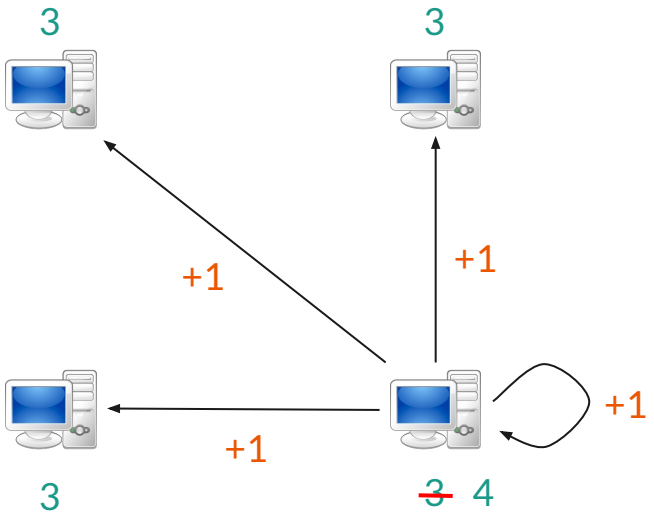
Composing and Decomposing Op-Based CRDTs with Semidirect Products

Matthew Weidner, Heather Miller, and Christopher Meiklejohn

Carnegie Mellon University
Pittsburgh, PA, USA

PaPoC Workshop 2020
Virtual Heraklion, Crete, Greece
27/4/2020

Recall: Operation-based Conflict-free Replicated Data Types (Op-based CRDTs)



- Eventually consistent replicated data types
- Messages describing operations are delivered to all replicas in causal order.
- Concurrent operations must commute.
  - Typically add metadata to states, messages
- Ex: counter (commutative)
- Ex: add-wins set (not commutative; need conflict resolution)
- When ops don't naturally commute, we have to think of a new design each time & prove it's correct.
- Thus it's **hard to design CRDTs for new data types.**

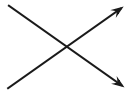
Motivating Example

Goal: Design an integer register CRDT supporting add and mult operations.

1

- add operations alone are easy: they already commute.
- Same for mult operations alone.
- But they don't commute with each other:

A 1



B 1

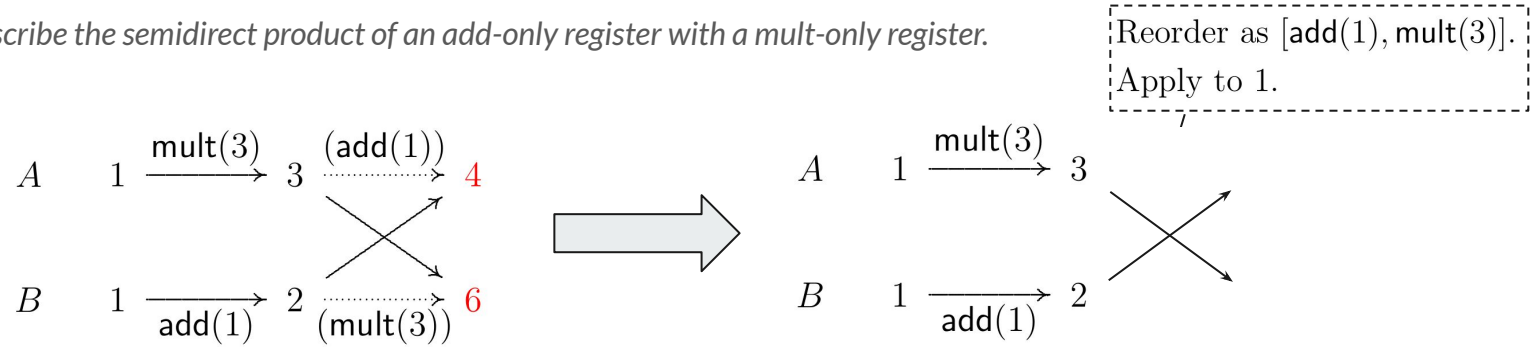
So we need some form of conflict resolution that handles concurrent add and mult operations.

