Verifying Strong Eventual Consistency in δ-CRDTs

Taylor Blau

University of Washington

June, 2020

Contributions

This thesis:

- Mechanized proofs in Isabelle that two δ-state CRDTs inhabit SEC.
  - Reuse a library for verifying operation-based CRDTs of Victor Gomes of Cambridge to reason about δ-state CRDTs.
  - Weaken the network model of Gomes' to support duplicated messages.
- Two reductions that allow us to reason about δ-state CRDTs in terms of operation-based CRDTs.
- Two encodings of the latter reduction.

Contributions

This thesis:

- Mechanized proofs in Isabelle that two δ-state CRDTs inhabit SEC.
  - Reuse a library for verifying operation-based CRDTs of Victor Gomes of Cambridge to reason about δ-state CRDTs.
  - Weaken the network model of Gomes' to support duplicated messages.
- Two reductions that allow us to reason about δ-state CRDTs in terms of operation-based CRDTs.
- Two encodings of the latter reduction.

Contributions

This thesis:

- Mechanized proofs in Isabelle that two δ-state CRDTs inhabit SEC.
  - Reuse a library for verifying operation-based CRDTs of Victor Gomes of Cambridge to reason about δ-state CRDTs.
  - Weaken the network model of Gomes' to support duplicated messages.
- Two reductions that allow us to reason about δ-state CRDTs in terms of operation-based CRDTs.
- Two encodings of the latter reduction.

Contributions

This thesis:

- Mechanized proofs in Isabelle that two δ-state CRDTs inhabit SEC.
  - Reuse a library for verifying operation-based CRDTs of Victor Gomes of Cambridge to reason about δ-state CRDTs.
  - Weaken the network model of Gomes' to support duplicated messages.
- Two reductions that allow us to reason about δ-state CRDTs in terms of operation-based CRDTs.
- Two encodings of the latter reduction.

Contributions

This thesis:

- Mechanized proofs in Isabelle that two δ-state CRDTs inhabit SEC.
  - Reuse a library for verifying operation-based CRDTs of Victor Gomes of Cambridge to reason about δ-state CRDTs.
  - Weaken the network model of Gomes' to support duplicated messages.
- Two reductions that allow us to reason about δ-state CRDTs in terms of operation-based CRDTs.
- Two encodings of the latter reduction.

This talk

- Why distributed systems?
- Consistency models: classic approaches and relaxed approximations.
- CRDTs: operation-, state- and δ-state based, and the trade-offs each makes.
- Reductions between CRDT variants.
- Mechanized proofs in two encodings.
- Conclusion.

This talk

- Why distributed systems?
- Consistency models: classic approaches and relaxed approximations.
- CRDTs: operation-, state- and δ-state based, and the trade-offs each makes.
- Reductions between CRDT variants.
- Mechanized proofs in two encodings.
- Conclusion.

This talk

- Why distributed systems?
- Consistency models: classic approaches and relaxed approximations.
- CRDTs: operation-, state- and δ-state based, and the trade-offs each makes.
- Reductions between CRDT variants.
- Mechanized proofs in two encodings.
- Conclusion.

This talk

- Why distributed systems?
- Consistency models: classic approaches and relaxed approximations.
- CRDTs: operation-, state- and δ-state based, and the trade-offs each makes.
- Reductions between CRDT variants.
- Mechanized proofs in two encodings.
- Conclusion.

This talk

- Why distributed systems?
- Consistency models: classic approaches and relaxed approximations.
- CRDTs: operation-, state- and δ-state based, and the trade-offs each makes.
- Reductions between CRDT variants.
- Mechanized proofs in two encodings.
- Conclusion.

This talk

- Why distributed systems?
- Consistency models: classic approaches and relaxed approximations.
- CRDTs: operation-, state- and δ-state based, and the trade-offs each makes.
- Reductions between CRDT variants.
- Mechanized proofs in two encodings.
- Conclusion.

Distributed Systems

Why distributed systems?

1. *Resiliency*. Tolerates failure of any one (or more) participants.
2. *Scalability*. Meeting the demands of an increased workload as simple as adding more hardware.
3. *Locality*. Service requests to varied locations by placing hardware close to where requests originate.

Distributed Systems

Why distributed systems?

1. *Resiliency*. Tolerates failure of any one (or more) participants.
2. *Scalability*. Meeting the demands of an increased workload as simple as adding more hardware.
3. *Locality*. Service requests to varied locations by placing hardware close to where requests originate.

Distributed Systems

Why distributed systems?

1. *Resiliency*. Tolerates failure of any one (or more) participants.
2. *Scalability*. Meeting the demands of an increased workload as simple as adding more hardware.
3. *Locality*. Service requests to varied locations by placing hardware close to where requests originate.

Distributed Consensus Algorithms

Definition (Distributed Consensus Algorithm, Howard and Mortier [2020])

An algorithm is said to solve distributed consensus if it has the following three safety requirements:

1. *Non-triviality*: The decided value must have been proposed by a participant.
2. *Safety*: Once a value has been decided, no other value will be decided.
3. *Safe learning*: If a participant learns a value, it must learn the decided value.

In addition, it must satisfy the following two progress requirements:

1. *Progress*: Under previously agreed-upon liveness conditions, if a value is proposed by a participant, then a value is eventually decided.
2. *Eventual learning*: Under the same conditions as above, if a value is decided, then that value must be eventually learned.

Distributed Consensus Algorithms

Definition (Distributed Consensus Algorithm, Howard and Mortier [2020])

An algorithm is said to solve distributed consensus if it has the following three safety requirements:

1. *Non-triviality*: The decided value must have been proposed by a participant.
2. *Safety*: Once a value has been decided, no other value will be decided.
3. *Safe learning*: If a participant learns a value, it must learn the decided value.

In addition, it must satisfy the following two progress requirements:

1. *Progress*: Under previously agreed-upon liveness conditions, if a value is proposed by a participant, then a value is eventually decided.
2. *Eventual learning*: Under the same conditions as above, if a value is decided, then that value must be eventually learned.

Distributed Consensus Algorithms

Definition (Distributed Consensus Algorithm, Howard and Mortier [2020])

An algorithm is said to solve distributed consensus if it has the following three safety requirements:

1. *Non-triviality*: The decided value must have been proposed by a participant.
2. *Safety*: Once a value has been decided, no other value will be decided.
3. *Safe learning*: If a participant learns a value, it must learn the decided value.

In addition, it must satisfy the following two progress requirements:

1. *Progress*: Under previously agreed-upon liveness conditions, if a value is proposed by a participant, then a value is eventually decided.
2. *Eventual learning*: Under the same conditions as above, if a value is decided, then that value must be eventually learned.

Distributed Consensus Algorithms

Definition (Distributed Consensus Algorithm, Howard and Mortier [2020])

An algorithm is said to solve distributed consensus if it has the following three safety requirements:

1. *Non-triviality*: The decided value must have been proposed by a participant.
2. *Safety*: Once a value has been decided, no other value will be decided.
3. *Safe learning*: If a participant learns a value, it must learn the decided value.

In addition, it must satisfy the following two progress requirements:

1. *Progress*: Under previously agreed-upon liveness conditions, if a value is proposed by a participant, then a value is eventually decided.
2. *Eventual learning*: Under the same conditions as above, if a value is decided, then that value must be eventually learned.

Distributed Consensus Algorithms

Definition (Distributed Consensus Algorithm, Howard and Mortier [2020])

An algorithm is said to solve distributed consensus if it has the following three safety requirements:

1. *Non-triviality*: The decided value must have been proposed by a participant.
2. *Safety*: Once a value has been decided, no other value will be decided.
3. *Safe learning*: If a participant learns a value, it must learn the decided value.

In addition, it must satisfy the following two progress requirements:

1. *Progress*: Under previously agreed-upon liveness conditions, if a value is proposed by a participant, then a value is eventually decided.
2. *Eventual learning*: Under the same conditions as above, if a value is decided, then that value must be eventually learned.

Distributed Consensus Algorithms

Definition (Distributed Consensus Algorithm, Howard and Mortier [2020])

An algorithm is said to solve distributed consensus if it has the following three safety requirements:

1. *Non-triviality*: The decided value must have been proposed by a participant.
2. *Safety*: Once a value has been decided, no other value will be decided.
3. *Safe learning*: If a participant learns a value, it must learn the decided value.

In addition, it must satisfy the following two progress requirements:

1. *Progress*: Under previously agreed-upon liveness conditions, if a value is proposed by a participant, then a value is eventually decided.
2. *Eventual learning*: Under the same conditions as above, if a value is decided, then that value must be eventually learned.

Distributed Consensus Algorithms

Two of the most popular algorithms in this field:

- Paxos [Lamport, 1998]
- Raft [Ongaro and Ousterhout, 2014]

...are notoriously difficult to implement in practice [Howard and Mortier, 2020].

- Often the subject of advanced undergraduate-level courses in Distributed Systems (CSE 452).
- Subject of much mechanized verification effort [Wilcox et al., 2015, Woos et al., 2016].

Distributed Consensus Algorithms

Two of the most popular algorithms in this field:

- Paxos [Lamport, 1998]
- Raft [Ongaro and Ousterhout, 2014]

...are notoriously difficult to implement in practice [Howard and Mortier, 2020].

- Often the subject of advanced undergraduate-level courses in Distributed Systems (CSE 452).
- Subject of much mechanized verification effort [Wilcox et al., 2015, Woos et al., 2016].

Distributed Consensus Algorithms

Two of the most popular algorithms in this field:

- Paxos [Lamport, 1998]
- Raft [Ongaro and Ousterhout, 2014]

...are notoriously difficult to implement in practice [Howard and Mortier, 2020].

- Often the subject of advanced undergraduate-level courses in Distributed Systems (CSE 452).
- Subject of much mechanized verification effort [Wilcox et al., 2015, Woos et al., 2016].

Distributed Consensus Algorithms

Two of the most popular algorithms in this field:

- Paxos [Lamport, 1998]
- Raft [Ongaro and Ousterhout, 2014]

...are notoriously difficult to implement in practice [Howard and Mortier, 2020].

- Often the subject of advanced undergraduate-level courses in Distributed Systems (CSE 452).
- Subject of much mechanized verification effort [Wilcox et al., 2015, Woos et al., 2016].

Distributed Consensus Algorithms

Why? ...one possible answer: *safety*.

1. Coordinating a shared value between multiple replicas is difficult.
2. Unreliable networks make this task even more difficult.
3. Ensuring that all nodes learn the same value makes this even more difficult still.

Distributed Consensus Algorithms

Why? ...one possible answer: *safety*.

1. Coordinating a shared value between multiple replicas is difficult.
2. Unreliable networks make this task even more difficult.
3. Ensuring that all nodes learn the same value makes this even more difficult still.

Distributed Consensus Algorithms

Why? ...one possible answer: *safety*.

1. Coordinating a shared value between multiple replicas is difficult.
2. Unreliable networks make this task even more difficult.
3. Ensuring that all nodes learn the same value makes this even more difficult still.

Distributed Consensus Algorithms

Why? ...one possible answer: *safety*.

1. Coordinating a shared value between multiple replicas is difficult.
2. Unreliable networks make this task even more difficult.
3. Ensuring that all nodes learn the same value makes this even more difficult still.

Eventual Consistency

Eventual consistency captures the informal notion that if all clients stop submitting updates to the system, all replicas in the system eventually reach the same value.

More formally:

Definition (Eventual Consistency [Shapiro et al., 2011])

1. *Eventual delivery.* An update delivered at some correct replica is eventually delivered at all replicas.

$$\forall r_1, r_2.\, f \in (\text{delivered } r_1) \Rightarrow \Diamond f \in (\text{delivered } r_2)$$

2. *Convergence.* Correct replicas which have received the same *set* of updates eventually reflect the same state.

$$\forall r_1, r_2.\ \Box\ (\text{delivered } r_1) = (\text{delivered } r_2) \Rightarrow \Diamond\ \Box\ q(r_1) = q(r_2)$$

3. *Termination.* All method executions terminate.

Eventual Consistency

Eventual consistency captures the informal notion that if all clients stop submitting updates to the system, all replicas in the system eventually reach the same value.
More formally:

Definition (Eventual Consistency [Shapiro et al., 2011])

1. *Eventual delivery.* An update delivered at some correct replica is eventually delivered at all replicas.

$$\forall r_1, r_2.\, f \in (\text{delivered } r_1) \Rightarrow \Diamond f \in (\text{delivered } r_2)$$

2. *Convergence.* Correct replicas which have received the same *set* of updates eventually reflect the same state.

$$\forall r_1, r_2.\ \Box\ (\text{delivered } r_1) = (\text{delivered } r_2) \Rightarrow \Diamond\ \Box\ q(r_1) = q(r_2)$$

3. *Termination.* All method executions terminate.

