Certified Mergeable Replicated Data Types

Vimala Soundarapandian
IIT Madras, India

KC Sivaramakrishnan
IIT Madras, India

Kartik Nagar
IIT Madras, India

Abstract

Constructing correct replicated data types is a challenging endeavour due to the complexity of reasoning about independently evolving states of the replicas. Mergeable Replicated Data Types (MRDTs) simplify the construction by allowing the merge of concurrent states to be expressed as a 3-way merge function between the two concurrent states and the state of the lowest common ancestor where the replicas diverged. While MRDTs bring in additional benefits of reasoning about intent preservation and composition, there is still an air-gap between the efficient implementations of MRDTs and their intended specifications.

In this work, we present principled approach to developing MRDTs that are *correct-by-construction* using F* solver-aided programming language. In our system, an MRDT is described as a triple of the state, operations that manipulate the state and the 3-way merge function. In addition, we expect proofs of algebraic properties of the 3-way merge functions – idempotence, associativity and commutativity. In return, the system ensures that MRDT is *convergent* (assuming only the algebraic properties) for arbitrary number of replicas and merges. Such certified MRDT implementations in F* are extracted to OCaml for use in Irmin, a distributed database built on the principles of Git.

1 Introduction

Mergeable Replicated Data Types (MRDTs) are inspired from the Git version control system. Similar to Git, an MRDT data store maintains a causal history of the states, and the states at different replicas may evolve on separate branches. However, unlike Git, the objects in the data store may be arbitrary data types equipped with a 3-way merge function to reconcile conflicting updates. When conflicting updates need to be reconciled, the causal history is used to determine the lowest common ancestor for use in the 3-way merge function along with the conflicting states. This branch-consistent view of replication not only makes it easier to develop individual data types, but also leads to a natural transactional semantics [Crooks et al. 2016; Dubey et al. 2020].

One of the challenges of replicated data types is that in a weakly consistent setting, it becomes tricky to reason about whether the implementation preserves the high-level intent. Indeed, individually correct data types may fail to preserve convergence when put together [Kleppmann 2020]. Kaki et al [Kaki et al. 2019] present a relational interpretation of replicated data types, where the characteristic properties of complex data types are captured as relations over its constituent elements. Then, the merge function devolves to a merge of these relations (sets) expressed as MRDTs.

While useful as a reasoning technique, mapping complex data types to sets does not lead to efficient implementations. For example, a queue in Kaki et al. is represented by two characteristic relations – a unary relation membership and a binary relation ordering. For a queue with n elements, the ordering relation contains $O(n^2)$ elements. Mapping the queue to its characteristic relations and back for every merge is inefficient.

The goal of the work is to provide a principled approach to construct certified MRDTs that are guaranteed to converge and preserve the high-level intent of the data type. Our primary contribution is an F* library for implementing MRDTs and verifying their convergence and intent preservation. The client of the library implements the MRDT as a state, the operations that transform the state, the 3-way merge function and proofs of associativity, commutativity and idempotence of merge. The library returns an OCaml implementation that satisfies convergence in the presence of many replicas. This OCaml implementation of MRDT is compatible with Irmin [Irmin 2021], a distributed database built on the principles of Git.

2 Merge function

As is the tradition, consider an increment-only counter MRDT which supports one operation: `Inc n` that increments the counter value by `n`. The state of the counter is the current counter value `type state = nat`, and the only operation is increment `type op = Inc of nat`. The operation is interpreted by the `apply_op` function:

```
let apply_op s (Inc n) = s + n
```

with the merge function

```
let merge lca a b = a + b - lca
```

where `lca` is the state of the lowest common ancestor of the two commits whose states `a` and `b` are being merged. Surprisingly, F* reports a type error in the merge function. Since the merge involves subtraction, without the *context*, F* is unable to prove that the result would be a natural number.

