A framework for establishing Strong Eventual Consistency for Conflict-free Replicated Data types

Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, Alastair R. Beresford

October 27, 2022

Abstract

In this work, we focus on the correctness of Conflict-free Replicated Data Types (CRDTs), a class of algorithm that provides strong eventual consistency guarantees for replicated data. We develop a modular and reusable framework for verifying the correctness of CRDT algorithms. We avoid correctness issues that have dogged previous mechanised proofs in this area by including a network model in our formalisation, and proving that our theorems hold in all possible network behaviours. Our axiomatic network model is a standard abstraction that accurately reflects the behaviour of real-world computer networks. Moreover, we identify an abstract convergence theorem, a property of order relations, which provides a formal definition of strong eventual consistency. We then obtain the first machine-checked correctness theorems for three concrete CRDTs: the Replicated Growable Array, the Observed-Remove Set, and an Increment-Decrement Counter.

Contents

1 Introduction 2
2 Technical Lemmas 2
  2.1 Kleisli arrow composition 3
  2.2 Lemmas about sets 3
  2.3 Lemmas about list 3
3 Strong Eventual Consistency 6
  3.1 Concurrent operations 6
  3.2 Happens-before consistency 7
  3.3 Apply operations 9
  3.4 Concurrent operations commute 9
  3.5 Abstract convergence theorem 10
  3.6 Convergence and progress 11
4 Axiomatic network models 12
  4.1 Node histories 12
  4.2 Asynchronous broadcast networks 15
  4.3 Causal networks 16
  4.4 Dummy network models 22
5 Replicated Growable Array 22
  5.1 Insert and delete operations 22
  5.2 Well-definedness of insert and delete 23
  5.3 Preservation of element indices 24
1 Introduction

Strong eventual consistency (SEC) is a model that strikes a compromise between strong and eventual consistency [12]. Informally, it guarantees that whenever two nodes have received the same set of messages—possibly in a different order—their view of the shared state is identical, and any conflicting concurrent updates must be merged automatically. Large-scale deployments of SEC algorithms include datacentre-based applications using the Riak distributed database [3], and collaborative editing applications such as Google Docs [5]. Unlike strong consistency models, it is possible to implement SEC in decentralised settings without any central server or leader, and it allows local execution at each node to proceed without waiting for communication with other nodes. However, algorithms for achieving decentralised SEC are currently poorly understood: several such algorithms, published in peer-reviewed venues, were subsequently shown to violate their supposed guarantees [6, 7, 9]. Informal reasoning has repeatedly produced plausible-looking but incorrect algorithms, and there have even been examples of mechanised formal proofs of SEC algorithm correctness later being shown to be flawed. These mechanised proofs failed because, in formalising the algorithm, they made false assumptions about the execution environment.

In this work we use the Isabelle/HOL proof assistant [13] to create a framework for reliably reasoning about the correctness of a particular class of decentralised replication algorithms. We do this by formalising not only the replication algorithms, but also the network in which they execute, allowing us to prove that the algorithm’s assumptions hold in all possible network behaviours. We model the network using the axioms of asynchronous unreliable causal broadcast, a well-understood abstraction that is commonly implemented by network protocols, and which can run on almost any computer network, including large-scale networks that delay, reorder, or drop messages, and in which nodes may fail.

We then use this framework to produce machine-checked proofs of correctness for three Conflict-Free Replicated Data Types (CRDTs), a class of replication algorithms that ensure strong eventual consistency [11, 12]. To our knowledge, this is the first machine-checked verification of SEC algorithms that explicitly models the network and reasons about all possible network behaviours. The framework is modular and reusable, making it easy to formulate proofs for new algorithms. We provide the first mechanised proofs of the Replicated Growable Array, the operation-based Observed-Remove Set, and the operation-based counter CRDT.

2 Technical Lemmas

This section contains a list of helper definitions and lemmas about sets, lists and the option monad.

theory Util
imports Main
HOL-Library.Monad-Syntax
2.1 Kleisli arrow composition

**Definition)** \( \text{kleisli} :: (\'b \Rightarrow \text{option}) \Rightarrow (\'b \Rightarrow \text{option}) \Rightarrow (\'b \Rightarrow \text{option}) \) (\text{infixr} \ 65) where \( f \triangleright g \equiv \lambda x. (f x \gg (\lambda y. g y)) \)

**Lemma** \( \text{kleisli-comm-cong} \):
- Assumptions: \( x \triangleright y = y \triangleright x \)
- Shows: \( z \triangleright x \triangleright y = z \triangleright y \triangleright x \)
- Using: \( \text{assms} \ \text{by} (\text{clarsimp simp add: kleisli-def}) \)

**Lemma** \( \text{kleisli-assoc} \):
- Shows: \( (z \triangleright x) \triangleright y = z \triangleright (x \triangleright y) \)
- By: \( (\text{auto simp add: kleisli-def}) \)

2.2 Lemmas about sets

**Lemma** \( \text{distinct-set-notin} [\text{dest}] \):
- Assumptions: \( \text{distinct} (x \# xs) \)
- Shows: \( x \notin \text{set xs} \)
- Using: \( \text{assms} \ \text{by} (\text{induction xs, auto}) \)

**Lemma** \( \text{set-membership-equality-technicalD} [\text{dest}] \):
- Assumptions: \( \{x\} \cup (\text{set xs}) = \{y\} \cup (\text{set ys}) \)
- Shows: \( x = y \lor y \in \text{set xs} \)
- Using: \( \text{assms} \ \text{by} (\text{induction xs, auto}) \)

**Lemma** \( \text{set-equality-technical} \):
- Assumptions: \( \{x\} \cup (\text{set xs}) = \{y\} \cup (\text{set ys}) \)
  - and \( x \notin \text{set xs} \)
  - and \( y \notin \text{set ys} \)
  - and \( y \in \text{set xs} \)
- Shows: \( \{x\} \cup (\text{set xs} - \{y\}) = \text{set ys} \)
- Using: \( \text{assms} \ \text{by} (\text{induction xs, auto}) \)

**Lemma** \( \text{set-elem-nth} \):
- Assumptions: \( x \in \text{set xs} \)
- Shows: \( \exists m. m < \text{length xs} \land x \in \text{xs} ! m = x \)
- Using: \( \text{assms} \ \text{by} (\text{induction xs, simp}) \ (\text{meson in-set-conv-nth}) \)

2.3 Lemmas about list

**Lemma** \( \text{list-nil-or-snoc} \):
- Shows: \( \text{xs} = [] \lor (\exists y \ y \in \text{xs} = y \# [y]) \)
- By: \( (\text{induction xs, auto}) \)

**Lemma** \( \text{suffix-eq-distinct-list} \):
- Assumptions: \( \text{distinct xs} \)
  - and \( \text{ys@sufl} = \text{xs} \)
  - and \( \text{ys@sufl} = \text{xs} \)
- Shows: \( \text{sufl} = \text{sufl} \)
- Using: \( \text{assms} \ \text{by} (\text{induction xs arbitrary: sufl sufl rule: rev-induct, simp}) \ (\text{metis append-eq-append-cone}) \)