Eventual Consistency

Eventual consistency captures the informal notion that if all clients stop submitting updates to the system, all replicas in the system eventually reach the same value.
More formally:

Definition (Eventual Consistency [Shapiro et al., 2011])

1. *Eventual delivery*. An update delivered at some correct replica is eventually delivered at all replicas.

$$\forall r_1, r_2.\, f \in (\text{delivered } r_1) \Rightarrow \Diamond f \in (\text{delivered } r_2)$$

2. *Convergence*. Correct replicas which have received the same *set* of updates eventually reflect the same state.

$$\forall r_1, r_2.\ \Box\, (\text{delivered } r_1) = (\text{delivered } r_2) \Rightarrow \Diamond\, \Box\, q(r_1) = q(r_2)$$

3. *Termination*. All method executions terminate.

Eventual Consistency

Eventual consistency captures the informal notion that if all clients stop submitting updates to the system, all replicas in the system eventually reach the same value.
More formally:

Definition (Eventual Consistency [Shapiro et al., 2011])

1. *Eventual delivery*. An update delivered at some correct replica is eventually delivered at all replicas.

$$\forall r_1, r_2.\, f \in (\text{delivered } r_1) \Rightarrow \Diamond f \in (\text{delivered } r_2)$$

2. *Convergence*. Correct replicas which have received the same *set* of updates eventually reflect the same state.

$$\forall r_1, r_2.\ \Box\ (\text{delivered } r_1) = (\text{delivered } r_2) \Rightarrow \Diamond\ \Box\ q(r_1) = q(r_2)$$

3. *Termination*. All method executions terminate.

Shortcomings of Eventual Consistency

EC is a relatively weak form of consistency:

1. EC systems will sometimes execute an update immediately only to discover that it produces a conflict with some future update, and so frequent roll-backs may be performed [Shapiro et al., 2011].
2. EC is merely a liveness guarantee. It does not impose any restriction on nodes which have received the same set or even sequence of messages.

Shortcomings of Eventual Consistency

EC is a relatively weak form of consistency:

1. EC systems will sometimes execute an update immediately only to discover that it produces a conflict with some future update, and so frequent roll-backs may be performed [Shapiro et al., 2011].
2. EC is merely a liveness guarantee. It does not impose any restriction on nodes which have received the same set or even sequence of messages.

Strong Eventual Consistency

Definition (Strong Eventual Consistency [Shapiro et al., 2011])

1. The system is EC, as previously described.
2. *Strong convergence.* Any pair of replicas which have received the same set of messages must return the same value when queried immediately.

$$\forall r_1, r_2. (\text{delivered } r_1) = (\text{delivered } r2) \Rightarrow q(r_1) = q(r_2)$$

Strong Eventual Consistency

Definition (Strong Eventual Consistency [Shapiro et al., 2011])

1. The system is EC, as previously described.
2. *Strong convergence*. Any pair of replicas which have received the same set of messages must return the same value when queried immediately.

$$\forall r_1, r_2. (\text{delivered } r_1) = (\text{delivered } r2) \Rightarrow q(r_1) = q(r_2)$$

Strong Eventual Consistency

Why is SEC an appealing model?

- No requirements on replicas which have not received the same sequence/set of updates.
- Trade linearizability for the ability to let replicas drift.
- Allow replicas which haven't yet received all updates to return an earlier value of the computation.

Practical (in certain applications): offline synchronization (iOS Notes), Facebook "like" counters, Cassandra, etc.

Strong Eventual Consistency

Why is SEC an appealing model?

- No requirements on replicas which have not received the same sequence/set of updates.
- Trade linearizability for the ability to let replicas drift.
- Allow replicas which haven't yet received all updates to return an earlier value of the computation.

Practical (in certain applications): offline synchronization (iOS Notes), Facebook "like" counters, Cassandra, etc.

Strong Eventual Consistency

Why is SEC an appealing model?

- No requirements on replicas which have not received the same sequence/set of updates.
- Trade linearizability for the ability to let replicas drift.
- Allow replicas which haven't yet received all updates to return an earlier value of the computation.

Practical (in certain applications): offline synchronization (iOS Notes), Facebook "like" counters, Cassandra, etc.

Strong Eventual Consistency

Why is SEC an appealing model?

- No requirements on replicas which have not received the same sequence/set of updates.
- Trade linearizability for the ability to let replicas drift.
- Allow replicas which haven't yet received all updates to return an earlier value of the computation.

Practical (in certain applications): offline synchronization (iOS Notes), Facebook "like" counters, Cassandra, etc.

Strong Eventual Consistency

Why is SEC an appealing model?

- No requirements on replicas which have not received the same sequence/set of updates.
- Trade linearizability for the ability to let replicas drift.
- Allow replicas which haven't yet received all updates to return an earlier value of the computation.

Practical (in certain applications): offline synchronization (iOS Notes), Facebook "like" counters, Cassandra, etc.

Conflict-free Replicated Datatypes

CRDTs are a class of replicated datatypes which implement SEC Shapiro et al. [2011]. There exist two broad classes:

1. State-based CRDTs. States form a join lattice, progress is made by sharing states with other replicas and merging with local state.
2. Operation-based CRDTs. Operations are serialized and delivered to all replicas in order.

Conflict-free Replicated Datatypes

CRDTs are a class of replicated datatypes which implement SEC Shapiro et al. [2011]. There exist two broad classes:

1. State-based CRDTs. States form a join lattice, progress is made by sharing states with other replicas and merging with local state.
2. Operation-based CRDTs. Operations are serialized and delivered to all replicas in order.

State-based CRDTs

A state-based CRDT is a 5-tuple (S, s^0, q, u, m):

1. Individual CRDT replicas each have some state $s^i \in S$ for $i \geq 0$, and is initially s^0.
2. The value may be queried by any client or other replica by invoking q.
3. It may be updated with u, which has a unique type per CRDT object.
4. Finally, m merges the state of some other remote replica.

Example state-based CRDT

Grow-only counter: increments a (grow-only) shared value over time, supports queries of the last-known value.

$$\text{G-Counter}_s = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s.\ \sum_{i \in \mathcal{I}} s(i) \\ u : \lambda s, i.\ s\,\{i \mapsto s(i) + 1\} \\ m : \lambda s_1, s_2.\ [\max\{s_1(i), s_2(i)\} : i \in \mathrm{dom}(s_1) \cup \mathrm{dom}(s_2)] \end{cases}$$

state-based properties

1. Crucially, the states of a given state-based CRDT form a partially-ordered set $\langle S, \sqsubseteq \rangle$. This poset is used to form a join semi-lattice, where any finite subset of elements has a natural least upper-bound.
2. For every state-based CRDT whose states S form some join semi-lattice (with join operator $\sqcup$), we assume that:

$$m(s_1, s_2) = s_1 \sqcup s_2$$

state-based properties

1. Crucially, the states of a given state-based CRDT form a partially-ordered set $\langle S, \sqsubseteq \rangle$. This poset is used to form a join semi-lattice, where any finite subset of elements has a natural least upper-bound.
2. For every state-based CRDT whose states S form some join semi-lattice (with join operator $\sqcup$), we assume that:
$$m(s_1, s_2) = s_1 \sqcup s_2$$

state-based properties of $\sqcup$

$\sqcup$ must satisfy three mathematical identities:

- The operator is *commutative*, i.e., that $s_1 \sqcup s_2 = s_2 \sqcup s_1$, or that order does not matter.
- The operator is *idempotent*, i.e., that $(s_1 \sqcup s_2) \sqcup s_2 = s_1 \sqcup s_2$, or that repeated updates reach a fixed point.
- Finally, the operator is *associative*, i.e., that $s_1 \sqcup (s_2 \sqcup s_3) = (s_1 \sqcup s_2) \sqcup s_3$, or that grouping of arguments does not matter.

state-based properties of $\sqcup$

$\sqcup$ must satisfy three mathematical identities:

- The operator is *commutative*, i.e., that $s_1 \sqcup s_2 = s_2 \sqcup s_1$, or that order does not matter.
- The operator is *idempotent*, i.e., that $(s_1 \sqcup s_2) \sqcup s_2 = s_1 \sqcup s_2$, or that repeated updates reach a fixed point.
- Finally, the operator is *associative*, i.e., that $s_1 \sqcup (s_2 \sqcup s_3) = (s_1 \sqcup s_2) \sqcup s_3$, or that grouping of arguments does not matter.

state-based properties of $\sqcup$

$\sqcup$ must satisfy three mathematical identities:

- The operator is *commutative*, i.e., that $s_1 \sqcup s_2 = s_2 \sqcup s_1$, or that order does not matter.
- The operator is *idempotent*, i.e., that $(s_1 \sqcup s_2) \sqcup s_2 = s_1 \sqcup s_2$, or that repeated updates reach a fixed point.
- Finally, the operator is *associative*, i.e., that $s_1 \sqcup (s_2 \sqcup s_3) = (s_1 \sqcup s_2) \sqcup s_3$, or that grouping of arguments does not matter.

state-based properties of $\sqcup$

$\sqcup$ must satisfy three mathematical identities:

- The operator is *commutative*, i.e., that $s_1 \sqcup s_2 = s_2 \sqcup s_1$, or that order does not matter.
- The operator is *idempotent*, i.e., that $(s_1 \sqcup s_2) \sqcup s_2 = s_1 \sqcup s_2$, or that repeated updates reach a fixed point.
- Finally, the operator is *associative*, i.e., that $s_1 \sqcup (s_2 \sqcup s_3) = (s_1 \sqcup s_2) \sqcup s_3$, or that grouping of arguments does not matter.

state-based properties of $\sqcup$

...why place these restrictions on $\sqcup$? Because:

- Commutativity means that updates can be delivered from other replicas in any order.
- Idempotency means that updates can be delivered any number of times without changing the effect.
- Associativity means that updates can be applied in any grouping (useful for causality-preserving CRDTs, but not studied further here).

state-based properties of $\sqcup$

...why place these restrictions on $\sqcup$? Because:

- Commutativity means that updates can be delivered from other replicas in any order.
- Idempotency means that updates can be delivered any number of times without changing the effect.
- Associativity means that updates can be applied in any grouping (useful for causality-preserving CRDTs, but not studied further here).

state-based properties of $\sqcup$

...why place these restrictions on $\sqcup$? Because:

- Commutativity means that updates can be delivered from other replicas in any order.
- Idempotency means that updates can be delivered any number of times without changing the effect.
- Associativity means that updates can be applied in any grouping (useful for causality-preserving CRDTs, but not studied further here).

state-based properties of $\sqcup$

...why place these restrictions on $\sqcup$? Because:

- Commutativity means that updates can be delivered from other replicas in any order.
- Idempotency means that updates can be delivered any number of times without changing the effect.
- Associativity means that updates can be applied in any grouping (useful for causality-preserving CRDTs, but not studied further here).

Example state-based CRDT

Grow-only counter: increments a (grow-only) shared value over time, supports queries of the last-known value.

$$\text{G-Counter}_s = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} & \text{Each element in the lattice a vector of naturals.} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s. \sum_{i \in \mathcal{I}} s(i) \\ u : \lambda s, i. \, s\,\{i \mapsto s(i) + 1\} \\ m : \lambda s_1, s_2. \, [\max\{s_1(i), s_2(i)\} : i \in \mathrm{dom}(s_1) \cup \mathrm{dom}(s_2)] \\ \quad \text{Least upper bound } \sqcup \text{ defined by the element-wise maximum.} \end{cases}$$

Example state-based CRDT

Grow-only counter: increments a (grow-only) shared value over time, supports queries of the last-known value.

$$\text{G-Counter}_s = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} & \text{Each element in the lattice a vector of naturals.} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s. \sum_{i \in \mathcal{I}} s(i) \\ u : \lambda s, i. \, s\{i \mapsto s(i) + 1\} \\ m : \lambda s_1, s_2. \, [\max\{s_1(i), s_2(i)\} : i \in \mathrm{dom}(s_1) \cup \mathrm{dom}(s_2)] \\ \quad \text{Least upper bound } \sqcup \text{ defined by the element-wise maximum.} \end{cases}$$

Example state-based CRDT

Grow-only counter: increments a (grow-only) shared value over time, supports queries of the last-known value.

$$\text{G-Counter}_s = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} & \text{Each element in the lattice a vector of naturals.} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s.\ \sum_{i \in \mathcal{I}} s(i) \\ u : \lambda s, i.\ s\,\{i \mapsto s(i) + 1\} \\ m : \lambda s_1, s_2.\ [\max\{s_1(i), s_2(i)\} : i \in \mathrm{dom}(s_1) \cup \mathrm{dom}(s_2)] \\ \quad \text{Least upper bound } \sqcup \text{ defined by the element-wise maximum.} \end{cases}$$

Example state-based CRDT

Grow-only set: replicated monotonic (supports $\cup$, but not $\setminus$) set, query q defines a unary relation over items in the set.

$$\text{G-Set}_s(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) & \text{Each element in the latice is some subset of } \mathcal{X}. \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ u : \lambda x.\, s \cup \{x\} & \text{The set is updated by replacing the current set with the union.} \\ m : \lambda s_1, s_2.\, s_1 \cup s_2 & \text{The union of sets defines a least-upper bound in the lattice.} \end{cases}$$

The lattice-of-sets (for some family of items $\mathcal{X}$) is $\langle \mathcal{P}(\mathcal{X}), \subseteq \rangle$, and the least-upper bound is defined by $\cup$.