Highlights

- We present a general CRDT construction technique, the semidirect product, which gives a CRDT for this example and many others.
- It combines the operations of two op-based CRDTs that have the same state space, resolving conflicts between their (non-commuting) operations in a uniform way.
  - Ex. Add-only set + remove-only set → set CRDT
  - Special case: adding operations to an existing CRDT
- We use it to construct both old and new CRDTs.
  - Old ex: add observed-reset operation to a CRDT (used to remove values in Riak map)
  - New ex: integer register supporting add and mult operations ← this talk

Goal: Integer register CRDT supporting add and mult operations

We will describe the semidirect product of an add-only register with a mult-only register.

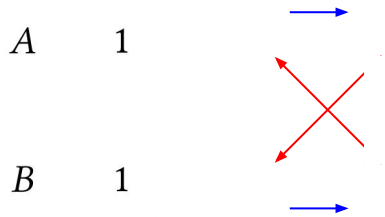


What to do when two replicas concurrently issue add and mult operations?

Attempt 1. Dictate that adds are always reordered to be before concurrent mults.

Goal: Integer register CRDT supporting add and mult operations

Error: Not well-defined in general (might not be an ordering respecting this & the causal order).

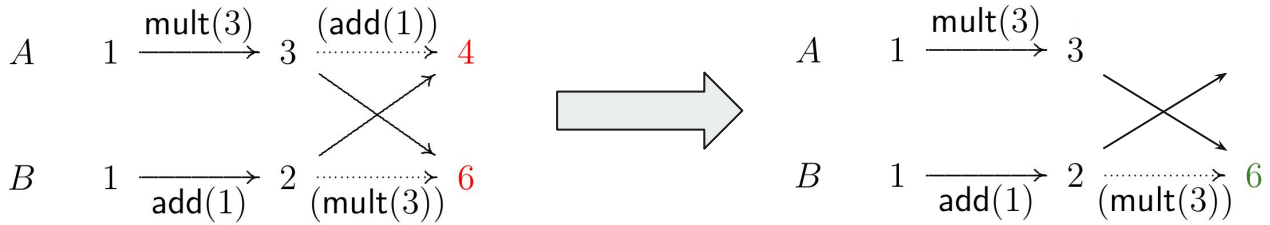


Adds come before concurrent mults

Causally ordered operations stay in order

No valid order: makes a cycle.

Goal: Integer register CRDT supporting add and mult operations

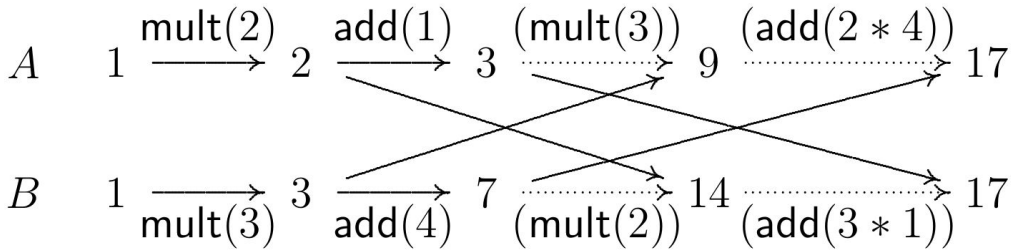


What to do when two replicas concurrently issue add and mult operations?

Attempt 2. Observation: If we receive concurrent add and mult operations in the "wrong" order (mult before add), we get the right result if we *transform* the add by the mult.

- Works by distributive law: $3 * (1 + 1) = 3 * 1 + (3 * 1)$

Goal: Integer register CRDT supporting add and mult operations



In general, when you receive an add operation, transform it by all concurrent mult operations you've already received.

