb{N}\} \cup \{\mathsf{rd}\}, \{\mathcal{P}(\mathbb{N}), \bot\})$. 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 resp. from a tuple. This implementation may have duplicate entries of the same element with different time stamps.

2.1.2 Space-efficient OR-set.

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 remove of e on a different branch, then e will be removed when the branches are merged. The effect of the duplicate add was not recorded in the state of the set, and hence, is lost. The key insight is that the effect of the duplicate add has to be recorded so as to not lose updates.

We reimplement the OR-set such that the add operation now only adds the element if the element is not already present, and otherwise, updates the timestamp of the existing entry to the new timestamp. The read and the remove operations remain the same as the earlier implementation. Given that our timestamps are unique, the new operation's timestamp will be distinct from the old time stamp. 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 takes care of this possibility by picking the entry with the larger timestamp.

We have further optimized the space-efficient OR-set to make it a time-efficient one by implementing the set as a binary search tree (BST).

2.2 Specification

We now present a declarative framework for specifying MRDTs which closely follows the framework presented by Burckhardt et al. [2]. We define our specifications on *abstract states*, which capture the state of the distributed store. It consists of *events* in a 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,
- $oper : E \longrightarrow Op_\tau$ associates the data type operation with each event,
- $rval : E \longrightarrow Val_\tau$ associates the return value with each event,
- $time : E \longrightarrow Timestamp$ associates the timestamp at which an event was performed,
- $vis \subseteq E \times E$ is an irreflexive, asymmetric and transitive ***visibility relation***.

$e \xrightarrow{vis} f$ means e causally precedes 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 on 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 $\mathcal{F}_\tau(o, I) \in Val_\tau$.

2.2.1 OR-set specification.

For the OR-set, both add and remove operations always return $\bot$. We can formally specify the 'add-wins' conflict resolution strategy as follows:

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

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 $\mathcal{M}_{D_\tau} = (\Phi, \longrightarrow)$. 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 $b_\perp$, and is represented by $C_\perp = (\phi_\perp, \delta_\perp, 0)$ where $\phi_\perp = [b_\perp \mapsto \sigma_0]$ and $\delta_\perp = [b_\perp \mapsto I_0]$.

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:

$$\begin{aligned}
do^\#\langle I, e, op, a, t\rangle &= \langle I.E \cup \{e\}, I.oper[e \mapsto op], I.rval[e \mapsto a],\\
&\qquad I.time[e \mapsto t], I.vis \cup \{(f, e) \mid f \in I.E)\}\rangle\\
merge^\#(I_a, I_b) &= I_m\\
&\text{where } I_m.E = I_a.E \cup I_b.E\\
&I_m.\text{prop}(e) = \begin{cases} I_a(e) & \text{if } e \in I_a.E\\ I_b(e) & \text{if } e \in I_b.E\end{cases}\\
&I_m.vis = I_a.vis \cup I_b.vis\\
lca^\#(I_a, I_b) &= \langle E_l, I_a.oper\mid_{E_l}, I_a.rval\mid_{E_l}, I_a.time\mid_{E_l},\\
&\qquad I_a.vis\mid_{E_l}\rangle\\
&\text{where } E_l = I_a.E \cap I_b.E
\end{aligned}$$

Note that prop $\in \{oper, rval, time\}$. 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.

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

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

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

Figure 2. Semantics of the replicated datastore

Figure 2 describes the transition function $\longrightarrow$. The first rule describes the creation of new branch b_2 from the current branch b_1. Both the concrete and abstract states of the new 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 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.

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,

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

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 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 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 simulation relations used in [2], unlike [2], we apply this technique in the setting of mergeable replicated data types. Further, we also mechanize and automate simulation-based proofs by deriving simple sufficient conditions which can easily discharged by tools such as F*. Finally, we apply our proof technique on a wide range of MRDTs, with substantially complex specifications (e.g. queue MRDT).