To overcome this problem, we associate a *history* with the merge function that captures the causal relationship between the three states. In particular, since the only operation allowed in the increment-only counter is increment, both `a` and `b` are greater than or equal to the `lca`. With this additional lemma, the merge function successfully type checks.

PaPoC '21, April 26, 2021, Edinburgh, UK

We can build more complex data types using the increment-only counter. For example, we can build a enables-wins flag, which is a replicated boolean flag with an enable and disable operation, and in the case of concurrent enable and disable, the flag is enabled in the merged state. The state of the enable-wins flag is:

```
type state = counter * bool
```

which maintains a pair of counter (which keeps track of enables) and the current state of the flag. The operations are:

```
type op = Enable | Disable
```

and their interpretations are:

```
let apply_op (c,_) o =
  match o with
  | Enable -> (increment c, true)
  | Disable -> (c, false)
```

The merge function in this case is:

```
let merge (lc,lf) (ac,af) (bc,bf) =
  (Counter.merge lc ac bc,
   (* if both flags are enabled *)
   if af && bf then true
   (* if both flags are disabled *)
   else if not af && not bf then false
   (* otherwise, check if there is
      a concurrent enable *)
   else if af then ac > lc
   else bc > lc)
```

Thanks to F* discharging the proof obligations to the SMT solver, the proof of convergence of enables-wins flag is automatically discharged, building upon the proof of convergence of the counter.

3 History

As we've seen in the case of increment-only counter, we need to associate the causal history of the execution to derive invariants such as monotonicity of the counter value to be used in the implementation of the merge functions and the proofs of its algebraic properties. To this end, we model history as a directed-acyclic graph (DAG) of commits, starting from an initial commit.

Each commit has a state, and is related to its descendants through a trace of operations defined on the datatype – the child state is obtained by applying the trace of operations to the parent state. The history also captures the branching and merging behaviour that is observed in a Git-like replicated store.

We also impose an additional requirement that any pair of commits in a well-formed history has a unique lowest common ancestor (LCA). This requirement mirrors the effect of recursive merges in Git in the case when a pair of commits have more than one LCA.

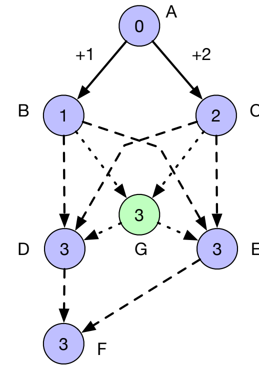

Figure 1. Recursive merge

A well-formed history of an increment-only counter is shown in Figure 1. The states of the replicas are represented by circles. The solid edge between the nodes represents the trace and the dashed edges represent the merge. In the merge of D and E, we see that there are two LCAs B and C. Using either one of them as the LCA for the merge would lead to an incorrect result. Hence, just like Git, we assume that there is a recursive merge commit G, which is the merge of the two LCAs B and C. Using G as the LCA gives us the correct merge result.

We use the notion of well-formed histories to identify sufficient conditions for convergence of a MRDT. In particular, our strategy is to show that if the merge operation of the MRDT satisfies simple algebraic properties such as commutativity, associativity and idempotence, then in any well-formed history, all replicas would eventually converge to the same state.

4 Final Remarks

Using the MRDT library we have proved the convergence properties of three MRDTs – increment-only counter, grow-only set, and enable-wins flag. In the future, we plan to verify more complex MRDTs like queues, graphs, ropes, etc., and build a library of certified MRDTs for Irmin.

References

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

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 - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings (Lecture Notes in Computer Science)*, Bruno C. d. S. Oliveira (Ed.), Vol. 12470. Springer, 231–250. https://doi.org/10.1007/978-3-030-64437-6_12

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

Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable replicated data types. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct 2019), 1–29. https://doi.org/10.1145/3360580

M. Kleppmann. 2020. Twitter thread. (Nov 2020). https://twitter.com/martinkl/status/1327020435419041792