Example state-based CRDT

Grow-only set: replicated monotonic (supports $\cup$, but not $\setminus$) set, query q defines a unary relation over items in the set.

$$\text{G-Set}_s(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) & \text{Each element in the latice is some subset of } \mathcal{X}. \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ u : \lambda x.\, s \cup \{x\} & \text{The set is updated by replacing the current set with the union.} \\ m : \lambda s_1, s_2.\, s_1 \cup s_2 & \text{The union of sets defines a least-upper bound in the lattice.} \end{cases}$$

The lattice-of-sets (for some family of items $\mathcal{X}$) is $\langle \mathcal{P}(\mathcal{X}), \subseteq \rangle$, and the least-upper bound is defined by $\cup$.

Example state-based CRDT

Grow-only set: replicated monotonic (supports $\cup$, but not $\setminus$) set, query q defines a unary relation over items in the set.

$$\text{G-Set}_s(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) & \text{Each element in the latice is some subset of } \mathcal{X}. \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ u : \lambda x.\, s \cup \{x\} & \text{The set is updated by replacing the current set with the union.} \\ m : \lambda s_1, s_2.\, s_1 \cup s_2 & \text{The union of sets defines a least-upper bound in the lattice.} \end{cases}$$

The lattice-of-sets (for some family of items $\mathcal{X}$) is $\langle \mathcal{P}(\mathcal{X}), \subseteq \rangle$, and the least-upper bound is defined by $\cup$.

op-based CRDTs

An op-based CRDT is a 6-tuple (S, s^0, q, t, u, P).

- S, s^0, and q retain the same meaning as for the state-based CRDTs.
- S need not necessairly form a semi-lattice.
- Operations are communicated instead of state. To deliver an operation:
  1. The *prepare-update* implementation t is applied at the locally to prepare a representation of the operation.
  2. The *effect-update* implementation u is applied at the local and remote replicas if and only if the delivery precondition P is met, causing the desired update to take effect.

op-based CRDTs

An op-based CRDT is a 6-tuple (S, s^0, q, t, u, P).

- S, s^0, and q retain the same meaning as for the state-based CRDTs.
- S need not necessairly form a semi-lattice.
- Operations are communicated instead of state. To deliver an operation:
  1. The *prepare-update* implementation t is applied at the locally to prepare a representation of the operation.
  2. The *effect-update* implementation u is applied at the local and remote replicas if and only if the delivery precondition P is met, causing the desired update to take effect.

op-based CRDTs

An op-based CRDT is a 6-tuple (S, s^0, q, t, u, P).

- S, s^0, and q retain the same meaning as for the state-based CRDTs.
- S need not necessairly form a semi-lattice.
- Operations are communicated instead of state. To deliver an operation:
  1. The *prepare-update* implementation t is applied at the locally to prepare a representation of the operation.
  2. The *effect-update* implementation u is applied at the local and remote replicas if and only if the delivery precondition P is met, causing the desired update to take effect.

op-based CRDTs

An op-based CRDT is a 6-tuple (S, s^0, q, t, u, P).

- S, s^0, and q retain the same meaning as for the state-based CRDTs.
- S need not necessarily form a semi-lattice.
- Operations are communicated instead of state. To deliver an operation:
  1. The *prepare-update* implementation t is applied at the locally to prepare a representation of the operation.
  2. The *effect-update* implementation u is applied at the local and remote replicas if and only if the delivery precondition P is met, causing the desired update to take effect.

op-based CRDTs

An op-based CRDT is a 6-tuple (S, s^0, q, t, u, P).

- S, s^0, and q retain the same meaning as for the state-based CRDTs.
- S need not necessairly form a semi-lattice.
- Operations are communicated instead of state. To deliver an operation:
  1. The *prepare-update* implementation t is applied at the locally to prepare a representation of the operation.
  2. The *effect-update* implementation u is applied at the local and remote replicas if and only if the delivery precondition P is met, causing the desired update to take effect.

Example op-based CRDT

To illustrate the difference between state- and op-based CRDTs, here the analogue to G-Set$_s$:

$$\text{G-Set}_o(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ t : \lambda x.\, (\text{ins}, x) & \textit{Representation of the operation.} \\ u : \lambda p.\, s \cup \{(\text{snd } p)\} & \textit{Application of the operation.} \end{cases}$$

Example op-based CRDT

To illustrate the difference between state- and op-based CRDTs, here the analogue to G-Set$_s$:

$$\text{G-Set}_o(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ t : \lambda x.\, (\text{ins}, x) & \textit{Representation} \text{ of the operation.} \\ u : \lambda p.\, s \cup \{(\text{snd } p)\} & \textit{Application} \text{ of the operation.} \end{cases}$$

Example op-based CRDT

To illustrate the difference between state- and op-based CRDTs, here the analogue to G-Set$_s$:

$$\text{G-Set}_o(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ t : \lambda x.\, (\text{ins}, x) & \textit{Representation} \text{ of the operation.} \\ u : \lambda p.\, s \cup \{(\text{snd } p)\} & \textit{Application} \text{ of the operation.} \end{cases}$$

Example op-based CRDT

To illustrate the difference between state- and op-based CRDTs, here the analogue to G-Counter$_s$:

$$\text{G-Counter}'_o = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s.\ \sum_{i \in \mathcal{I}} s(i) \\ t : (\mathrm{inc}, i) \\ u : \lambda s, p.\ s\{i \mapsto s(i) + 1\} \end{cases}$$

$$\text{G-Counter}_o = \begin{cases} S : \mathbb{N}_0 \\ s^0 : 0 \\ q : \lambda s.\ s \\ t : \mathrm{inc} \\ u : \lambda s, p.\ s + 1 \end{cases}$$

Example op-based CRDT

To illustrate the difference between state- and op-based CRDTs, here the analogue to G-Counter$_s$:

$$\text{G-Counter}'_o = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s.\ \sum_{i \in \mathcal{I}} s(i) \\ t : (\text{inc}, i) \\ u : \lambda s, p.\ s\{i \mapsto s(i) + 1\} \end{cases}$$

$$\text{G-Counter}_o = \begin{cases} S : \mathbb{N}_0 \\ s^0 : 0 \\ q : \lambda s.\ s \\ t : \text{inc} \\ u : \lambda s, p.\ s + 1 \end{cases}$$

Example op-based CRDT

To illustrate the difference between state- and op-based CRDTs, here the analogue to G-Counter$_s$:

$$\text{G-Counter}'_o = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s.\ \sum_{i \in \mathcal{I}} s(i) \\ t : (\text{inc}, i) \\ u : \lambda s, p.\ s\{i \mapsto s(i) + 1\} \end{cases} \qquad \text{G-Counter}_o = \begin{cases} S : \mathbb{N}_0 \\ s^0 : 0 \\ q : \lambda s.\ s \\ t : \text{inc} \\ u : \lambda s, p.\ s + 1 \end{cases}$$

op- and state-based trade-offs

- state-based CRDTs are resilient to degenerate network behaviors, such as delaying, dropping, and reordering messages in transit, but suffer from large payload size
- op-based CRDTs have relatively small payload size, but require that the network deliver messages at-most-once

Is there a middle ground?

op- and state-based trade-offs

- state-based CRDTs are resilient to degenerate network behaviors, such as delaying, dropping, and reordering messages in transit, but suffer from large payload size
- op-based CRDTs have relatively small payload size, but require that the network deliver messages at-most-once

Is there a middle ground?

op- and state-based trade-offs

- state-based CRDTs are resilient to degenerate network behaviors, such as delaying, dropping, and reordering messages in transit, but suffer from large payload size
- op-based CRDTs have relatively small payload size, but require that the network deliver messages at-most-once

Is there a middle ground?

op- and state-based trade-offs

- state-based CRDTs are resilient to degenerate network behaviors, such as delaying, dropping, and reordering messages in transit, but suffer from large payload size
- op-based CRDTs have relatively small payload size, but require that the network deliver messages at-most-once

Is there a middle ground?

δ-state CRDTs

Like state-based CRDTs, a δ-state CRDT is a 5-tuple: $(S, s^0, q, u^\delta, m^\delta)$ [Almeida et al., 2018].

- u^δ produces an δ-*mutation*, which is representative of the update.
- m^δ is capable of merging a state $s \in S$ with the δ-mutation produced by u^δ.

Goal: the size of a δ mutation should be smaller than the state.

Example δ-state CRDT

Recall the original state-based G-Set, and consider how it might be represented as a δ-state CRDT:

$$\text{G-Set}_s(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ u : \lambda x.\, s \cup \{x\} \\ m : \lambda s_1, s_2.\, s_1 \cup s_2 \end{cases}$$

Observe that both $u : S \to S \to S$ and $u^\delta : S \to S \to S$.

- Standard requirement from Almeida et al. [2018] (they let S for the G-Counter be $S : \mathcal{I} \hookrightarrow \mathbb{N}$).
- Not a requirement in this work.

Example δ-state CRDT

Recall the original state-based G-Set, and consider how it might be represented as a δ-state CRDT:

$$\text{G-Set}_\delta(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ u^\delta : \lambda x.\, \{x\} \\ m^\delta : \lambda s_1, s_2.\, s_1 \cup s_2 \end{cases}$$

Observe that both $u : S \to S \to S$ and $u^\delta : S \to S \to S$.

- Standard requirement from Almeida et al. [2018] (they let S for the G-Counter be $S : \mathcal{I} \hookrightarrow \mathbb{N}$).
- Not a requirement in this work.

Example δ-state CRDT

Recall the original state-based G-Set, and consider how it might be represented as a δ-state CRDT:

$$\text{G-Set}_\delta(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ u^\delta : \lambda x.\, \{x\} \\ m^\delta : \lambda s_1, s_2.\, s_1 \cup s_2 \end{cases}$$

Observe that both $u : S \to S \to S$ and $u^\delta : S \to S \to S$.

- Standard requirement from Almeida et al. [2018] (they let S for the G-Counter be $S : \mathcal{I} \hookrightarrow \mathbb{N}$).
- Not a requirement in this work.

Example δ-state CRDT

Recall the original state-based G-Set, and consider how it might be represented as a δ-state CRDT:

$$\text{G-Set}_\delta(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ u^\delta : \lambda x.\, \{x\} \\ m^\delta : \lambda s_1, s_2.\, s_1 \cup s_2 \end{cases}$$

Observe that both $u : S \to S \to S$ and $u^\delta : S \to S \to S$.

- Standard requirement from Almeida et al. [2018] (they let S for the G-Counter be $S : \mathcal{I} \hookrightarrow \mathbb{N}$).
- Not a requirement in this work.

Example δ-state CRDT

Recall the original state-based G-Set, and consider how it might be represented as a δ-state CRDT:

$$\text{G-Set}_\delta(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ u^\delta : \lambda x.\, \{x\} \\ m^\delta : \lambda s_1, s_2.\, s_1 \cup s_2 \end{cases}$$

Observe that both $u : S \to S \to S$ and $u^\delta : S \to S \to S$.

- Standard requirement from Almeida et al. [2018] (they let S for the G-Counter be $S : \mathcal{I} \hookrightarrow \mathbb{N}$).
- Not a requirement in this work.

Example δ-state CRDT (G-Counter)

Let's consider the state- and δ-state encodings of the G-Counter:

$$\text{G-Counter}_s = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s. \sum_{i \in \mathcal{I}} s(i) \\ u : \lambda s, i. \, s\{i \mapsto s(i) + 1\} \\ m : \lambda s_1, s_2. \, [\max\{s_1(i), s_2(i)\} : i \in \mathrm{dom}(s_1) \cup \mathrm{dom}(s_2)] \end{cases}$$

Use the notation $\{i \mapsto x\}$ to encode an *update* (index, new value) in the vector.

Example δ-state CRDT (G-Counter)

Let's consider the state- and δ-state encodings of the G-Counter:

$$\text{G-Counter}_s = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s.\ \sum_{i \in \mathcal{I}} s(i) \\ u : \lambda s, i.\ s\{i \mapsto s(i) + 1\} \\ m : \lambda s_1, s_2.\ [\max\{s_1(i), s_2(i)\} : i \in \mathrm{dom}(s_1) \cup \mathrm{dom}(s_2)] \end{cases}$$

Use the notation $\{i \mapsto x\}$ to encode an *update* (index, new value) in the vector.

Example δ-state CRDT (G-Counter)

Let's consider the state- and δ-state encodings of the G-Counter:

$$\text{G-Counter}_\delta = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s.\ \sum_{i \in \mathcal{I}} s(i) \\ u^\delta : \lambda s, i.\ \{i \mapsto s(i) + 1\} \\ m^\delta : \lambda s_1, s_2.\ [\max\{s_1(i), s_2(i)\} : i \in \operatorname{dom}(s_1) \cup \operatorname{dom}(s_2)] \end{cases}$$

Use the notation $\{i \mapsto x\}$ to encode an *update* (index, new value) in the vector.