**Lemma** \( \text{pre-suf-eq-distinct-list} \):
- Assumptions: \( \text{distinct xs} \)
  - and \( \text{ys} \neq [] \)
  - and \( \text{pre1@ys@sufl} = \text{xs} \)
and \( \text{pre2@ys@suf2} = xs \)
shows \( \text{pre1} = \text{pre2} \land \text{suf1} = \text{suf2} \)

using assms
apply(induction xs arbitrary: \( \text{pre1} \) \( \text{pre2} \) \( \text{ys} \), simp)
apply(case-tac \( \text{pre1} \); case-tac \( \text{pre2} \); clarify)
apply(metis suffix-eq-distinct-list append-Nil)
apply(metis Un-iff append-eq-Cons-conv distinct.simps(2) list.set-intros(1) set-append suffix-eq-distinct-list)
apply(metis Un-iff append-eq-Cons-conv distinct.simps(2) list.set-intros(1) set-append suffix-eq-distinct-list)
apply(metis distinct.simps(2) hd-append\(2 \) list.sel\(1 \) list.sel\(3 \) list.simps(3) tl-append\(2 \))
done

lemma list-head-unaffected:
assumes \( \text{hd} (x \# [y, z]) = v \)
shows \( \text{hd} (x \# [y, \ ]) = v \)
using assms by (metis \( \text{hd-append list.sel}(1) \))

lemma list-head-butlast:
assumes \( \text{hd} \text{xs} = v \)
and \( \text{length} \text{xs} > 1 \)
shows \( \text{hd} (\text{butlast} \text{xs}) = v \)
using assms by (metis \( \text{hd-conv-nth length-butlast length-greater-0-conv less-trans nth-butlast zero-less-diff zero-less-one} \))

lemma list-head-length-one:
assumes \( \text{hd} \text{xs} = x \)
and \( \text{length} \text{xs} = 1 \)
shows \( \text{xs} = [x] \)
using assms by(metis One-nat-def Suc-length-conv \( \text{hd-tl length-0-conv list.sel}(3) \))

lemma list-two-at-end:
assumes \( \text{length} \text{xs} > 1 \)
shows \( \exists x\ ' y . \text{xs} = x\ ' \# [x, y] \)
using assms
apply(induction \( \text{xs} \) rule: rev-induct, simp)
apply(case-tac \( \text{length} \text{xs} = 1 \), simp)
apply(metis append-self-conv2 length-0-conv length-Suc-conv)
apply(rule-tac \( x=\text{butlast} \text{xs} \) in exI, rule-tac \( x=\text{last} \text{xs} \) in exI, simp)
done

lemma list-nth-split-technical:
assumes \( m < \text{length} cs \)
and \( cs \neq [] \)
shows \( \exists x\ ' y . cs = x\ ' \# ([cs]m)\# ys \)
using assms
apply(induction \( \text{ys} \) arbitrary: \( cs \))
apply(meson in-set-conv-decomp nth-mem)
apply(metis in-set-conv-decomp length-list-update set-swap set-update-memI)
done

lemma list-nth-split:
assumes \( m < \text{length} cs \)
and \( n < m \)
and \( 1 < \text{length} cs \)
shows \( \exists x\ ' y . cs = x\ ' ([cs]n)\# ys\@([cs]m)\# zs \)
using assms proof(induction \( n \) arbitrary: \( cs \) \( m \))
case 0 thus \( ?case \)
apply(case-tac \( cs \); clarsimp)
apply(rule-tac \( x=[] \) in exI, clarsimp)
apply (rule list-nth-split-technical, simp, force)
done

next
case (Suc n)
thus ?case
proof (cases cs)
case Nil
then show ?thesis
  using Suc.prems by auto
next
case (Cons a as)
hence m − 1 < length as ∧ n < m − 1
  using Suc by force+
then obtain xs ys zs where as = xs @! as ∧ n ≠ ys @! as ∧ (m − 1) ≠ zs
  using Suc by force
thus ?thesis
apply (rule-tac x = a @! xs in exI)
using Suc Cons apply force
done
qed

lemma list-split-two-elems:
assumes distinct cs
  and x ∈ set cs
  and y ∈ set cs
  and x ≠ y
shows ∃ pre mid suf. cs = pre @! x @! mid @! y @! suf ∨ cs = pre @! y @! mid @! x @! suf
proof
  obtain xi yi where *: xi < length cs ∧ x = cs ! xi yi < length cs ∧ y = cs ! yi xi ≠ yi
    using set-elem-nth linorder-neqE-nat assms by metis
thus ?thesis
  by (metis list-nth-split One-nat-def less-Suc-eq linorder-neqE-nat not-less-zero)
qed

lemma split-list-unique-prefix:
assumes x ∈ set xs
shows ∃ pre suf. xs = pre @! x @! suf ∧ (∀ y ∈ set pre. x ≠ y)
using assms proof (induction xs)
case Nil thus ?case by clarsimp
next
case (Cons y ys)
then show ?case
proof (cases y = x)
case True
then show ?thesis by force
next
case False
then obtain pre suf where ys = pre @! x @! suf ∧ (∀ y ∈ set pre. x ≠ y)
  using assms Cons by auto
thus ?thesis
  using split-list-first by force
qed

lemma map-filter-append:
shows List.map-filter P (xs @! ys) = List.map-filter P xs @! List.map-filter P ys
by (auto simp add: List.map-filter-def)
3 Strong Eventual Consistency

In this section we formalise the notion of strong eventual consistency. We do not make any assumptions about networks or data structures; instead, we use an abstract model of operations that may be reordered, and we reason about the properties that those operations must satisfy. We then provide concrete implementations of that abstract model in later sections.

theory
Convergence
imports
Util
begin

The happens-before relation, as introduced by [8], captures causal dependencies between operations. It can be defined in terms of sending and receiving messages on a network. However, for now, we keep it abstract, our only restriction on the happens-before relation is that it must be a strict partial order, that is, it must be irreflexive and transitive, which implies that it is also antisymmetric. We describe the state of a node using an abstract type variable. To model state changes, we assume the existence of an interpretation function interp which lifts an operation into a state transformer—a function that either maps an old state to a new state, or fails.

locale happens-before = preorder hb-weak hb
  for hb-weak :: 'a ⇒ 'a ⇒ bool (infix ≤ 50)
  and hb :: 'a ⇒ 'a ⇒ bool (infix < 50) +
  fixes interp :: 'a ⇒ 'b ⇒ 'b (⟨⟩ [0] 1000)
begin

3.1 Concurrent operations

We say that two operations x and y are concurrent, written \( x \parallel y \), whenever one does not happen before the other: \( \neg (x \prec y) \) and \( \neg (y \prec x) \).