- So $\mathsf{add}(m)$ becomes $\mathsf{add}(n_1 * \cdots * n_k * m)$. $\mathsf{mult}(n_1), \ldots, \mathsf{mult}(n_k)$ operations you've previously received that are concurrent to $\mathsf{add}(m)$.
- Formally, track concurrency by attaching vector clocks to each operation. Store mult operations (with their vector clocks) in a history set, as part of the CRDT state.

Old Example: Observed-Reset CRDTs

In the Riak map CRDT, when a key is removed, the map applies an *observed-reset* operation to its value, restoring it to its initial state.

- Only affects “observed” (causally prior) ops; concurrent ops are applied to the reset state.
- Ignoring the map, we consider the CRDT-with-reset-op as a new CRDT.

We decompose the CRDT-with-reset-op as a semidirect product of two simpler CRDTs: the original (non-resettable) CRDT, and a CRDT with reset operations only.

To do this, we have to choose which operations should “go first” in case of concurrency (like how adds go before mults). We choose the resets. Thus resets don’t affect concurrent operations.

Having made that choice, the semidirect product does much of the design work.

Further Examples

New CRDTs (Composition)	Old CRDTs (Decomposition)
• Add/mult integer register • Commutative semirings • Sequence w/ reverse op • Sequence w/ range remove op • Map* with higher-order map ops (apply an op to all values) *excluding removes for now	• Observed-reset CRDTs • Reset-wins CRDTs • Flags (enable wins & disable wins) • Sets (add wins & remove wins)

Thanks for watching :)

Full construction and more in the arXiv paper (proofs, optimizations, generality, name origin…).

Questions?

Relation to Operational Transformation

Recall: OT is a competing approach to eventually consistent replicated data types. Instead of making concurrent operations commute, you define a *transformation function* tf_1: O x O -> O.

- Received operations are transformed against previously-received concurrent operations using the (somewhat complicated) adOPTed algorithm (Ressel et al. ‘96).
- Eventual consistency follows from 2 *Transformation Properties*, which are hard to verify (esp. TP2).

The semidirect product is the special case of OT when O = O_1 \sqcup O_2 and tf_1(p, q) = p except when p \in O_1, q \in O_2. (O_2 ops transform concurrent O_1 ops, but that’s it.)

- With these restrictions, adOPTed becomes the simple algorithm described for the add/mult integer register, and TP2 is typically trivial to verify.

It is interesting that we get common CRDT semantics from this special case of OT.

Generality

Q: When can a CRDT be decomposed as a semidirect product of simpler CRDTs?

Let C be a CRDT in the POLog model of pure op-based CRDTs. I.e., C is defined by a function *f* mapping the partially ordered log of operations to a visible state.

Suppose the ops of C can be partitioned as $O_1 \cup O_2$ so that for any POLog L, $f(L) = f(L')$ whenever $L = L'$ except that we have added some edges $o_1 < o_2$ with $o_1 \in O_1$, $o_2 \in O_2$. Then C is the semidirect product of a CRDT with only O_1 ops and a CRDT with only O_2 ops.

This expresses the rule that O_1 ops "go before" concurrent O_2 ops.

Name Origin: Semidirect Product of Groups

In abstract algebra, a *group* is a set G with a binary operation $\circ : G \times G \to G$ satisfying multiplication-like properties.

- $\circ$ is associative $(g \circ (h \circ k) = (g \circ h) \circ k)$; $\exists\ 1 \in G$ s.t. $g \circ 1 = 1 \circ g = g$; $\forall g \in G\ \exists\ g^{-1} \in G$ s.t. $g \circ g^{-1} = 1$.
- I.e., a monoid with inverses.

The *semidirect product* combines two groups G, H into a group with set G x H. It contains copies of G and H which commute as

$$h \circ g = (h \triangleright g) \circ h$$

for a suitable *action* $\triangleright$ of H on G.

The semidirect product of CRDTs uses an analogous rule to reorder concurrent ops from the components.