SEC & δ-CRDTs?

1. We have a "best-of-both-worlds" CRDT: the δ-state CRDT.
2. Small update payload (more like $\mathcal{O}$(size of update) instead of $\mathcal{O}(|\mathcal{I}|)$).
3. m^δ is still elegant: commutative, associative, and idempotent $\Rightarrow$ weak network requirements (opposed to op-based CRDTs).

Big question: does it satisfy SEC?

SEC & δ-CRDTs?

1. We have a "best-of-both-worlds" CRDT: the δ-state CRDT.
2. Small update payload (more like $\mathcal{O}$(size of update) instead of $\mathcal{O}(|\mathcal{I}|)$).
3. m^δ is still elegant: commutative, associative, and idempotent $\Rightarrow$ weak network requirements (opposed to op-based CRDTs).

Big question: does it satisfy SEC?

SEC & δ-CRDTs?

1. We have a "best-of-both-worlds" CRDT: the δ-state CRDT.
2. Small update payload (more like $\mathcal{O}$(size of update) instead of $\mathcal{O}(|\mathcal{I}|)$).
3. m^δ is still elegant: commutative, associative, and idempotent $\Rightarrow$ weak network requirements (opposed to op-based CRDTs).

Big question: does it satisfy SEC?

SEC & δ-CRDTs?

1. We have a "best-of-both-worlds" CRDT: the δ-state CRDT.
2. Small update payload (more like $\mathcal{O}$(size of update) instead of $\mathcal{O}(|\mathcal{I}|)$).
3. m^{δ} is still elegant: commutative, associative, and idempotent $\Rightarrow$ weak network requirements (opposed to op-based CRDTs).

Big question: does it satisfy SEC?

SEC & δ-CRDTs?

1. We have a "best-of-both-worlds" CRDT: the δ-state CRDT.
2. Small update payload (more like $\mathcal{O}$(size of update) instead of $\mathcal{O}(|\mathcal{I}|)$).
3. m^δ is still elegant: commutative, associative, and idempotent $\Rightarrow$ weak network requirements (opposed to op-based CRDTs).

Big question: does it satisfy SEC?

The rest of the talk

1. Answer the question of "do δ-state CRDTs achieve SEC?" in the affirmative, with a mechanically checked proof.
2. Build our proofs on the work of Gomes et al. [2017], verification library in Isabelle/HOL for op-based CRDTs.
3. State two reductions for viewing state- and δ-state based CRDTs as op-based.
4. Overview of our proofs.
5. Future directions.

The rest of the talk

1. Answer the question of "do δ-state CRDTs achieve SEC?" in the affirmative, with a mechanically checked proof.
2. Build our proofs on the work of Gomes et al. [2017], verification library in Isabelle/HOL for op-based CRDTs.
3. State two reductions for viewing state- and δ-state based CRDTs as op-based.
4. Overview of our proofs.
5. Future directions.

The rest of the talk

1. Answer the question of "do δ-state CRDTs achieve SEC?" in the affirmative, with a mechanically checked proof.
2. Build our proofs on the work of Gomes et al. [2017], verification library in Isabelle/HOL for op-based CRDTs.
3. State two reductions for viewing state- and δ-state based CRDTs as op-based.
4. Overview of our proofs.
5. Future directions.

The rest of the talk

1. Answer the question of "do δ-state CRDTs achieve SEC?" in the affirmative, with a mechanically checked proof.
2. Build our proofs on the work of Gomes et al. [2017], verification library in Isabelle/HOL for op-based CRDTs.
3. State two reductions for viewing state- and δ-state based CRDTs as op-based.
4. Overview of our proofs.
5. Future directions.

The rest of the talk

1. Answer the question of "do δ-state CRDTs achieve SEC?" in the affirmative, with a mechanically checked proof.
2. Build our proofs on the work of Gomes et al. [2017], verification library in Isabelle/HOL for op-based CRDTs.
3. State two reductions for viewing state- and δ-state based CRDTs as op-based.
4. Overview of our proofs.
5. Future directions.

Reduction I: state- to op-based

We have a type mismatch: want to verify properties of δ-state CRDTs, but library is designed for verifying op-based CRDTs.

- Design a reduction from δ-state CRDTs to op-based.
- Convince ourselves of its correctness.
- Encode δ-state CRDTs as op-based in Isabelle, write proofs over the *encoded* CRDTs.

Two reductions: state- to op-based, then δ- to op-based.

- Call these $\phi_{\text{state}\to\text{op}}$ and $\phi_{\delta\to\text{op}}$, respectively.
- First is a "warm-up" to illustrate the general shape of these reductions.
- Latter is the reduction we use in our proofs.

Reduction I: state- to op-based

We have a type mismatch: want to verify properties of δ-state CRDTs, but library is designed for verifying op-based CRDTs.

- Design a reduction from δ-state CRDTs to op-based.
- Convince ourselves of its correctness.
- Encode δ-state CRDTs as op-based in Isabelle, write proofs over the *encoded* CRDTs.

Two reductions: state- to op-based, then δ- to op-based.

- Call these $\phi_{\text{state}\to\text{op}}$ and $\phi_{\delta\to\text{op}}$, respectively.
- First is a "warm-up" to illustrate the general shape of these reductions.
- Latter is the reduction we use in our proofs.

Reduction I: state- to op-based

We have a type mismatch: want to verify properties of δ-state CRDTs, but library is designed for verifying op-based CRDTs.

- Design a reduction from δ-state CRDTs to op-based.
- Convince ourselves of its correctness.
- Encode δ-state CRDTs as op-based in Isabelle, write proofs over the *encoded* CRDTs.

Two reductions: state- to op-based, then δ- to op-based.

- Call these $\phi_{\text{state}\to\text{op}}$ and $\phi_{\delta\to\text{op}}$, respectively.
- First is a "warm-up" to illustrate the general shape of these reductions.
- Latter is the reduction we use in our proofs.

Reduction I: state- to op-based

We have a type mismatch: want to verify properties of δ-state CRDTs, but library is designed for verifying op-based CRDTs.

- Design a reduction from δ-state CRDTs to op-based.
- Convince ourselves of its correctness.
- Encode δ-state CRDTs as op-based in Isabelle, write proofs over the *encoded* CRDTs.

Two reductions: state- to op-based, then δ- to op-based.

- Call these $\phi_{\text{state}\to\text{op}}$ and $\phi_{\delta\to\text{op}}$, respectively.
- First is a "warm-up" to illustrate the general shape of these reductions.
- Latter is the reduction we use in our proofs.

Reduction I: state- to op-based

We have a type mismatch: want to verify properties of δ-state CRDTs, but library is designed for verifying op-based CRDTs.

- Design a reduction from δ-state CRDTs to op-based.
- Convince ourselves of its correctness.
- Encode δ-state CRDTs as op-based in Isabelle, write proofs over the *encoded* CRDTs.

Two reductions: state- to op-based, then δ- to op-based.

- Call these $\phi_{\text{state}\to\text{op}}$ and $\phi_{\delta\to\text{op}}$, respectively.
- First is a “warm-up” to illustrate the general shape of these reductions.
- Latter is the reduction we use in our proofs.

Reduction I: state- to op-based

We have a type mismatch: want to verify properties of δ-state CRDTs, but library is designed for verifying op-based CRDTs.

- Design a reduction from δ-state CRDTs to op-based.
- Convince ourselves of its correctness.
- Encode δ-state CRDTs as op-based in Isabelle, write proofs over the *encoded* CRDTs.

Two reductions: state- to op-based, then δ- to op-based.

- Call these $\phi_{\text{state}\to\text{op}}$ and $\phi_{\delta\to\text{op}}$, respectively.
- First is a "warm-up" to illustrate the general shape of these reductions.
- Latter is the reduction we use in our proofs.

Reduction I: state- to op-based

We have a type mismatch: want to verify properties of δ-state CRDTs, but library is designed for verifying op-based CRDTs.

- Design a reduction from δ-state CRDTs to op-based.
- Convince ourselves of its correctness.
- Encode δ-state CRDTs as op-based in Isabelle, write proofs over the *encoded* CRDTs.

Two reductions: state- to op-based, then δ- to op-based.

- Call these $\phi_{\text{state}\to\text{op}}$ and $\phi_{\delta\to\text{op}}$, respectively.
- First is a "warm-up" to illustrate the general shape of these reductions.
- Latter is the reduction we use in our proofs.

Reduction I: state- to op-based

We have a type mismatch: want to verify properties of δ-state CRDTs, but library is designed for verifying op-based CRDTs.

- Design a reduction from δ-state CRDTs to op-based.
- Convince ourselves of its correctness.
- Encode δ-state CRDTs as op-based in Isabelle, write proofs over the *encoded* CRDTs.

Two reductions: state- to op-based, then δ- to op-based.

- Call these $\phi_{\text{state}\to\text{op}}$ and $\phi_{\delta\to\text{op}}$, respectively.
- First is a “warm-up” to illustrate the general shape of these reductions.
- Latter is the reduction we use in our proofs.

Reduction I: state- to op-based

Want a reduction of the following form:

$$\phi_{\text{state}\to\text{op}} : \underbrace{(S, s^0, q, u, m)}_{\text{state-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

Simple idea:

- Let state (specifically: S, s^0, q) be identical under the reduction.[1]
- Let t return the result of (the state-based) u.
- Let u perform as (the state-based) m.
- Let P always be enabled.

That is: let the op-based reduction of a state-based CRDT the CRDT which applies updates by performing a state-based merge.

[1] Can often be more clever than this (for eg., op-based G-Counter, but simplifies the reduction.)

Reduction I: state- to op-based

Want a reduction of the following form:

$$\phi_{\text{state}\to\text{op}} : \underbrace{(S, s^0, q, u, m)}_{\text{state-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

Simple idea:

- Let state (specifically: S, s^0, q) be identical under the reduction.[1]
- Let t return the result of (the state-based) u.
- Let u perform as (the state-based) m.
- Let P always be enabled.

That is: let the op-based reduction of a state-based CRDT the CRDT which applies updates by performing a state-based merge.

[1] Can often be more clever than this (for eg., op-based G-Counter, but simplifies the reduction.)

Reduction I: state- to op-based

Want a reduction of the following form:

$$\phi_{\text{state}\to\text{op}} : \underbrace{(S, s^0, q, u, m)}_{\text{state-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

Simple idea:

- Let state (specifically: S, s^0, q) be identical under the reduction.[1]
- Let t return the result of (the state-based) u.
- Let u perform as (the state-based) m.
- Let P always be enabled.

That is: let the op-based reduction of a state-based CRDT the CRDT which applies updates by performing a state-based merge.

[1] Can often be more clever than this (for eg., op-based G-Counter, but simplifies the reduction.)

Reduction I: state- to op-based

Want a reduction of the following form:

$$\phi_{\text{state}\to\text{op}} : \underbrace{(S, s^0, q, u, m)}_{\text{state-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

Simple idea:

- Let state (specifically: S, s^0, q) be identical under the reduction.[1]
- Let t return the result of (the state-based) u.
- Let u perform as (the state-based) m.
- Let P always be enabled.

That is: let the op-based reduction of a state-based CRDT the CRDT which applies updates by performing a state-based merge.

[1] Can often be more clever than this (for eg., op-based G-Counter, but simplifies the reduction.)

Reduction I: state- to op-based

Want a reduction of the following form:

$$\phi_{\text{state}\to\text{op}} : \underbrace{(S, s^0, q, u, m)}_{\text{state-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

Simple idea:

- Let state (specifically: S, s^0, q) be identical under the reduction.[1]
- Let t return the result of (the state-based) u.
- Let u perform as (the state-based) m.
- Let P always be enabled.

That is: let the op-based reduction of a state-based CRDT the CRDT which applies updates by performing a state-based merge.

[1] Can often be more clever than this (for eg., op-based G-Counter, but simplifies the reduction.)

Reduction I: state- to op-based

Want a reduction of the following form:

$$\phi_{\text{state}\to\text{op}} : \underbrace{(S, s^0, q, u, m)}_{\text{state-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

Simple idea:

- Let state (specifically: S, s^0, q) be identical under the reduction.[1]
- Let t return the result of (the state-based) u.
- Let u perform as (the state-based) m.
- Let P always be enabled.

That is: let the op-based reduction of a state-based CRDT the CRDT which applies updates by performing a state-based merge.

[1]Can often be more clever than this (for eg., op-based G-Counter, but simplifies the reduction.)

Reduction I: state- to op-based

Want a reduction of the following form:

$$\phi_{\text{state}\to\text{op}} : \underbrace{(S, s^0, q, u, m)}_{\text{state-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

Simple idea:

- Let state (specifically: S, s^0, q) be identical under the reduction.[1]
- Let t return the result of (the state-based) u.
- Let u perform as (the state-based) m.
- Let P always be enabled.