definition concurrent :: 'a ⇒ 'a ⇒ bool (infix || 50) where
  s₁ || s₂ ≡ \neg (s₁ ⪯ s₂) ∧ \neg (s₂ ⪯ s₁)

lemma concurrentI [intro!]: \neg (s₁ ⪯ s₂) \Longrightarrow \neg (s₂ ⪯ s₁) \Longrightarrow s₁ || s₂
  by (auto simp: concurrent-def)

lemma concurrentD1 [dest]: s₁ || s₂ \Longrightarrow \neg (s₂ ⪯ s₁)
  by (auto simp: concurrent-def)

lemma concurrentD2 [dest]: s₁ || s₂ \Longrightarrow \neg (s₁ ⪯ s₂)
  by (auto simp: concurrent-def)

lemma concurrent-refl [intro!, simp]: s || s
  by (auto simp: concurrent-def)

lemma concurrent-comm: s₁ || s₂ ⇔ s₂ || s₁
  by (auto simp: concurrent-def)

definition concurrent-set :: 'a ⇒ 'a list ⇒ bool where
  concurrent-set x xs ≡ \forall y ∈ set xs. x || y

lemma concurrent-set-empty [simp, intro!]:
  concurrent-set x []
by (auto simp: concurrent-set-def)

lemma concurrent-set-ConsE [elim!]:
  assumes concurrent-set a (x#xs)
  and concurrent-set a xs \Rightarrow concurrent x a \Rightarrow G
  shows G
  using assms by (auto simp: concurrent-set-def)

lemma concurrent-set-ConsI [intro!]:
  concurrent-set a xs \Rightarrow concurrent a x \Rightarrow concurrent-set a (x#xs)
  by (auto simp: concurrent-set-def)

lemma concurrent-set-appendI [intro!]:
  concurrent-set a xs \Rightarrow concurrent-set a ys \Rightarrow concurrent-set a (xs@ys)
  by (auto simp: concurrent-set-def)

lemma concurrent-set-Cons-Snoc [simp]:
  concurrent-set a (xs@[x]) = concurrent-set a (x#xs)
  by (auto simp: concurrent-set-def)

3.2 Happens-before consistency

The purpose of the happens-before relation is to require that some operations must be applied in a particular order, while allowing concurrent operations to be reordered with respect to each other. We assume that each node applies operations in some sequential order (a standard assumption for distributed algorithms), and so we can model the execution history of a node as a list of operations.

inductive hb-consistent :: 'a list \Rightarrow bool where
  [intro!]: hb-consistent []
  [intro!]: \[ hb-consistent xs; \forall x \in set xs. \neg y \prec x \] \Rightarrow hb-consistent (xs@[y])

As a result, whenever two operations x and y appear in a hb-consistent list, and x \prec y, then x must appear before y in the list. However, if x \parallel y, the operations can appear in the list in either order.

lemma (x \prec y \lor concurrent x y) = (\neg y \prec x)
  using less-asym by blast

lemma consistentI [intro!]:
  assumes hb-consistent (xs@ys)
  and \forall x \in set (xs@ys). \neg z \prec x
  shows hb-consistent (xs@ys@[z])
  using assms hb-consistent.intros append-assoc by metis

inductive-cases hb-consistent-elim [elim]:
  hb-consistent []
  hb-consistent (xs@[y])
  hb-consistent (xs@ys)
  hb-consistent (xs@ys@[z])

inductive-cases hb-consistent-elim-gen:
  hb-consistent zs

lemma hb-consistent-append-D1 [dest]:
  assumes hb-consistent (xs@ys)
  shows hb-consistent xs
  using assms by (induction ys arbitrary; xs rule: List.rev-induct) auto
lemma hb-consistent-append-D2 [dest]:
assumes hb-consistent (xs @ ys)
shows hb-consistent ys
using assms by (induction ys arbitrary; xs rule: List.rev-induct) fastforce+

lemma hb-consistent-append-elim-ConsD [elim]:
assumes hb-consistent (y # ys)
shows hb-consistent ys
using assms hb-consistent-append-D2 by (metis append-Cons append-Nil)

lemma hb-consistent-remove1 [intro]:
assumes hb-consistent xs
shows hb-consistent (remove1 x xs)
using assms by (induction rule: hb-consistent.induct, simp) (auto simp: remove1-append)

lemma hb-consistent-singleton [intro]:
shows hb-consistent [x]
using hb-consistent.intros by fastforce

lemma hb-consistent-prefix-suffix-exists:
assumes hb-consistent ys
hb-consistent (xs @ [x])
{x} ∪ set xs = set ys
distinct (x # xs)
distinct ys
shows ∃ prefix suffix. ys = prefix @ x # suffix ∧ concurrent-set x suffix
using assms proof (induction arbitrary; xs rule: hb-consistent.induct, simp)
fix xs y ys
assume IH: (∀xs. hb-consistent (xs @ [x]) →
{x} ∪ set xs = set ys →
distinct (x # xs) → distinct ys →
∃ prefix suffix. ys = prefix @ x # suffix ∧ concurrent-set x suffix)
assume assms: hb-consistent ys ∀x∈set ys. ¬ hb y x
hb-consistent (xs @ [x])
{x} ∪ set xs = set (ys @ [y])
distinct (x # xs) distinct (ys @ [y])

hence x = y ∨ y ∈ set xs
using assms by auto

moreover {
assume x = y
hence ∃ prefix suffix. ys @ [y] = prefix @ x # suffix ∧ concurrent-set x suffix
by force
}

moreover {
assume y-in-xs: y ∈ set xs
hence {x} ∪ (set xs − {y}) = set ys
using assms by (auto intro: set-equality-technical)

hence remove-g-in-xs: {x} ∪ set (remove1 y xs) = set ys
using assms by auto

moreover have hb-consistent ((remove1 y xs) @ [x])
using assms hb-consistent-remove1 by force

moreover have distinct (x # (remove1 y xs))
using assms by simp

moreover have distinct ys
using assms by simp

ultimately obtain prefix suffix where ys-split: ys = prefix @ x # suffix ∧ concurrent-set x suffix
using IH by force

moreover {
have concurrent x y
using assms y-in-xs remove-y-in-xs concurrent-def by blast
hence concurrent-set x (suffix@y)
using ys-split by clarsimp
}
ultimately have \exists prefix suffix. ys @ [y] = prefix @ x # suffix \land concurrent-set x suffix
by force
}
ultimately show \exists prefix suffix. ys @ [y] = prefix @ x # suffix \land concurrent-set x suffix
by auto
qed

lemma hb-consistent-append [intro!]:
assumes hb-consistent suffix
hb-consistent prefix
\forall s p. s \in set suffix \implies p \in set prefix \implies \neg s \prec p
shows hb-consistent (prefix @ suffix)
using assms by (induction rule: hb-consistent.induct) force+

lemma hb-consistent-append-porder:
assumes hb-consistent (xs @ ys)
x \in set xs
y \in set ys
shows \neg y \prec x
using assms by (induction ys arbitrary: xs rule: rev-induct) force+

3.3 Apply operations

We can now define a function apply-operations that composes an arbitrary list of operations into a state transformer. We first map interp across the list to obtain a state transformer for each operation, and then collectively compose them using the Kleisli arrow composition combinator.

definition apply-operations :: 'a list \Rightarrow 'b \Rightarrow 'b
where
apply-operations es ≡ foldl (\⊿) Some (map interp es)

lemma apply-operations-empty [simp]: apply-operations [] s = Some s
by(auto simp: apply-operations-def)

lemma apply-operations-Snoc [simp]:
apply-operations (xs@[x]) = (apply-operations xs) \triangleright (x)
by(auto simp add: apply-operations-def kleisli-def)

3.4 Concurrent operations commute

We say that two operations x and y commute whenever \langle x \rangle \triangleright \langle y \rangle = \langle y \rangle \triangleright \langle x \rangle, i.e. when we can swap the order of the composition of their interpretations without changing the resulting state transformer. For our purposes, requiring that this property holds for all pairs of operations is too strong. Rather, the commutation property is only required to hold for operations that are concurrent.