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 on the branch. Verifying the correctness of a 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.

$$\begin{tikzcd}
I \arrow[r, "do^{\#}"] \arrow[d, leftrightarrow, "\mathcal{R}_{sim}"'] & I' \arrow[d, leftrightarrow, "\mathcal{R}_{sim}"] \\
\sigma \arrow[r, "\mathcal{D}_\tau.do"'] & \sigma'
\end{tikzcd}$$

Figure 3. Verifying operations

$$\begin{tikzcd}
\sigma_a \arrow[d, leftrightarrow, "\mathcal{R}_{sim}"'] \arrow[ddr] & I_{lca} \arrow[d, leftrightarrow, "\mathcal{R}_{sim}"] & \sigma_b \arrow[d, leftrightarrow, "\mathcal{R}_{sim}"] \arrow[ddl] \\
I_a \arrow[dddr] & \sigma_{lca} \arrow[d] & I_b \arrow[dddl] \\
& \mathcal{D}_\tau.merge(\sigma_{lca}, \sigma_a, \sigma_b) \arrow[d, leftrightarrow, "\mathcal{R}_{sim}"] & \\
& merge^{\#}(I_a, I_b) &
\end{tikzcd}$$

Figure 4. Verifying 3-way merge

This is depicted pictorially in figures 3 and 4. Figure 3 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.

Similarly, Figure 4 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. In order to discharge these conditions, we also consider two store properties, Ψ_{ts} and Ψ_{lca} that hold across all executions (shown in Table 1). These properties play an important role in discharging the conditions required for validity of the simulation relation.

$\Psi_{ts}(I)$ 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. Further, every event in the LCA will be visible to newly added events in either of the two branches. 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 3 and 4. 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 clearly 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 behavior.

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.*

We have also proved the soundness of our proof strategy.

4.2 Examples

Let us look at the simulation relations for verifying OR-set implementation in §2.1.1 against the specification $\mathcal{F}_{orset}$ in §2.2.1.

OR-set. Following is a candidate valid simulation relation for the OR-set:

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

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.

5 Peepul library in F*

In this section, we discuss the instantiation of the formalism developed thus far in Peepul, an F* library of certified efficient MRDTs. F*'s core is a functional programming language inspired by ML, with support for program verification 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 are able to directly utilise our MRDTs on top of Irmin [7], a Git-like distributed database whose execution model fits the MRDT system model.

Table 3 tabulates the results that correspond to the verification effort of building the Peepul library. The lines of code represents the number of lines of 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. For many of the proofs, F* is able to automatically verify the properties either without any

Table 1. Store properties

$\Psi_{ts}(I)$	$\forall e, e' \in I.E.\ e \xrightarrow{I.vis} e' \Rightarrow I.time(e) < I.time(e')$ $\land \forall e, e' \in I.E.\ I.time(e) = I.time(e') \Rightarrow e = e'$
$\Psi_{lca}(I_l, I_a, I_b)$	$I_l.vis = I_a.vis_{\mid I_l.E} = I_b.vis_{\mid I_l.E}$ $\land \forall e \in I_l.E.\ \forall e' \in (I_a.E \cup I_b.E) \setminus I_l.E.\ e \xrightarrow{I_a.vis \cup I_b.vis} e'$

Table 2. Sufficient conditions for showing validity of simulation relation

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

Table 3. Total lines of code and proofs for each MRDT and verification time in seconds.

MRDTs verified	**#Lines proof**	**#Lines code**	**Verif. Time (s)**
Increment-only counter	116	7	2.017
Enable-Wins flag	158	9	193.21
Last-Writer-Wins register	98	5	10.575
Grows-only set	67	9	0.481
Grows-only map	75	24	68.906
No tombstone OR-set	125	17	324.699
No tombstone OR-set (Space-efficient)	294	44	2108.456
No tombstone OR-set (BST)	280	38	2406.116
Functional queue	1123	32	5152.17

lemmas or 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 with an exception being OR-set and functional queues. As a whole, F* reduces manual effort and most of the proofs are checked within few seconds. We believe that some of the time consuming calls to the SMT solver may be profitably replaced by a few interactive proofs.

6 Conclusion

In this work, we present Peepul, a pragmatic approach to building and verifying MRDTs that retain the efficiency of sequential operations as well as merge. In order to capture the intent of the RDT, we use a declarative specification language to describe the sequential and replication semantics of RDT, which we use to prove the correctness of efficient implementations with the help of replication-aware simulations. We also introduce a new, notion of convergence modulo observable behavior, which allows replicas to converge to different states, as long as their observable behavior to clients remains the same. This notion allows us to build and verify even more efficient RDTs. We instantiate our technique as an F* library and mechanically verify the implementation of efficient purely functional implementations including an efficient replicated two-list queues. In the future, we plan to construct verified compound data types by composition of simpler data types through parametric polymorphism. We also plan to develop a methodology for the specification and verification of recursive MRDTs.

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] 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

[3] 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

[4] 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

[5] 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.

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

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

[8] 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

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

[10] 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

[11] 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

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

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

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

[15] 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.

[16] 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