That is: let the op-based reduction of a state-based CRDT the CRDT which applies updates by performing a state-based merge.

[1] Can often be more clever than this (for eg., op-based G-Counter, but simplifies the reduction.)

Reduction I: state- to op-based

Maxim

A state-based CRDT is an op-based CRDT where the prepare-update phase returns the updated state, and the effect-update is a join of two states.

Reduction I: state- to op-based

Abstract conversion from a state- to op-based CRDT under ϕ:

$$C_0 = \begin{cases} S_o : S \\ s_o^0 : s^0 \\ q_o : q \\ t_o : \lambda p.\, u(p\ldots) \\ u_o : \lambda s_2.\, m(s^t, s_2) \end{cases}$$

Reduction I: state- to op-based

Abstract conversion from a state- to op-based CRDT under ϕ:

$$C_0 = \begin{cases} S_o : S \\ s_o^0 : s^0 \\ q_o : q \\ t_o : \lambda p.\, u(p\ldots) \\ u_o : \lambda s_2.\, m(s^t, s_2) \end{cases}$$

Reduction I: state- to op-based

Abstract conversion from a state- to op-based CRDT under ϕ:

$$C_0 = \begin{cases} S_o : S \\ s_o^0 : s^0 \\ q_o : q \\ t_o : \lambda p.\ u(p\ldots) \\ u_o : \lambda s_2.\ m(s^t, s_2) \end{cases}$$

Reduction I: state- to op-based

Abstract conversion from a state- to op-based CRDT under ϕ:

$$C_0 = \begin{cases} S_o : S \\ s_o^0 : s^0 \\ q_o : q \\ t_o : \lambda p.\, u(p\ldots) \\ u_o : \lambda s_2.\, m(s^t, s_2) \end{cases}$$

Reduction II: δ- to op-based

Want a reduction of the following form:

$$\phi_{\delta\to\text{op}} : \underbrace{(S, s^0, q, u^\delta, m^\delta)}_{\delta\text{-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

General idea:

- Let S be the type of each state and T be the type of the δ-fragments.
- Let $t : S \to S \to T$ act like the *difference* between successive states.
- Let $u : S \to T \to S$ act like the pseudo-inverse of t which "unwinds" the state.
- Let P be always enabled.

That is: let the op-based reduction of a δ-state CRDT be the CRDT which applies updates over the δ-*fragments* of a state.

Reduction II: δ- to op-based

Want a reduction of the following form:

$$\phi_{\delta \to \text{op}} : \underbrace{(S, s^0, q, u^\delta, m^\delta)}_{\delta\text{-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

General idea:

- Let S be the type of each state and T be the type of the δ-fragments.
- Let $t : S \to S \to T$ act like the *difference* between successive states.
- Let $u : S \to T \to S$ act like the pseudo-inverse of t which "unwinds" the state.
- Let P be always enabled.

That is: let the op-based reduction of a δ-state CRDT be the CRDT which applies updates over the δ-*fragments* of a state.

Reduction II: δ- to op-based

Want a reduction of the following form:

$$\phi_{\delta\to\text{op}} : \underbrace{(S, s^0, q, u^\delta, m^\delta)}_{\delta\text{-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

General idea:

- Let S be the type of each state and T be the type of the δ-fragments.
- Let $t : S \to S \to T$ act like the *difference* between successive states.
- Let $u : S \to T \to S$ act like the pseudo-inverse of t which "unwinds" the state.
- Let P be always enabled.

That is: let the op-based reduction of a δ-state CRDT be the CRDT which applies updates over the δ-*fragments* of a state.

Reduction II: δ- to op-based

Want a reduction of the following form:

$$\phi_{\delta\to\text{op}} : \underbrace{(S, s^0, q, u^\delta, m^\delta)}_{\delta\text{-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

General idea:

- Let S be the type of each state and T be the type of the δ-fragments.
- Let $t : S \to S \to T$ act like the *difference* between successive states.
- Let $u : S \to T \to S$ act like the pseudo-inverse of t which "unwinds" the state.
- Let P be always enabled.

That is: let the op-based reduction of a δ-state CRDT be the CRDT which applies updates over the δ-*fragments* of a state.

Reduction II: δ- to op-based

Want a reduction of the following form:

$$\phi_{\delta\to\text{op}} : \underbrace{(S, s^0, q, u^\delta, m^\delta)}_{\delta\text{-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

General idea:

- Let S be the type of each state and T be the type of the δ-fragments.
- Let $t : S \to S \to T$ act like the *difference* between successive states.
- Let $u : S \to T \to S$ act like the pseudo-inverse of t which "unwinds" the state.
- Let P be always enabled.

That is: let the op-based reduction of a δ-state CRDT be the CRDT which applies updates over the δ-*fragments* of a state.

Reduction II: δ- to op-based

Want a reduction of the following form:

$$\phi_{\delta \to \text{op}} : \underbrace{(S, s^0, q, u^\delta, m^\delta)}_{\delta\text{-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

General idea:

- Let S be the type of each state and T be the type of the δ-fragments.
- Let $t : S \to S \to T$ act like the *difference* between successive states.
- Let $u : S \to T \to S$ act like the pseudo-inverse of t which "unwinds" the state.
- Let P be always enabled.

That is: let the op-based reduction of a δ-state CRDT be the CRDT which applies updates over the δ-*fragments* of a state.

Reduction II: δ- to op-based

Want a reduction of the following form:

$$\phi_{\delta \to \text{op}} : \underbrace{(S, s^0, q, u^\delta, m^\delta)}_{\delta\text{-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

General idea:

- Let S be the type of each state and T be the type of the δ-fragments.
- Let $t : S \to S \to T$ act like the *difference* between successive states.
- Let $u : S \to T \to S$ act like the pseudo-inverse of t which “unwinds” the state.
- Let P be always enabled.

That is: let the op-based reduction of a δ-state CRDT be the CRDT which applies updates over the δ-*fragments* of a state.

Reduction II: δ- to op-based

Maxim

A δ-state based CRDT is an op-based CRDT whose messages are δ-fragments, and whose operation is a pseudo-join between the current state and the δ fragment.

Reduction II: δ- to op-based

Example: apply $\phi_{\delta \to \mathsf{op}}$ to the δ-state G-Set.
Two questions:

1. What is the δ-fragment between two successive states $\Rightarrow$ what is t?
2. How to "join" a δ-fragment with our current state $\Rightarrow$ what is u?

Two answers:

1. Set difference.
2. Set union.

Reduction II: δ- to op-based

Example: apply $\phi_{\delta \to \mathsf{op}}$ to the δ-state G-Set.

Two questions:

1. What is the δ-fragment between two successive states $\Rightarrow$ what is t?
2. How to "join" a δ-fragment with our current state $\Rightarrow$ what is u?

Two answers:

1. Set difference.
2. Set union.

Reduction II: δ- to op-based

Example: apply $\phi_{\delta \to \mathsf{op}}$ to the δ-state G-Set.

Two questions:

1. What is the δ-fragment between two successive states $\Rightarrow$ what is t?
2. How to "join" a δ-fragment with our current state $\Rightarrow$ what is u?

Two answers:

1. Set difference.
2. Set union.

Reduction II: δ- to op-based

Example: apply $\phi_{\delta\to\text{op}}$ to the δ-state G-Set.
Two questions:

1. What is the δ-fragment between two successive states $\Rightarrow$ what is t?
2. How to "join" a δ-fragment with our current state $\Rightarrow$ what is u?

Two answers:

1. Set difference.
2. Set union.

Reduction II: δ- to op-based

Example: apply $\phi_{\delta \to \mathsf{op}}$ to the δ-state G-Set.

Two questions:

1. What is the δ-fragment between two successive states $\Rightarrow$ what is t?
2. How to "join" a δ-fragment with our current state $\Rightarrow$ what is u?

Two answers:

1. Set difference.
2. Set union.

Reduction II: δ- to op-based

Let's consider how $\phi_{\delta \to \mathsf{op}}$ behaves on the G-Set CRDT:

$$\phi_{\delta \to \mathsf{op}}(\text{G-Set}(\mathcal{X})) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ t : \lambda s_1, s_2.\, s_2 \setminus s_1 \\ u : \lambda s_2.\, s \cup s_2 \end{cases}$$

Example of reducing a δ-state CRDT to an op-based one where the type of the state *and* δ-fragment are the same (ie., $S = T = \mathcal{P}(\mathcal{X})$).

Reduction II: δ- to op-based

Let's consider how $\phi_{\delta \to \mathsf{op}}$ behaves on the G-Set CRDT:

$$\phi_{\delta \to \mathsf{op}}(\text{G-Set}(\mathcal{X})) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ t : \lambda s_1, s_2.\, s_2 \setminus s_1 \\ u : \lambda s_2.\, s \cup s_2 \end{cases}$$

Example of reducing a δ-state CRDT to an op-based one where the type of the state *and* δ-fragment are the same (ie., $S = T = \mathcal{P}(\mathcal{X})$).

Reduction II: δ- to op-based

Let's consider how $\phi_{\delta \to \text{op}}$ behaves on the G-Set CRDT:

$$\phi_{\delta \to \text{op}}(\text{G-Set}(\mathcal{X})) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ t : \lambda s_1, s_2.\, s_2 \setminus s_1 \\ u : \lambda s_2.\, s \cup s_2 \end{cases}$$

Example of reducing a δ-state CRDT to an op-based one where the type of the state *and* δ-fragment are the same (ie., $S = T = \mathcal{P}(\mathcal{X})$).

Reduction II: δ- to op-based

Let's consider how $\phi_{\delta \to \text{op}}$ behaves on the G-Set CRDT:

$$\phi_{\delta \to \text{op}}(\text{G-Set}(\mathcal{X})) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ t : \lambda s_1, s_2.\, s_2 \setminus s_1 \\ u : \lambda s_2.\, s \cup s_2 \end{cases}$$

Example of reducing a δ-state CRDT to an op-based one where the type of the state *and* δ-fragment are the same (ie., $S = T = \mathcal{P}(\mathcal{X})$).

Reduction II: δ- to op-based

Let's consider how $\phi_{\delta \to \mathsf{op}}$ behaves on the G-Set CRDT:

$$\phi_{\delta \to \mathsf{op}}(\text{G-Set}(\mathcal{X})) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ t : \lambda s_1, s_2.\, s_2 \setminus s_1 \\ u : \lambda s_2.\, s \cup s_2 \end{cases}$$

Example of reducing a δ-state CRDT to an op-based one where the type of the state *and* δ-fragment are the same (ie., $S = T = \mathcal{P}(\mathcal{X})$).

Reduction II: δ- to op-based

Let's consider how $\phi_{\delta \to \mathsf{op}}$ behaves on the G-Counter CRDT:

$$\phi_{\delta \to \mathsf{op}}(\text{G-Counter}) = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0j] \\ q : \lambda. \sum_{i \in \mathcal{I}} s(i) \\ t : \min\limits_{\substack{i \in \mathcal{I} \\ s_1[i] \neq s_2[i]}} (i, s_2[i]) \\ u : \lambda s, t.\, s\{(\mathsf{fst}\ t) \mapsto (\mathsf{snd}\ t)\} \end{cases}$$

Example of reducing a δ-state CRDT to an op-based one where the type of the state *and* δ-fragment are not same (ie., $S = \mathbb{N}_0^{|\mathcal{I}|}$, but $T = ('id, int)$.

Reduction II: δ- to op-based

Let's consider how $\phi_{\delta\to\text{op}}$ behaves on the G-Counter CRDT:

$$\phi_{\delta\to\text{op}}(\text{G-Counter}) = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0j] \\ q : \lambda. \sum_{i\in\mathcal{I}} s(i) \\ t : \min_{\substack{i\in\mathcal{I} \\ s_1[i]\neq s_2[i]}} (i, s_2[i]) \\ u : \lambda s, t.\, s\{(\text{fst } t) \mapsto (\text{snd } t)\} \end{cases}$$

Example of reducing a δ-state CRDT to an op-based one where the type of the state *and* δ-fragment are not same (ie., $S = \mathbb{N}_0^{|\mathcal{I}|}$, but $T = ('id, int)$.

Reduction II: δ- to op-based

Let's consider how $\phi_{\delta \to \text{op}}$ behaves on the G-Counter CRDT:

$$\phi_{\delta \to \text{op}}(\text{G-Counter}) = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0j] \\ q : \lambda. \sum_{i \in \mathcal{I}} s(i) \\ t : \min_{\substack{i \in \mathcal{I} \\ s_1[i] \neq s_2[i]}} (i, s_2[i]) \\ u : \lambda s, t. \, s\{(\text{fst } t) \mapsto (\text{snd } t)\} \end{cases}$$

Example of reducing a δ-state CRDT to an op-based one where the type of the state *and* δ-fragment are not same (ie., $S = \mathbb{N}_0^{|\mathcal{I}|}$, but $T = ('id, int)$.