definition concurrent-ops-commute :: 'a list \Rightarrow bool
where
concurrent-ops-commute xs ≡
\forall x y. \{x, y\} \subseteq set xs \implies concurrent x y \implies \langle x \rangle \triangleright \langle y \rangle = \langle y \rangle \triangleright \langle x \rangle

lemma concurrent-ops-commute-empty [intro!]: concurrent-ops-commute []
by(auto simp: concurrent-ops-commute-def)

lemma concurrent-ops-commute-singleton [intro!]: concurrent-ops-commute [x]
by (auto simp: concurrent-ops-commute-def)

lemma concurrent-ops-commute-appendD [dest]:
  assumes concurrent-ops-commute (xs@ys)
  shows concurrent-ops-commute xs
using assms by (auto simp: concurrent-ops-commute-def)

lemma concurrent-ops-commute-rearrange:
  assumes concurrent-ops-commute (xs@x#ys) = concurrent-ops-commute (xs@ys@x[x])
by (clarsimp simp: concurrent-ops-commute-def)

lemma concurrent-ops-commute-concurrent-set:
  assumes concurrent-ops-commute (prefix@suffix@x[x])
  shows apply-operations prefix @ suffix = apply-operations prefix @ x @ suffix @ prefix
using assms proof (induction prefix @ suffix arbitrary: rule: rev-induct, force)
fix xs
assume IH: concurrent-ops-commute (prefix @ xs @ x[x])
  concurrently x zs = dist (prefix @ x # zs) =
  apply-operations (prefix @ x # zs) =
show IH assms: concurrent-ops-commute (prefix @ xs @ [a])
  distinct (prefix @ x # xs @ [a])
hence ac-comm: (a) @ (x) = (x) @ (a)
  by (clarsimp simp: concurrent-ops-commute-def) blast
have copc: concurrent-ops-commute (prefix @ xs @ [x])
using assms by (clarsimp simp: concurrent-ops-commute-def) blast
have apply-operations ((prefix @ x # xs) @ [a]) = (apply-operations (prefix @ x # xs)) @ (a)
  by (simp del: append-assoc)
also have ... = (apply-operations (prefix @ xs @ [x])) @ (a)
  using IH assms copc by auto
also have ... = ((apply-operations (prefix @ xs))) @ (x) @ (a)
  by (simp add: append-assoc[symmetric] del: append-assoc)
also have ... = (apply-operations (prefix @ xs)) @ ((a) @ (x))
  using ac-comm kleisli-comm-cong kleisli-assoc by simp
finally show apply-operations (prefix @ xs @ [a]) @ [x] = apply-operations (prefix @ x # xs @ [a])
  by (metis Cons-eq-appendI append-assoc apply-operations-Snoc kleisli-assoc)
qed

3.5 Abstract convergence theorem

We can now state and prove our main theorem, convergence. This theorem states that two
hb-consistent lists of distinct operations, which are permutations of each other and in which
concurrent operations commute, have the same interpretation.

theorem convergence:
  assumes set xs = set ys
  concurrent-ops-commute xs
  distinct xs
  concurrent-ops-commute ys
  distinct ys
  hb-consistent xs
  hb-consistent ys
  apply-operations xs = apply-operations ys
using assms proof (induction xs arbitrary: ys rule: rev-induct, simp)
  case assms: (snoc x xs)
  then obtain prefix suffix where ys-split: ys = prefix @ x # suffix @ concurrent-set x suffix
  using hb-consistent-prefix-suffix-exists by fastforce
moreover hence ∗: distinct (prefix @ suffix) hb-consistent xs
using assms by auto
moreover {
    have hb-consistent prefix hb-consistent suffix
    using ys-split assms hb-consistent-append-D2 hb-consistent-append-elim-ConsD by blast+
    hence hb-consistent (prefix @ suffix)
    by (metis assms(8) hb-consistent-append hb-consistent-append-porder list.set-intros(2) ys-split)
}
moreover have ∗∗: concurrent-ops-commute (prefix @ suffix [x])
using assms ys-split by (clarsimp simp: concurrent-ops-commute-def)
moreover hence concurrent-ops-commute (prefix @ suffix)
by (force simp del: append-assoc simp add: append-assoc[symmetric])
ultimately have apply-operations xs = apply-operations (prefix@suffix)
using assms by simp (metis Diff-insert-absorb Un-iff * concurrent-ops-commute-appendD set-append)
moreover have apply-operations (prefix@suffix @ [x]) = apply-operations (prefix@x # suffix)
using ys-split assms ∗∗ concurrent-ops-commute-concurrent-set by force
ultimately show ?case
using ys-split by (force simp: append-assoc[symmetric] simp del: append-assoc)
qed

corollary convergence-ext:
assumes set xs = set ys
 concurrent-ops-commute xs
 concurrent-ops-commute ys
 distinct xs
 distinct ys
 hb-consistent xs
 hb-consistent ys
shows apply-operations xs s = apply-operations ys s
using convergence assms by metis
end

3.6 Convergence and progress

Besides convergence, another required property of SEC is progress: if a valid operation was issued on one node, then applying that operation on other nodes must also succeed—that is, the execution must not become stuck in an error state. Although the type signature of the interpretation function allows operations to fail, we need to prove that in all hb-consistent network behaviours such failure never actually occurs. We capture the combined requirements in the strong-eventual-consistency locale, which extends happens-before.

locale strong-eventual-consistency = happens-before +
fixes op-history :: 'a list ⇒ bool
    and initial-state :: 'b
assumes causality: op-history xs ⇒ hb-consistent xs
assumes distinctness: op-history xs ⇒ distinct xs
assumes commutativity: op-history xs ⇒ concurrent-ops-commute xs
assumes no-failure: op-history(xs@[x]) ⇒ apply-operations xs initial-state = Some state ⇒ ⟨x⟩ state ≠ None
assumes trunc-history: op-history(xs@[x]) ⇒ op-history xs
begin

theorem sec-convergence:
assumes set xs = set ys
    op-history xs
    op-history ys
shows apply-operations xs = apply-operations ys
by (meson assms convergence causality commutativity distinctness)
4 Axiomatic network models

In this section we develop a formal definition of an asynchronous unreliable causal broadcast network. We choose this model because it satisfies the causal delivery requirements of many operation-based CRDTs [1, 2]. Moreover, it is suitable for use in decentralised settings, as motivated in the introduction, since it does not require waiting for communication with a central server or a quorum of nodes.

locale node-histories =
  fixes history :: nat ⇒ 'evt list
  assumes histories-distinct [intro!, simp]: distinct (history i)

lemma (in node-histories) history-finite:
  shows finite (set (history i))
  by auto

definition (in node-histories) history-order :: 'evt ⇒ nat ⇒ 'evt ⇒ bool (-/ ⊑/ - [50,1000,50]
where
  x ⊑ i z ≡ ∃ xs ys zs. xs@x#ys@z#zs = history i

lemma (in node-histories) node-total-order-trans:
  assumes e1 ⊑ i e2
      and e2 ⊑ i e3
  shows e1 ⊑ i e3
proof

obtain \(xs_1\) \(xs_2\) \(ys_1\) \(ys_2\) \(zs_1\) \(zs_2\) where \(\ast\) : \(xs_1 \@ e_1 \# y_1 \@ e_2 \# z_1 \) = \(\text{history } i\)
(\(xs_2 \@ e_2 \# y_2 \@ e_3 \# z_2\) = \(\text{history } i\))
using \(\text{history-order-def}\) \(\text{assms}\) by \(\text{auto}\)

hence \(xs_1 \@ e_1 \# y_1 \# zs_1 = \text{history } i\)

by \((\text{rule-tac } xs = \text{history } i\) \&\& \(ys = [c_2]\) \text{ in } \text{pre-suf-}eq\text{-}\text{distinct-list})\) \(\text{auto}\)

thus \(?\text{thesis}\)
by \((\text{clarsimp simp: } \text{history-order-def})\) \((\text{metis } \ast\) \((2)\) \(\text{append.assoc append-Cons})\)

qed

lemma (in \(\text{node-histories}\)) \(\text{local-order-carrier-closed}\):
assumes \(e_1 \sqsubseteq i\) \(e_2\)
shows \(\{e_1, e_2\} \subseteq \text{set } (\text{history } i)\)
using \(\text{assms}\) by \(\text{(clarsimp simp add: } \text{history-order-def})\)