Reduction II: δ- to op-based

Let's consider how $\phi_{\delta\to\mathsf{op}}$ behaves on the G-Counter CRDT:

$$\phi_{\delta\to\mathsf{op}}(\text{G-Counter}) = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0j] \\ q : \lambda. \sum_{i\in\mathcal{I}} s(i) \\ t : \min\limits_{\substack{i\in\mathcal{I} \\ s_1[i]\neq s_2[i]}} (i, s_2[i]) \\ u : \lambda s, t.\, s\{(\text{fst } t) \mapsto (\text{snd } t)\} \end{cases}$$

Example of reducing a δ-state CRDT to an op-based one where the type of the state *and* δ-fragment are not same (ie., $S = \mathbb{N}_0^{|\mathcal{I}|}$, but $T = ('id, int)$.

Reduction II: δ- to op-based

Let's consider how $\phi_{\delta\to\mathsf{op}}$ behaves on the G-Counter CRDT:

$$\phi_{\delta\to\mathsf{op}}(\text{G-Counter}) = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0j] \\ q : \lambda. \sum_{i\in\mathcal{I}} s(i) \\ t : \min_{\substack{i\in\mathcal{I} \\ s_1[i]\neq s_2[i]}} (i, s_2[i]) \\ u : \lambda s, t.\, s\{(\text{fst } t) \mapsto (\text{snd } t)\} \end{cases}$$

Example of reducing a δ-state CRDT to an op-based one where the type of the state *and* δ-fragment are not same (ie., $S = \mathbb{N}_0^{|\mathcal{I}|}$, but $T = ('id, int)$.

Motivating network relaxations

The network model from Gomes et al. [2017] is already fairly permissive:

1. Supports delaying and dropping of messages.
2. ...which implies that we can re-order messages on the network.

But, if messages are never *duplicated* we can't be sure that we're exercising the idempotency of $\sqcup$.

Motivating network relaxations

The network model from Gomes et al. [2017] is already fairly permissive:

1. Supports delaying and dropping of messages.
2. ...which implies that we can re-order messages on the network.

But, if messages are never *duplicated* we can't be sure that we're exercising the idempotency of $\sqcup$.

Motivating network relaxations

The network model from Gomes et al. [2017] is already fairly permissive:

1. Supports delaying and dropping of messages.
2. ...which implies that we can re-order messages on the network.

But, if messages are never *duplicated* we can't be sure that we're exercising the idempotency of $\sqcup$.

Motivating network relaxations

The network model from Gomes et al. [2017] is already fairly permissive:

1. Supports delaying and dropping of messages.
2. ...which implies that we can re-order messages on the network.

But, if messages are never *duplicated* we can't be sure that we're exercising the idempotency of $\sqcup$.

Network relaxation

locale *network* = *node-histories history*
  for *history* :: *nat* ⇒ *'msg event list* +
  fixes *msg-id* :: *'msg* ⇒ *'msgid*
  assumes *delivery-has-a-cause*:

$$\llbracket \textit{Deliver } m \in \textit{set } (\textit{history } i) \rrbracket \Longrightarrow \exists j.\ \textit{Broadcast } m \in \textit{set } (\textit{history } j)$$

and *deliver-locally*:

$$\llbracket \textit{Broadcast } m \in \textit{set } (\textit{history } i) \rrbracket \Longrightarrow \textit{Broadcast } m \sqsubset^{i} \textit{Deliver } m$$

and *msg-id-unique*:

$$\llbracket \textit{Broadcast } m1 \in \textit{set } (\textit{history } i);\ \textit{Broadcast } m2 \in \textit{set } (\textit{history } j);\ \textit{msg-id } m1 = \textit{msg-id } m2 \rrbracket \Longrightarrow i = j \wedge m1 = m2$$

Network relaxation

locale *network* = *node-histories history*
for *history* :: *nat* ⇒ *'msg event list* +
fixes *msg-id* :: *'msg* ⇒ *'msgid*
assumes *delivery-has-a-cause*:

$$\llbracket \textit{Deliver } m \in \textit{set } (\textit{history } i) \rrbracket \Longrightarrow \exists j.\ \textit{Broadcast } m \in \textit{set } (\textit{history } j)$$

and *deliver-locally*: $\llbracket \textit{Broadcast } m \in \textit{set } (\textit{history } i) \rrbracket \Longrightarrow \textit{Broadcast } m \sqsubset^i \textit{Deliver } m$

and *msg-id-unique*: $\llbracket \textit{Broadcast } m1 \in \textit{set } (\textit{history } i); \textit{Broadcast } m2 \in \textit{set } (\textit{history } j); \textit{msg-id } m1 = \textit{msg-id } m2 \rrbracket \Longrightarrow i = j \wedge m1 = m2$

Proof strategy

1. First, remove the assumption uniqueness assumption.
2. Identify the set of broken proofs. In each broken proof, do the following:
    1. Identify the earliest broken proof step.
    2. Delete it and all proof steps following it.
    3. Replace the proof body with the term *sorry*.
3. In any order, consider a proof which ends with *sorry*, and repair the proof.

All broken goals were able to be solved with Isabelle's built-in proof search (suggesting that this assumption was not used heavily in the work of Gomes et al. [2017]).

Proof strategy

1. First, remove the assumption uniqueness assumption.
2. Identify the set of broken proofs. In each broken proof, do the following:
   1. Identify the earliest broken proof step.
   2. Delete it and all proof steps following it.
   3. Replace the proof body with the term *sorry*.
3. In any order, consider a proof which ends with *sorry*, and repair the proof.

All broken goals were able to be solved with Isabelle's built-in proof search (suggesting that this assumption was not used heavily in the work of Gomes et al. [2017]).

Proof strategy

1. First, remove the assumption uniqueness assumption.
2. Identify the set of broken proofs. In each broken proof, do the following:
   1. Identify the earliest broken proof step.
   2. Delete it and all proof steps following it.
   3. Replace the proof body with the term *sorry*.
3. In any order, consider a proof which ends with *sorry*, and repair the proof.

All broken goals were able to be solved with Isabelle's built-in proof search (suggesting that this assumption was not used heavily in the work of Gomes et al. [2017]).

Proof strategy

1. First, remove the assumption uniqueness assumption.
2. Identify the set of broken proofs. In each broken proof, do the following:
   1. Identify the earliest broken proof step.
   2. Delete it and all proof steps following it.
   3. Replace the proof body with the term *sorry*.
3. In any order, consider a proof which ends with *sorry*, and repair the proof.

All broken goals were able to be solved with Isabelle's built-in proof search (suggesting that this assumption was not used heavily in the work of Gomes et al. [2017]).

Proof strategy

1. First, remove the assumption uniqueness assumption.
2. Identify the set of broken proofs. In each broken proof, do the following:
   1. Identify the earliest broken proof step.
   2. Delete it and all proof steps following it.
   3. Replace the proof body with the term *sorry*.
3. In any order, consider a proof which ends with *sorry*, and repair the proof.

All broken goals were able to be solved with Isabelle's built-in proof search (suggesting that this assumption was not used heavily in the work of Gomes et al. [2017]).

Proof strategy

1. First, remove the assumption uniqueness assumption.
2. Identify the set of broken proofs. In each broken proof, do the following:
   1. Identify the earliest broken proof step.
   2. Delete it and all proof steps following it.
   3. Replace the proof body with the term *sorry*.
3. In any order, consider a proof which ends with *sorry*, and repair the proof.

All broken goals were able to be solved with Isabelle's built-in proof search (suggesting that this assumption was not used heavily in the work of Gomes et al. [2017]).

Proof strategy

1. First, remove the assumption uniqueness assumption.
2. Identify the set of broken proofs. In each broken proof, do the following:
    1. Identify the earliest broken proof step.
    2. Delete it and all proof steps following it.
    3. Replace the proof body with the term *sorry*.
3. In any order, consider a proof which ends with *sorry*, and repair the proof.

All broken goals were able to be solved with Isabelle's built-in proof search (suggesting that this assumption was not used heavily in the work of Gomes et al. [2017]).

Proof strategy

1. First, remove the assumption uniqueness assumption.
2. Identify the set of broken proofs. In each broken proof, do the following:
    1. Identify the earliest broken proof step.
    2. Delete it and all proof steps following it.
    3. Replace the proof body with the term *sorry*.
3. In any order, consider a proof which ends with *sorry*, and repair the proof.

All broken goals were able to be solved with Isabelle's built-in proof search (suggesting that this assumption was not used heavily in the work of Gomes et al. [2017]).

State-based G-Counter

type-synonym $('id)$ *state* $=$ $'id \Rightarrow int\ option$
type-synonym $('id)$ *operation* $=$ $'id\ state$

fun *option-max* :: $int\ option \Rightarrow int\ option \Rightarrow int\ option$ **where**
$option\text{-}max\ (Some\ a)\ (Some\ b) = Some\ (max\ a\ b)\ |$
$option\text{-}max\ x\ None = x\ |$
$option\text{-}max\ None\ y = y$

fun *inc* :: $'id \Rightarrow ('id\ state) \Rightarrow ('id\ operation)$ **where**
$inc\ who\ st = (case\ (st\ who)\ of$
$\quad None \Rightarrow st(who := Some\ 0)$
$\quad |\ Some\ c \Rightarrow st(who := Some\ (c + 1)))$

fun *gcounter-op* :: $('id\ operation) \Rightarrow ('id\ state) \rightharpoonup ('id\ state)$ **where**
$gcounter\text{-}op\ theirs\ ours = Some\ (\lambda\ x.\ option\text{-}max\ (theirs\ x)\ (ours\ x))$

State-based G-Counter

type-synonym ($'id$) *state* $=$ $'id \Rightarrow$ *int option*
type-synonym ($'id$) *operation* $=$ $'id$ *state*

fun *option-max* :: *int option* $\Rightarrow$ *int option* $\Rightarrow$ *int option* **where**
option-max (*Some a*) (*Some b*) $=$ *Some* (*max a b*) |
option-max x None $=$ *x* |
option-max None y $=$ *y*

fun *inc* :: $'id \Rightarrow$ ($'id$ *state*) $\Rightarrow$ ($'id$ *operation*) **where**
inc who st $=$ (*case* (*st who*) *of*
None $\Rightarrow$ *st*(*who* := *Some 0*)
| *Some c* $\Rightarrow$ *st*(*who* := *Some* (*c* + *1*)))

fun *gcounter-op* :: ($'id$ *operation*) $\Rightarrow$ ($'id$ *state*) $\rightharpoonup$ ($'id$ *state*) **where**
gcounter-op theirs ours $=$ *Some* (λ *x*. *option-max* (*theirs x*) (*ours x*))

State-based G-Counter

type-synonym ($'id$) *state* $=$ $'id \Rightarrow$ *int option*
type-synonym ($'id$) *operation* $=$ $'id$ *state*

fun *option-max* :: *int option* $\Rightarrow$ *int option* $\Rightarrow$ *int option* **where**
option-max (*Some a*) (*Some b*) $=$ *Some* (*max a b*) |
option-max x None $=$ *x* |
option-max None y $=$ *y*

fun *inc* :: $'id \Rightarrow$ ($'id$ *state*) $\Rightarrow$ ($'id$ *operation*) **where**
inc who st $=$ (*case* (*st who*) *of*
  None $\Rightarrow$ *st*(*who* $:=$ *Some 0*)
  | *Some c* $\Rightarrow$ *st*(*who* $:=$ *Some* ($c + 1$)))

fun *gcounter-op* :: ($'id$ *operation*) $\Rightarrow$ ($'id$ *state*) $\rightharpoonup$ ($'id$ *state*) **where**
gcounter-op theirs ours $=$ *Some* (λ *x. option-max* (*theirs x*) (*ours x*))

State-based G-Counter

type-synonym ($'id$) *state* $=$ $'id \Rightarrow$ *int option*
type-synonym ($'id$) *operation* $=$ $'id$ *state*

fun *option-max* :: *int option* $\Rightarrow$ *int option* $\Rightarrow$ *int option* **where**
option-max (*Some a*) (*Some b*) $=$ *Some* (*max a b*) |
option-max x None $=$ *x* |
option-max None y $=$ *y*

fun *inc* :: $'id \Rightarrow$ ($'id$ *state*) $\Rightarrow$ ($'id$ *operation*) **where**
inc who st $=$ (*case* (*st who*) *of*
None $\Rightarrow$ *st*(*who* := *Some 0*)
| *Some c* $\Rightarrow$ *st*(*who* := *Some* ($c + 1$)))

fun *gcounter-op* :: ($'id$ *operation*) $\Rightarrow$ ($'id$ *state*) $\rightharpoonup$ ($'id$ *state*) **where**
gcounter-op theirs ours $=$ *Some* (λ *x*. *option-max* (*theirs x*) (*ours x*))

State-based G-Counter

type-synonym $('id)$ *state* $=$ $'id \Rightarrow int\ option$
type-synonym $('id)$ *operation* $=$ $'id\ state$

fun $option\text{-}max :: int\ option \Rightarrow int\ option \Rightarrow int\ option$ **where**
$option\text{-}max\ (Some\ a)\ (Some\ b) = Some\ (max\ a\ b) \mid$
$option\text{-}max\ x\ None = x \mid$
$option\text{-}max\ None\ y = y$

fun $inc :: 'id \Rightarrow ('id\ state) \Rightarrow ('id\ operation)$ **where**
$inc\ who\ st = (case\ (st\ who)\ of$
$\quad None \Rightarrow st(who := Some\ 0)$
$\mid Some\ c \Rightarrow st(who := Some\ (c + 1)))$

fun $gcounter\text{-}op :: ('id\ operation) \Rightarrow ('id\ state) \rightharpoonup ('id\ state)$ **where**
$gcounter\text{-}op\ theirs\ ours = Some\ (\lambda\ x.\ option\text{-}max\ (theirs\ x)\ (ours\ x))$

State-based G-Counter

A few additional steps omitted here, including:

1. Proof that concurrent operations commute (ie., can be applied in arbitrary order and the resulting state is unchanged).
2. G-Counter convergence: corollary of the above, which states that *all* operations can be applied in any order.
3. Commutativity and associativity of option-max (idempotency proof is inferred automatically).