(by \((\text{metis } \text{in-set-conv-decomp Un-iff } \text{Un-subset-iff} \text{ list.simps(15)}\) \(\text{set-append})+)\)

lemma (in \(\text{node-histories}\)) \(\text{node-total-order-irrefl}\):
shows \(\sim(e \sqsubseteq i)\)
by \((\text{clarsimp simp add: } \text{history-order-def})\)

(by \((\text{metis } \text{Un-iff } \text{histories-distinct } \text{distinct-append distinct-set-notin}\) \(\text{list.set-intros(1) set-append})+)\)

lemma (in \(\text{node-histories}\)) \(\text{node-total-order-antisym}\):
assumes \(e_1 \sqsubseteq i\) \(e_2\)
and \(e_2 \sqsubseteq i\)
shows \(\text{False}\)
using \(\text{assms}\) \(\text{node-total-order-irrefl}\) \(\text{node-total-order-trans}\) by \(\text{blast}\)

lemma (in \(\text{node-histories}\)) \(\text{node-order-is-total}\):
assumes \(e_1 \in \text{set } (\text{histories } i)\)
and \(e_2 \in \text{set } (\text{histories } i)\)
and \(e_1 \neq e_2\)
shows \(e_1 \sqsubseteq i\) \(e_2 \sqsubseteq i\) \(e_1\)
using \(\text{assms}\) \(\text{unfolding } \text{history-order-def}\) by \((\text{metis list-split-two-elems histories-distinct})\)

definition (in \(\text{node-histories}\)) \(\text{prefix-of-node-history} :: 'e\text{evt list } \Rightarrow \text{nat } \Rightarrow \text{bool}\) \((\text{infix } \text{prefix of } 50)\) where
\(xs\) \(\text{prefix of } i = \exists y_1. x_1 \# y_1 = \text{history } i\)

lemma (in \(\text{node-histories}\)) \(\text{carriers-head-lt}\):
assumes \(y \# y_1 = \text{history } i\)
shows \(\sim(x \sqsupset y)\)
using \(\text{assms}\)
apply \((\text{clarsimp simp add: } \text{history-order-def})\)
apply \((\text{rename-tac } xs_1 y_1 z_1)\)
apply \((\text{subgoal-tac } xs_1 \# x \# y_1 = [\] \&\& \(\text{zs}_1 = y_1\))\)
apply \(\text{clarsimp}\)
apply \((\text{rule-tac } xs = \text{history } i\) \&\& \(ys = [y]\) \text{ in } \text{pre-suf-}eq\text{-}\text{distinct-list})\)
apply \(\text{auto}\)
done

lemma (in \(\text{node-histories}\)) \(\text{prefix-of-ConsD}\) \[dest\] :
assumes \(x \# x_1 = \text{prefix of } i\)
shows \([x]\) \(\text{prefix of } i\)
using \(\text{assms}\) by \((\text{auto simp: } \text{prefix-of-node-history-def})\)

lemma (in \(\text{node-histories}\)) \(\text{prefix-of-appendD}\) \[dest\] :
assumes \(xs \@ y_1 = \text{prefix of } i\)

shows \( xs \) prefix of \( i \)
using assms by (auto simp: prefix-of-node-history-def)

lemma (in node-histories) prefix-distinct:
assumes \( xs \) prefix of \( i \)
shows \( \text{distinct } xs \)
using assms by (clarsimp simp: prefix-of-node-history-def) (metis histories-distinct distinct-append)

lemma (in node-histories) prefix-to-carriers [intro]:
assumes \( xs \) prefix of \( i \)
shows \( \text{set } xs \subseteq \text{set } \text{history } i \)
using assms by (clarsimp simp: prefix-of-node-history-def) (metis Un_iff set_append)

lemma (in node-histories) prefix-elem-to-carriers:
assumes \( xs \) prefix of \( i \) and \( x \in \text{set } xs \)
shows \( x \in \text{set } \text{history } i \)
using assms by (clarsimp simp: prefix-of-node-history-def) (metis Un_iff set_append)

lemma (in node-histories) local-order-prefix-closed:
assumes \( x \sqsubseteq i \) \( y \)
and \( xs \) prefix of \( i \)
and \( y \in \text{set } xs \)
shows \( x \in \text{set } xs \)
proof -
obtain \( ys \) where \( xs @ ys = \text{history } i \)
using assms by blast
moreover obtain \( as \) \( bs \) \( cs \) where \( as @ x # bs @ y # cs = \text{history } i \)
using assms by blast
moreover obtain \( \langle pre, suf \rangle \) where \( \ast: xs = pre @ y # suf \)
using assms by fastforce
ultimately have \( pre = as @ x # bs \land suf @ ys = cs \)
by (rule-tac \( xs=\text{history } i \) \( \text{and } ys=[y] \) in pre-suf-eq-distinct-list) auto
thus ?thesis
using assms \( \ast \) by clarsimp
qed

lemma (in node-histories) local-order-prefix-closed-last:
assumes \( x \sqsubseteq i \) \( y \)
and \( xs@[y] \) prefix of \( i \)
shows \( x \in \text{set } xs \)
proof -
have \( x \in \text{set } (xs@[y]) \)
using assms by (force dest: local-order-prefix-closed)
thus ?thesis
using assms by (force simp add: node-total-order-irrefl prefix-to-carriers)
qed

lemma (in node-histories) events-before-exist:
assumes \( x \in \text{set } \text{history } i \)
shows \( \exists \text{pre } \text{pre}@[x] \) prefix of \( i \)
proof -
have \( \exists \text{idx } \text{idx} < \text{length } \text{history } i \) \( \text{and } \text{history } i ! \text{idx} = x \)
using assms by (simp add: set-elem-nth)
thus ?thesis
by (metis append_take_drop_id take_Suc_conv_app_nth prefix-of-node-history-def)
qed
lemma (in node-histories) events-in-local-order:
  assumes pre @ [e2] prefix of i
  and e1 ∈ set pre
  shows e1 ⊑ e2
  using assms split-list unfolding history-order-def prefix-of-node-history-def by fastforce

4.2 Asynchronous broadcast networks

We define a new locale *network* containing three axioms that define how broadcast and deliver events may interact, with these axioms defining the properties of our network model.

```plaintext
datatype 'msg event
  = Broadcast 'msg
  | Deliver 'msg

locale network = node-histories history for history :: nat ⇒ 'msg event list +
fixes msg-id :: 'msg ⇒ 'msgid

assumes delivery-has-a-cause: [ Deliver m ∈ set (history i) ] ⇒ ∃j. Broadcast m ∈ set (history j)
and deliver-locally: [ Broadcast m ∈ set (history i) ] ⇒ Broadcast m ⊑ i Deliver m
and msg-id-unique: [ Broadcast m1 ∈ set (history i); Broadcast m2 ∈ set (history j); msg-id m1 = msg-id m2 ] ⇒ i = j ∧ m1 = m2
```

The axioms can be understood as follows:

delivery-has-a-cause: If some message *m* was delivered at some node, then there exists some node on which *m* was broadcast. With this axiom, we assert that messages are not created “out of thin air” by the network itself, and that the only source of messages are the nodes.

deliver-locally: If a node broadcasts some message *m*, then the same node must subsequently also deliver *m* to itself. Since *m* does not actually travel over the network, this local delivery is always possible, even if the network is interrupted. Local delivery may seem redundant, since the effect of the delivery could also be implemented by the broadcast event itself; however, it is standard practice in the description of broadcast protocols that the sender of a message also sends it to itself, since this property simplifies the definition of algorithms built on top of the broadcast abstraction [4].

msg-id-unique: We do not assume that the message type *msg* has any particular structure; we only assume the existence of a function *msg-id::'msg⇒'msgid* that maps every message to some globally unique identifier of type *msgid*. We assert this uniqueness by stating that if *m1* and *m2* are any two messages broadcast by any two nodes, and their *msgid*-s are the same, then they were in fact broadcast by the same node and the two messages are identical. In practice, these globally unique IDs can by implemented using unique node identifiers, sequence numbers or timestamps.

lemma (in network) broadcast-before-delivery:
  assumes Deliver m ∈ set (history i)
  shows ∃j. Broadcast m ⊑ i Deliver m
  using assms deliver-locally delivery-has-a-cause by blast

lemma (in network) broadcasts-unique:
  assumes i ≠ j
  and Broadcast m ∈ set (history i)
  shows Broadcast m Φ set (history j)
Based on the well-known definition by [8], we say that \( m_1 \prec m_2 \) if any of the following is true:

1. \( m_1 \) and \( m_2 \) were broadcast by the same node, and \( m_1 \) was broadcast before \( m_2 \).

2. The node that broadcast \( m_2 \) had delivered \( m_1 \) before it broadcast \( m_2 \).

3. There exists some operation \( m_3 \) such that \( m_1 \prec m_3 \) and \( m_3 \prec m_2 \).

\[
\text{inductive (in network) } hb :: \text{'msg ⇒ 'msg ⇒ bool where}
\]
\[
hb\text{-broadcast: [Broadcast } m_1 \rightleftharpoons \text{Broadcast } m_2 \text{ ]} \implies hb m_1 m_2
\]
\[
hb\text{-deliver: [Deliver } m_1 \rightleftharpoons \text{Broadcast } m_2 \text{ ]} \implies hb m_1 m_2
\]
\[
hb\text{-trans: [ } hb m_1 m_2; hb m_2 m_3 \text{ ]} \implies hb m_1 m_3
\]

\[
\text{inductive-cases (in network) } hb\text{-elim: } hb x y
\]

\[
\text{definition (in network) } weak\text{-hb :: 'msg ⇒ 'msg ⇒ bool where}
\]
\[
weak\text{-hb } m_1 m_2 \equiv hb m_1 m_2 \lor m_1 = m_2
\]

\[
\text{locale causal-network = network +}
\]
\[
\text{assumes causal-delivery: Deliver } m_2 \in \text{set (history } j \text{ ) } \implies hb m_1 m_2 \implies \text{Deliver } m_1 \rightleftharpoons \text{Deliver } m_2
\]

\[
\text{lemma (in causal-network) causal-broadcast:}
\]
\[
\text{assumes Deliver } m_2 \in \text{set (history } j \text{ )}
\]
\[
\text{and } \text{Deliver } m_1 \rightleftharpoons \text{Broadcast } m_2
\]
\[
\text{shows Deliver } m_1 \rightleftharpoons \text{Deliver } m_2
\]
\[
\text{using assms causal-delivery } hb\text{-intros(2) by blast}
\]

\[
\text{lemma (in network) } hb\text{-broadcast-exists1:}
\]
\[
\text{assumes } hb m_1 m_2
\]
\[
\text{shows } \exists i. \text{Broadcast } m_1 \in \text{set (history } i \text{ )}
\]
\[
\text{using assms}
\]
\[
\text{apply(induction rule: } hb\text{-induct)}
\]
\[
\text{apply(meson insert-subset node-histories.local-order-carrier-closed node-histories-axioms)}
\]
\[
\text{apply(meson delivery-has-a-cause insert-subset local-order-carrier-closed)}
\]
\[
\text{apply simp}
\]
\[
\text{done}
\]

\[
\text{lemma (in network) } hb\text{-broadcast-exists2:}
\]
\[
\text{assumes } hb m_1 m_2
\]
\[
\text{shows } \exists i. \text{Broadcast } m_2 \in \text{set (history } i \text{ )}
\]
\[
\text{using assms}
\]
\[
\text{apply(induction rule: } hb\text{-induct)}
\]
\[
\text{apply(meson insert-subset node-histories.local-order-carrier-closed node-histories-axioms)}
\]
\[
\text{apply(meson delivery-has-a-cause insert-subset local-order-carrier-closed)}
\]
\[
\text{apply simp}
\]
\[
\text{done}
\]

\[
\text{4.3 Causal networks}
\]

\[
\text{lemma (in causal-network) } hb\text{-has-a-reason:}
\]
\[
\text{assumes } hb m_1 m_2
\]
\[
\text{and } \text{Broadcast } m_2 \in \text{set (history } i \text{ )}
\]
\[
\text{shows } \text{Deliver } m_1 \in \text{set (history } i \text{ )} \lor \text{Broadcast } m_1 \in \text{set (history } i \text{ )}
\]
\[
\text{using assms apply (induction rule: } hb\text{-induct)}
\]
\[
\text{apply(meson insert-subset local-order-carrier-closed network.broadcasts-unique network-axioms)}
\]
\[
\text{apply(meson insert-subset local-order-carrier-closed network.broadcasts-unique network-axioms)}
\]
\[
\text{using } hb\text{-trans causal-delivery local-order-carrier-closed apply blast}
\]
done

lemma (in causal-network) hb-cross-node-delivery:
assumes hb m1 m2
and Broadcast m1 ∈ set (history i)
and Broadcast m2 ∈ set (history j)
and i ≠ j
shows Deliver m1 ∈ set (history j)
using assms
apply(induction rule: hb.induct)
  apply(metis broadcasts-unique insert-subset local-order-carrier-closed)
  apply(metis insert-subset local-order-carrier-closed network.broadcasts-unique network-axioms)
using broadcasts-unique hb.intros(3) hb-has-a-reason apply blast
done

lemma (in causal-network) hb-irrefl:
assumes hb m1 m2
shows m1 ≠ m2
using assms
proof
  induction rule: hb.induct
case (hb-broadcast m1 i m2) thus ?case
    using node-total-order-antisym by blast
next
case (hb-deliver m1 i m2) thus ?case
  by(meson causal-broadcast insert-subset local-order-carrier-closed node-total-order-irrefl)
next
case (hb-trans m1 m2 m3)
then obtain i j where Broadcast m3 ∈ set (history i) Broadcast m2 ∈ set (history j)
using hb-broadcast-exists2 by blast
then show ?case
  using assms hb-trans by (meson causal-network.causal-delivery causal-network-axioms
deliver-locally insert-subset network hb.intros(3) network-axioms
node-histories.local-order-carrier-closed assms hb-trans
node-histories-axioms node-total-order-irrefl)
qed