Then:

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
$\lambda ops.\ \exists\, xs\ i.\ xs\ \mathit{prefix\ of}\ i \wedge \mathit{node\text{-}deliver\text{-}messages}\ xs = ops\ \lambda\ x.\ \mathit{None}$

State-based G-Counter

A few additional steps omitted here, including:

1. Proof that concurrent operations commute (ie., can be applied in arbitrary order and the resulting state is unchanged).
2. G-Counter convergence: corollary of the above, which states that *all* operations can be applied in any order.
3. Commutativity and associativity of option-max (idempotency proof is inferred automatically).

Then:

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
$\lambda ops.\ \exists\, xs\ i.\ xs\ prefix\ of\ i \wedge node\text{-}deliver\text{-}messages\ xs = ops\ \lambda\ x.\ None$

State-based G-Counter

A few additional steps omitted here, including:

1. Proof that concurrent operations commute (ie., can be applied in arbitrary order and the resulting state is unchanged).
2. G-Counter convergence: corollary of the above, which states that *all* operations can be applied in any order.
3. Commutativity and associativity of option-max (idempotency proof is inferred automatically).

Then:

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
$\lambda ops.\ \exists\, xs\ i.\ xs\ prefix\ of\ i \wedge node\text{-}deliver\text{-}messages\ xs = ops\ \lambda\ x.\ None$

State-based G-Counter

A few additional steps omitted here, including:

1. Proof that concurrent operations commute (ie., can be applied in arbitrary order and the resulting state is unchanged).
2. G-Counter convergence: corollary of the above, which states that *all* operations can be applied in any order.
3. Commutativity and associativity of option-max (idempotency proof is inferred automatically).

Then:

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
$\lambda ops.\ \exists\, xs\ i.\ xs\ \mathit{prefix\ of}\ i \wedge \mathit{node\text{-}deliver\text{-}messages}\ xs = ops\ \lambda\ x.\ \mathit{None}$

State-based G-Counter

A few additional steps omitted here, including:

1. Proof that concurrent operations commute (ie., can be applied in arbitrary order and the resulting state is unchanged).
2. G-Counter convergence: corollary of the above, which states that *all* operations can be applied in any order.
3. Commutativity and associativity of option-max (idempotency proof is inferred automatically).

Then:

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
$\lambda ops.\ \exists\, xs\ i.\ xs\ prefix\ of\ i \wedge node\text{-}deliver\text{-}messages\ xs = ops\ \lambda\ x.\ None$

State-based G-Set

type-synonym ($'a$) *state* $=$ $'a$ *set*
type-synonym ($'a$) *operation* $=$ $'a$ *state*

fun *insert* $:: \ 'a \Rightarrow ('a \ state) \Rightarrow ('a \ operation)$ **where**
$insert \ a \ as = as \cup \{ \ a \ \}$

fun *gset-op* $:: ('a \ operation) \Rightarrow ('a \ state) \rightharpoonup ('a \ state)$ **where**
$gset\text{-}op \ a \ as = Some \ (\ as \cup a \ )$

Since we're using Isabelle's built-in set library, no additional substantial proofs required.

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
$\lambda ops. \ \exists xs \ i. \ xs \ prefix \ of \ i \land node\text{-}deliver\text{-}messages \ xs = ops \ \{\}$

State-based G-Set

type-synonym ('a) *state* = 'a *set*
type-synonym ('a) *operation* = 'a *state*

fun *insert* :: 'a ⇒ ('a *state*) ⇒ ('a *operation*) **where**
insert a as = *as* ∪ { *a* }

fun *gset-op* :: ('a *operation*) ⇒ ('a *state*) ⇀ ('a *state*) **where**
gset-op a as = *Some* (*as* ∪ *a*)

Since we're using Isabelle's built-in set library, no additional substantial proofs required.

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
$\lambda ops.\ \exists xs\ i.\ xs\ prefix\ of\ i \wedge node\text{-}deliver\text{-}messages\ xs = ops$ {}

State-based G-Set

type-synonym $('a)$ *state* $=$ $'a$ *set*
type-synonym $('a)$ *operation* $=$ $'a$ *state*

fun *insert* :: $'a \Rightarrow ('a\ state) \Rightarrow ('a\ operation)$ **where**
insert a as $= as \cup \{\ a\ \}$

fun *gset-op* :: $('a\ operation) \Rightarrow ('a\ state) \rightharpoonup ('a\ state)$ **where**
gset-op a as $=$ *Some* $(\ as \cup a\ )$

Since we're using Isabelle's built-in set library, no additional substantial proofs required.

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
$\lambda ops.\ \exists xs\ i.\ xs\ prefix\ of\ i \wedge node\text{-}deliver\text{-}messages\ xs = ops\ \{\}$

State-based G-Set

type-synonym $('a)$ *state* $=$ $'a$ *set*
type-synonym $('a)$ *operation* $=$ $'a$ *state*

fun $insert :: 'a \Rightarrow ('a\ state) \Rightarrow ('a\ operation)$ **where**
$insert\ a\ as = as \cup \{\ a\ \}$

fun $gset\text{-}op :: ('a\ operation) \Rightarrow ('a\ state) \rightharpoonup ('a\ state)$ **where**
$gset\text{-}op\ a\ as = Some\ (\ as \cup a\ )$

Since we're using Isabelle's built-in set library, no additional substantial proofs required.

sublocale *sec*: $strong\text{-}eventual\text{-}consistency\ weak\text{-}hb\ hb\ interp\text{-}msg$
$\lambda ops.\ \exists xs\ i.\ xs\ prefix\ of\ i \wedge node\text{-}deliver\text{-}messages\ xs = ops\ \{\}$

State-based G-Set

type-synonym $('a)$ *state* $=$ $'a$ *set*
type-synonym $('a)$ *operation* $=$ $'a$ *state*

fun *insert* $:: \; 'a \Rightarrow ('a \; state) \Rightarrow ('a \; operation)$ **where**
insert a as $=$ *as* $\cup \{ \; a \; \}$

fun *gset-op* $:: \; ('a \; operation) \Rightarrow ('a \; state) \rightharpoonup ('a \; state)$ **where**
gset-op a as $=$ *Some* $(\; as \cup a \;)$

Since we're using Isabelle's built-in set library, no additional substantial proofs required.

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
$\lambda ops. \; \exists xs \; i. \; xs \; prefix \; of \; i \wedge node\text{-}deliver\text{-}messages \; xs = ops \; \{\}$

State-based G-Set

type-synonym $('a)$ *state* $=$ $'a$ *set*
type-synonym $('a)$ *operation* $=$ $'a$ *state*

fun *insert* $::$ $'a \Rightarrow ('a\ state) \Rightarrow ('a\ operation)$ **where**
insert a as $=$ $as \cup \{\ a\ \}$

fun *gset-op* $::$ $('a\ operation) \Rightarrow ('a\ state) \rightharpoonup ('a\ state)$ **where**
gset-op a as $=$ $Some\ (\ as \cup a\ )$

Since we're using Isabelle's built-in set library, no additional substantial proofs required.

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
$\lambda ops.\ \exists\, xs\ i.\ xs\ prefix\ of\ i \wedge node\text{-}deliver\text{-}messages\ xs = ops\ \{\}$

δ-state G-Counter

type-synonym $('id)$ *state* $=$ $'id \Rightarrow$ *int option*
type-synonym $('id)$ *operation* $=$ $'id$ *state*

fun $inc :: 'id \Rightarrow ('id\ state) \Rightarrow ('id\ operation)$ **where**
$inc\ who\ st = (case\ (st\ who)\ of$
$\quad None \Rightarrow st(who := Some\ 0)$
$\quad |\ Some\ c \Rightarrow st(who := Some\ (c + 1)))$

δ-state G-Counter

type-synonym ($'id$) $state$ = $'id \Rightarrow int\ option$
type-synonym ($'id$) $operation$ = $'id \times int$

fun $inc :: 'id \Rightarrow ('id\ state) \Rightarrow ('id\ operation)$ **where**
$inc\ who\ st = (case\ (st\ who)\ of$
$\quad None \Rightarrow st(who := Some\ 0)$
$\quad |\ Some\ c \Rightarrow st(who := Some\ (c + 1)))$

δ-state G-Counter

type-synonym ($'id$) $state$ = $'id \Rightarrow int\ option$
type-synonym ($'id$) $operation$ = $'id \times int$

fun $inc :: 'id \Rightarrow ('id\ state) \Rightarrow ('id\ operation)$ **where**
$inc\ who\ st = (case\ (st\ who)\ of$
$\quad None \Rightarrow st(who := Some\ 0)$
$\ \mid Some\ c \Rightarrow st(who := Some\ (c + 1)))$

δ-state G-Counter

type-synonym $('id)$ *state* $=$ $'id \Rightarrow$ *int option*
type-synonym $('id)$ *operation* $=$ $'id \times$ *int*

fun *inc* :: $'id \Rightarrow ('id\ state) \Rightarrow ('id\ operation)$ **where**
inc who st $=$ (*who*, (*1* $+$ (*case* (*st who*) *of None* $\Rightarrow$ *0* | *Some* (*x*) $\Rightarrow$ *x*)))

δ-state G-Counter

type-synonym (*'id*) *state* = *'id* ⇒ *int option*
type-synonym (*'id*) *operation* = *'id* × *int*

fun *inc* :: *'id* ⇒ (*'id state*) ⇒ (*'id operation*) **where**
inc who st = (*who*, (*1* + (*case* (*st who*) *of None* ⇒ *0* | *Some* (*x*) ⇒ *x*)))

fun *op-to-state* :: (*'id operation*) ⇒ (*'id state*) **where**
op-to-state (*who*, *count*) = (λx. *if* x = *who then Some count else None*)

δ-state G-Counter

type-synonym $('id)$ *state* $= 'id \Rightarrow$ *int option*
type-synonym $('id)$ *operation* $= 'id \times$ *int*

fun *inc* :: $'id \Rightarrow ('id\ state) \Rightarrow ('id\ operation)$ **where**
inc who st $= (who, (1 + (case\ (st\ who)\ of\ None \Rightarrow 0 \mid Some\ (x) \Rightarrow x)))$

fun *op-to-state* :: $('id\ operation) \Rightarrow ('id\ state)$ **where**
op-to-state $(who, count) = (\lambda x.\ if\ x = who\ then\ Some\ count\ else\ None)$

fun *delta-gcounter-op* :: $('id\ operation) \Rightarrow ('id\ state) \rightharpoonup ('id\ state)$ **where**
delta-gcounter-op theirs ours $= Some\ (\lambda\ x.\ option\text{-}max\ ((op\text{-}to\text{-}state\ theirs)\ x)\ (ours\ x))$

δ-state G-Counter

type-synonym $('id)\ state = 'id \Rightarrow int\ option$
type-synonym $('id)\ operation = 'id \times int$

fun $inc :: 'id \Rightarrow ('id\ state) \Rightarrow ('id\ operation)$ **where**
$inc\ who\ st = (who, (1 + (case\ (st\ who)\ of\ None \Rightarrow 0 \mid Some\ (x) \Rightarrow x)))$

fun $op\text{-}to\text{-}state :: ('id\ operation) \Rightarrow ('id\ state)$ **where**
$op\text{-}to\text{-}state\ (who, count) = (\lambda x.\ if\ x = who\ then\ Some\ count\ else\ None)$

fun $delta\text{-}gcounter\text{-}op :: ('id\ operation) \Rightarrow ('id\ state) \rightharpoonup ('id\ state)$ **where**
$delta\text{-}gcounter\text{-}op\ theirs\ ours = Some\ (\lambda\ x.\ option\text{-}max\ ((op\text{-}to\text{-}state\ theirs)\ x)\ (ours\ x))$