lemma (in causal-network) hb-broadcast-broadcast-order:
assumes hb m1 m2
and Broadcast m1 ∈ set (history i)
and Broadcast m2 ∈ set (history i)
shows Broadcast m1 ⊂ i Broadcast m2
using assms
proof
  induction rule: hb.induct
case (hb-broadcast m1 i m2) thus ?case
  by(metis insertI1 local-order-carrier-closed network.broadcasts-unique network-axioms subsetCE)
next
case (hb-deliver m1 i m2) thus ?case
  by(metis broadcasts-unique insert-subset local-order-carrier-closed
network.broadcast-before-delivery network-axioms node-total-order-trans)
next
case (hb-trans m1 m2 m3)
then show ?case
proof
  cases Broadcast m2 ∈ set (history i)
  case True thus ?thesis
    using hb-trans node-total-order-trans by blast
next
  case False hence Deliver m2 ∈ set (history i) m1 ≠ m2 m2 ≠ m3
  using hb-has-a-reason hb-trans by auto
  thus ?thesis
  by(metis hb-trans event.inject(1) hb.intros(1) hb-irrefl network hb.intros(3) network-axioms node-order-is-total)
lemma (in causal-network) hb-antisym:
  assumes hb x y and hb y x
  shows False
using assms proof (induction rule: hb.induct)
fix m1 i m2
assume hb m2 m1 and Broadcast m1 ⊑ i Broadcast m2
thus False
  apply − proof (erale hb-elim)
  show ∃ ia. Broadcast m1 ⊑ i Broadcast m2 ⇒ Broadcast m2 ⊑ i a Broadcast m1 ⇒ False
    by (metis broadcasts-unique insert-subset local-order-carrier-closed node-total-order-irrefl node-total-order-trans)
next
  show ∃ ia. Broadcast m1 ⊑ i Broadcast m2 ⇒ Deliver m2 ⊑ i a Broadcast m1 ⇒ False
    by (metis broadcast-before-delivery broadcasts-unique insert-subset local-order-carrier-closed node-total-order-irrefl
     node-total-order-trans)
next
  show ∃ m2a. Broadcast m1 ⊑ i Broadcast m2 ⇒ hb m2 m2a ⇒ hb m2a m1 ⇒ False
    using assms (1) assms (2) hb.intros (3) hb-irrefl by blast
qed
next
fix m1 i m2
assume hb m2 m1
  and Deliver m1 ⊑ i Broadcast m2
thus False
  apply − proof (erale hb-elim)
  show ∃ ia. Deliver m1 ⊑ i Broadcast m2 ⇒ Deliver m2 ⊑ i a Broadcast m1 ⇒ False
    by (metis broadcast-before-delivery broadcasts-unique insert-subset local-order-carrier-closed node-total-order-irrefl
     node-total-order-trans)
next
  show ∃ ia. Deliver m1 ⊑ i Broadcast m2 ⇒ Deliver m2 ⊑ i a Broadcast m1 ⇒ False
    by (meson causal-network.causal-delivery causal-network-axioms hb.intros (2) hb.intros (3) insert-subset
     local-order-carrier-closed node-total-order-irrefl)
next
  show ∃ m2a. Deliver m1 ⊑ i Broadcast m2 ⇒ hb m2 m2a ⇒ hb m2a m1 ⇒ False
    by (meson causal-delivery hb.intros (2) insert-subset local-order-carrier-closed network hb.intros (3)
     network-axioms node-total-order-irrefl)
qed
next
fix m1 m2 m3
assume hb m1 m2 hb m2 m3 hb m3 m1
  and (hb m2 m1 ⇒ False) (hb m3 m2 ⇒ False)
thus False
  using hb.intros (3) by blast
qed

definition (in network) node-deliver-messages :: 'msg event list ⇒ 'msg list where
node-deliver-messages cs ≡ List.map-filter (λ e. case e of Deliver m ⇒ Some m | - ⇒ None) cs

lemma (in network) node-deliver-messages-empty [simp]:
  shows node-deliver-messages [] = []
  by (auto simp add: node-deliver-messages-def List.map-filter-simps)

lemma (in network) node-deliver-messages-Cons:
  shows node-deliver-messages (x#xs) = (node-deliver-messages [x])@(node-deliver-messages xs)
by\((\text{auto\ simp\ add:\ node-deliver-messages-def\ map-filter-def})\)

\textbf{lemma \ (in network) node-deliver-messages-append:}
\textbf{shows} node-deliver-messages \((xs @ ys)\) = \((\text{node-deliver-messages } xs) @ (\text{node-deliver-messages } ys)\)
\textbf{by\((\text{auto\ simp\ add:\ node-deliver-messages-def\ map-filter-def})\)}

\textbf{lemma \ (in network) node-deliver-messages-Broadcast \ [simp]:}
\textbf{shows} node-deliver-messages \([\text{Broadcast } m]\) = []
\textbf{by\((\text{clarsimp simp\ add:\ node-deliver-messages-def\ map-filter-def})\)}

\textbf{lemma \ (in network) node-deliver-messages-Deliver \ [simp]:}
\textbf{shows} node-deliver-messages \([\text{Deliver } m]\) = [m]
\textbf{by\((\text{clarsimp simp\ add:\ node-deliver-messages-def\ map-filter-def})\)}

\textbf{lemma \ (in network) prefix-msg-in-history:}
\textbf{assumes} es \text{prefix of } i \text{ and } m \in \text{set (node-deliver-messages } es\text{)}
\textbf{shows} \text{Deliver } m \in \text{set } \text{history } i
\textbf{using assms prefix-to-carriers by\((\text{fastforce simp\ add:\ node-deliver-messages-def\ map-filter-def\ split: event.split-asm})\)}

\textbf{lemma \ (in network) prefix-contains-msg:}
\textbf{assumes} es \text{prefix of } i \text{ and } m \in \text{set (node-deliver-messages } es\text{)}
\textbf{shows} \text{Deliver } m \in \text{set } es
\textbf{using assms by\((\text{auto simp\ add:\ node-deliver-messages-def\ map-filter-def\ split: event.split-asm})\)}

\textbf{lemma \ (in network) node-deliver-messages-distinct:}
\textbf{assumes} xs \text{prefix of } i
\textbf{shows} \text{distinct (node-deliver-messages } xs\text{)}
\textbf{using assms proof(\text{induction } xs \text{ rule: rev-induct})}
\textbf{case} \text{Nil} \text{ thus } ??\text{case by simp}\n\textbf{next}
\textbf{case} \text{snoc } x \text{ xs}
\textbf{fix} y \text{ assume } ??: y \in \text{set (node-deliver-messages } xs\text{)} \text{ y } \in \text{set (node-deliver-messages } [x]\text{)}
\textbf{moreover have} \text{distinct } (xs @ [x])
\textbf{ultimately have} False
\textbf{using assms apply(\text{case-tac } x; \text{clarsimp simp add: map-filter-def node-deliver-messages-def})}
\textbf{using } ??\text{ by blast}
\textbf{} thus ??\text{ by blast}
\textbf{using} snoc \text{ by\((\text{fastforce simp add: node-deliver-messages-append node-deliver-messages-def map-filter-def)})\)
\textbf{qed}