δ-state G-Set

type-synonym $('a)$ *state* $=$ $'a$ *set*
type-synonym $('a)$ *operation* $=$ $'a$ *state*

fun *insert* $:: \ 'a \Rightarrow ('a\ state) \Rightarrow ('a\ operation)$ **where**
$insert\ a\ as = as \cup \{\ a\ \}$

fun *gset-op* $:: \ ('a\ operation) \Rightarrow ('a\ state) \rightharpoonup ('a\ state)$ **where**
$gset\text{-}op\ a\ as = Some\ (\ as \cup a\ )$

δ-state G-Set

$$
\begin{array}{l}
\textbf{type-synonym}\ ('a)\ \mathit{state} = \ 'a\ \mathit{set} \\
\textbf{type-synonym}\ ('a)\ \mathit{operation} = \ 'a
\end{array}
$$

$$
\begin{array}{l}
\textbf{fun}\ \mathit{insert} :: \ 'a \Rightarrow ('a\ \mathit{state}) \Rightarrow ('a\ \mathit{operation})\ \textbf{where} \\
\mathit{insert}\ a\ \mathit{as} = \mathit{as} \cup \{\ a\ \}
\end{array}
$$

$$
\begin{array}{l}
\textbf{fun}\ \mathit{gset\text{-}op} :: ('a\ \mathit{operation}) \Rightarrow ('a\ \mathit{state}) \rightharpoonup ('a\ \mathit{state})\ \textbf{where} \\
\mathit{gset\text{-}op}\ a\ \mathit{as} = \mathit{Some}\ (\ \mathit{as} \cup a\ )
\end{array}
$$

δ-state G-Set

type-synonym $('a)$ *state* $=$ $'a$ *set*
type-synonym $('a)$ *operation* $=$ $'a$

fun *insert* $::$ $'a \Rightarrow ('a\ state) \Rightarrow ('a\ operation)$ **where**
$insert\ a\ as = as \cup \{\ a\ \}$

fun *gset-op* $::$ $('a\ operation) \Rightarrow ('a\ state) \rightharpoonup ('a\ state)$ **where**
$gset\text{-}op\ a\ as = Some\ (\ as \cup a\ )$

δ-state G-Set

type-synonym $('a)$ *state* $=$ $'a$ *set*
type-synonym $('a)$ *operation* $=$ $'a$

fun *insert* $::$ $'a \Rightarrow ('a\ state) \Rightarrow ('a\ operation)$ **where**
insert a _ $=$ a

fun *gset-op* $::$ $('a\ operation) \Rightarrow ('a\ state) \rightharpoonup ('a\ state)$ **where**
gset-op a as $=$ *Some* $(\ as \cup a\ )$

δ-state G-Set

type-synonym $('a)$ *state* $= \ 'a$ *set*
type-synonym $('a)$ *operation* $= \ 'a$

fun *insert* $:: \ 'a \Rightarrow ('a \ state) \Rightarrow ('a \ operation)$ **where**
insert a - = a

fun *gset-op* $:: ('a \ operation) \Rightarrow ('a \ state) \rightharpoonup ('a \ state)$ **where**
gset-op a as $=$ *Some* $(\ as \cup a \ )$

δ-state G-Set

type-synonym $('a)$ *state* $=$ $'a$ *set*
type-synonym $('a)$ *operation* $=$ $'a$

fun *insert* $::$ $'a \Rightarrow ('a\ state) \Rightarrow ('a\ operation)$ **where**
insert a - $=$ *a*

fun *delta-gset-op* $::$ $('a\ operation) \Rightarrow ('a\ state) \rightharpoonup ('a\ state)$ **where**
delta-gset-op a as $=$ *Some* $(\ as \cup \{\ a\ \})$

δ-state G-Set

type-synonym $('a)$ *state* $=$ $'a$ *set*
type-synonym $('a)$ *operation* $=$ $'a$

fun *insert* $::$ $'a \Rightarrow ('a\ state) \Rightarrow ('a\ operation)$ **where**
insert a - = a

fun *delta-gset-op* $::$ $('a\ operation) \Rightarrow ('a\ state) \rightharpoonup ('a\ state)$ **where**
delta-gset-op a as $=$ *Some* $(\ as \cup \{\ a\ \})$

Future work

1. Pair type locales; parameterize a proof that combinations of CRDTs are SEC.
    1. Immediately: PN-Counter.
    2. Immediately: 2P-Set.
2. Pure δ-state encodings.
    1. Anti-entropy algorithms [Almeida et al., 2018].
    2. No delivery precondition.
3. Proofs of causally consistent δ-state CRDTs [Almeida et al., 2018]:
    1. *δ-interval*:
    $$\Delta_i^{a,b} = \bigsqcup \{d_i^k : k \in [a, b)\}$$
    2. Causal merging condition: Replica i only joins a δ-interval $\Delta_j^{a,b}$ into its own state X_i if: $X_i \sqsupseteq X_j^a$

Future work

1. Pair type locales; parameterize a proof that combinations of CRDTs are SEC.
   1. Immediately: PN-Counter.
   2. Immediately: 2P-Set.
2. Pure δ-state encodings.
   1. Anti-entropy algorithms [Almeida et al., 2018].
   2. No delivery precondition.
3. Proofs of causally consistent δ-state CRDTs [Almeida et al., 2018]:
   1. *δ-interval:*

$$\Delta_i^{a,b} = \bigsqcup \{d_i^k : k \in [a, b)\}$$

   2. Causal merging condition: Replica i only joins a δ-interval $\Delta_j^{a,b}$ into its own state X_i if: $X_i \sqsupseteq X_j^a$

Future work

1. Pair type locales; parameterize a proof that combinations of CRDTs are SEC.
    1. Immediately: PN-Counter.
    2. Immediately: 2P-Set.
2. Pure δ-state encodings.
    1. Anti-entropy algorithms [Almeida et al., 2018].
    2. No delivery precondition.
3. Proofs of causally consistent δ-state CRDTs [Almeida et al., 2018]:
    1. *δ-interval*:

$$\Delta_i^{a,b} = \bigsqcup \{ d_i^k : k \in [a, b) \}$$

    2. Causal merging condition: Replica i only joins a δ-interval $\Delta_j^{a,b}$ into its own state X_i if: $X_i \sqsupseteq X_j^a$

Future work

1. Pair type locales; parameterize a proof that combinations of CRDTs are SEC.
   1. Immediately: PN-Counter.
   2. Immediately: 2P-Set.
2. Pure δ-state encodings.
   1. Anti-entropy algorithms [Almeida et al., 2018].
   2. No delivery precondition.
3. Proofs of causally consistent δ-state CRDTs [Almeida et al., 2018]:
   1. *δ-interval:*

$$\Delta_i^{a,b} = \bigsqcup \{d_i^k : k \in [a, b)\}$$

   2. Causal merging condition: Replica i only joins a δ-interval $\Delta_j^{a,b}$ into its own state X_i if: $X_i \sqsupseteq X_j^a$

Future work

1. Pair type locales; parameterize a proof that combinations of CRDTs are SEC.
   1. Immediately: PN-Counter.
   2. Immediately: 2P-Set.
2. Pure δ-state encodings.
   1. Anti-entropy algorithms [Almeida et al., 2018].
   2. No delivery precondition.
3. Proofs of causally consistent δ-state CRDTs [Almeida et al., 2018]:
   1. *δ-interval*:

      $$\Delta_i^{a,b} = \bigsqcup \{d_i^k : k \in [a, b)\}$$

   2. Causal merging condition: Replica i only joins a δ-interval $\Delta_j^{a,b}$ into its own state X_i if: $X_i \sqsupseteq X_j^a$

Future work

1. Pair type locales; parameterize a proof that combinations of CRDTs are SEC.
    1. Immediately: PN-Counter.
    2. Immediately: 2P-Set.
2. Pure δ-state encodings.
    1. Anti-entropy algorithms [Almeida et al., 2018].
    2. No delivery precondition.
3. Proofs of causally consistent δ-state CRDTs [Almeida et al., 2018]:
    1. *δ-interval:*

$$\Delta_i^{a,b} = \bigsqcup \{ d_i^k : k \in [a, b) \}$$

    2. Causal merging condition: Replica i only joins a δ-interval $\Delta_j^{a,b}$ into its own state X_i if: $X_i \sqsupseteq X_j^a$

Future work

1. Pair type locales; parameterize a proof that combinations of CRDTs are SEC.
   1. Immediately: PN-Counter.
   2. Immediately: 2P-Set.
2. Pure δ-state encodings.
   1. Anti-entropy algorithms [Almeida et al., 2018].
   2. No delivery precondition.
3. Proofs of causally consistent δ-state CRDTs [Almeida et al., 2018]:
   1. *δ-interval:*
   $$\Delta_i^{a,b} = \bigsqcup \{d_i^k : k \in [a, b)\}$$
   2. Causal merging condition: Replica i only joins a δ-interval $\Delta_j^{a,b}$ into its own state X_i if: $X_i \sqsupseteq X_j^a$

Future work

1. Pair type locales; parameterize a proof that combinations of CRDTs are SEC.
   1. Immediately: PN-Counter.
   2. Immediately: 2P-Set.
2. Pure δ-state encodings.
   1. Anti-entropy algorithms [Almeida et al., 2018].
   2. No delivery precondition.
3. Proofs of causally consistent δ-state CRDTs [Almeida et al., 2018]:
   1. *δ-interval*:
   $$\Delta_i^{a,b} = \bigsqcup \{d_i^k : k \in [a, b)\}$$
   2. Causal merging condition: Replica i only joins a δ-interval $\Delta_j^{a,b}$ into its own state X_i if: $X_i \sqsupseteq X_j^a$

Future work

1. Pair type locales; parameterize a proof that combinations of CRDTs are SEC.
    1. Immediately: PN-Counter.
    2. Immediately: 2P-Set.
2. Pure δ-state encodings.
    1. Anti-entropy algorithms [Almeida et al., 2018].
    2. No delivery precondition.
3. Proofs of causally consistent δ-state CRDTs [Almeida et al., 2018]:
    1. *δ-interval*:
    $$\Delta_i^{a,b} = \bigsqcup \{ d_i^k : k \in [a, b) \}$$
    2. Causal merging condition: Replica i only joins a δ-interval $\Delta_j^{a,b}$ into its own state X_i if: $X_i \sqsupseteq X_j^a$

Conclusion

1. Extended the work of Gomes et al. [2017] to mechanize that δ-state CRDTs [Almeida et al., 2018] are SEC.
2. Two reductions: $\phi_{\text{state}\to\text{op}}$ and $\phi_{\delta\to\text{op}}$.
3. Network relaxations to allow duplication of messages.
4. Mechanized proof that two δ-state CRDTs (G-Counter, G-Set) are SEC.

Conclusion

1. Extended the work of Gomes et al. [2017] to mechanize that δ-state CRDTs [Almeida et al., 2018] are SEC.
2. Two reductions: $\phi_{\text{state}\rightarrow\text{op}}$ and $\phi_{\delta\rightarrow\text{op}}$.
3. Network relaxations to allow duplication of messages.
4. Mechanized proof that two δ-state CRDTs (G-Counter, G-Set) are SEC.

Conclusion

1. Extended the work of Gomes et al. [2017] to mechanize that δ-state CRDTs [Almeida et al., 2018] are SEC.
2. Two reductions: $\phi_{\mathsf{state}\to\mathsf{op}}$ and $\phi_{\delta\to\mathsf{op}}$.
3. Network relaxations to allow duplication of messages.
4. Mechanized proof that two δ-state CRDTs (G-Counter, G-Set) are SEC.

Conclusion

1. Extended the work of Gomes et al. [2017] to mechanize that δ-state CRDTs [Almeida et al., 2018] are SEC.
2. Two reductions: $\phi_{\text{state}\to\text{op}}$ and $\phi_{\delta\to\text{op}}$.
3. Network relaxations to allow duplication of messages.
4. Mechanized proof that two δ-state CRDTs (G-Counter, G-Set) are SEC.

Thank you!

Questions?

Thank you!
Questions?

P. S. Almeida, A. Shoker, and C. Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, Jan 2018. ISSN 0743-7315. doi: 10.1016/j.jpdc.2017.08.003. URL http://dx.doi.org/10.1016/j.jpdc.2017.08.003.

V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. Verifying strong eventual consistency in distributed systems. *CoRR*, abs/1707.01747, 2017. URL http://arxiv.org/abs/1707.01747.

H. Howard and R. Mortier. Paxos vs raft. *Proceedings of the 7thWorkshop on Principles and Practice of Consistency for Distributed Data*, Apr 2020. doi: 10.1145/3380787.3393681. URL http://dx.doi.org/10.1145/3380787.3393681.

L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL https://doi.org/10.1145/279227.279229.

D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. Research Report RR-7687, July 2011. URL https://hal.inria.fr/inria-00609399.

J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *PLDI 2015: Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation*, pages 357–368, Portland, OR, USA, June 2015.

D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, page 154–165, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341271. doi: 10.1145/2854065.2854081. URL https://doi.org/10.1145/2854065.2854081.