\textbf{lemma \ (in network) drop-last-message:}
\textbf{assumes} evts \text{prefix of } i
\textbf{and} node-deliver-messages evts = msgs @ [last-msg]
\textbf{shows} \exists \text{ pre. pre } \text{prefix of } i \land \text{node-deliver-messages } pre = msgs
\textbf{proof –}
\textbf{have} Deliver last-msg \in \text{set } evts
\textbf{using assms network.prefix-contains-msg network-axioms by force}
\textbf{then obtain} idx where ???: idx < length evts evts ! idx = Deliver last-msg
\textbf{by \((\text{meson set-elem-nth})\)}
\textbf{then obtain} pre suf where evts = pre @ (evts ! idx) # suf
\textbf{using id-take-nth-drop by blast}
\textbf{hence }???: \text{evts} = \text{pre } @ (\text{Deliver last-msg}) # \text{suf}
\textbf{using assms }???: \text{by auto}
\textbf{moreover hence} \text{distinct (node-deliver-messages } ([\text{Deliver last-msg}] @ suf))
\textbf{by \((\text{metis assms(1) assms(2) distinct-singleton node-deliver-messages-Cons node-deliver-messages-Deliver})\)}
node-deliver-messages-append node-deliver-messages-distinct not-Cons-self2 pre-suf-eq-distinct-list)
ultimately have node-deliver-messages ([Deliver last-msg] @ suf) = [last-msg] @ []
by (metis append-self-cone assms(1) assms(2) node-deliver-messages-Cons node-deliver-messages-Deliver
node-deliver-messages-append node-deliver-messages-distinct not-Cons-self2 pre-suf-eq-distinct-list)
thus ?thesis
using assms * * by (metis append1-eq-conv append-Cons append-Nil node-deliver-messages-append
prefix-of-appendD)
qed

locale network-with-ops =
causal-network history fst
for history :: nat ⇒ ('msgid × 'op) event list +
fixes interp :: 'op ⇒ 'state ↭ 'state
and initial-state :: 'state

context network-with-ops begin

definition interp-msg :: 'msgid × 'op ⇒ 'state ↭ 'state
where
interp-msg msg state ≡ interp (snd msg) state

sublocale hb: happens-before weak-hb hb interp-msg
proof
fix x y :: 'msgid × 'op
show hb x y = (weak-hb x y ∧ ¬ weak-hb y x)
  unfolding weak-hb-def using hb-antisym by blast
next
fix x
show weak-hb x x
  using weak-hb-def by blast
next
fix x y z
assume weak-hb x y weak-hb y z
thus weak-hb x z
  using weak-hb-def by (metis network hb intros(3) network-axioms)
qed

end
definition (in network-with-ops) apply-operations :: ('msgid × 'op) event list ⇒ 'state where
apply-operations es ≡ hb.apply-operations (node-deliver-messages es) initial-state
definition (in network-with-ops) node-deliver-ops :: ('msgid × 'op) event list ⇒ 'op list where
node-deliver-ops es ⇔ map snd (node-deliver-messages es)
lemma (in network-with-ops) apply-operations-empty [simp]:
shows apply-operations [] = Some initial-state
by(auto simp add: apply-operations-def)
lemma (in network-with-ops) apply-operations-Broadcast [simp]:
shows apply-operations (xs @ [Broadcast m]) = apply-operations xs
by(auto simp add: apply-operations-def node-deliver-messages-def map-filter-def)
lemma (in network-with-ops) apply-operations-Deliver [simp]:
shows apply-operations (xs @ [Deliver m]) = (apply-operations xs ⇒ interp-msg m)
by(auto simp add: apply-operations-def node-deliver-messages-def map-filter-def kleisli-def)
lemma (in network-with-ops) hb-consistent-technical:
assumes | m n. m < length cs ⇒ n < m ⇒ cs! n ⊆ cs! m
shows hb hb-consistent (node-deliver-messages cs)
using assms proof (induction cs rule: rev-induct)
case Nil thus case by (simp add: node-deliver-messages-def hb hb-consistentintros(1) map-filter-simps(2))
next
case (snoc x xs)
hence case: \((\forall m. n < length xs \Longrightarrow n < m \Longrightarrow xs ! n \sqsubseteq xs ! m)\)
  by (-, erule-tac x=m in meta-allE, erule-tac x=n in meta-allE, clarsimp simp add: nth-append)
then show case proof (cases x)
case (Broadcast x1)
  thus ?thesis using snoc ∗ by (clarsimp simp add: node-deliver-messages-append)
next
case (Deliver x2)
  thus ?thesis using snoc ∗ [simproc del: defined-all]
  apply (clarsimp simp add: node-deliver-messages-def map-filter-def map-filter-append)
  apply (rename-tac m m1 m2)
  apply (case-tac m; clarsimp)
  apply (drule set-elem-nth, erule exE, erule conjE)
  apply (clarsimp simp add: nth-append)
  apply (clarsimp simp add: node-total-order-antisym)
  done
qed

corollary (in network-with-ops)
  shows hb hb-consistent (node-deliver-messages (history i))
  by (metis hb-consistent-technical history-order-def less-one linorder-neqE-nat list-nth-split zero-order(3))
lemma (in network-with-ops) hb-consistent-prefix:
  assumes xs prefix of i
  shows hb hb-consistent (node-deliver-messages xs)
using assms proof (clarsimp simp: prefix-of-node-history-def, rule-tac i=i in hb-consistent-technical)
fix m n ys assume ∗: xs @ ys = history i m < length xs n < m
consider (a) xs = [] | (b) \(\exists c. xs = [c]\) | (c) Suc 0 < length (xs)
  by (metis Suc-pred length-Suc-conv length-greater-0-conv zero-less-diff)
thus xs ! n \sqsubseteq xs ! m
proof (cases)
case a thus ?thesis using ∗ by clarsimp
next
case b thus ?thesis using assms ∗ by clarsimp
next
case c thus ?thesis using assms ∗ apply clarsimp
apply (erule list-nth-split, assumption, clarsimp simp: c)
apply (metis append.assoc append.simps(2) history-order-def)
  done
qed
locale network-with-constrained-ops = network-with-ops +
fixes valid-msg :: 'c ⇒ ('a × 'b) ⇒ bool
assumes broadcast-only-valid-msgs: pre @ [Broadcast m] prefix of i ⇒
  \(\exists state. apply-operations pre = Some state ∧ valid-msg state m\)
lemma (in network-with-constrained-ops) broadcast-is-valid:
  assumes Broadcast m ∈ set (history i)
  shows ∃ state. valid-msg state m
  using assms broadcast-only-valid-msgs events-before-exist by blast

lemma (in network-with-constrained-ops) deliver-is-valid:
  assumes Deliver m ∈ set (history i)
  shows ∃ j pre state. pre @ [Broadcast m] prefix of j ∧ apply-operations pre = Some state ∧ valid-msg state m
  using assms apply (clarsimp dest: delivery-has-a-cause)
  using broadcast-only-valid-msgs events-before-exist apply blast
  done

lemma (in network-with-constrained-ops) deliver-in-prefix-is-valid:
  assumes xs prefix of i
    and Deliver m ∈ set xs
  shows ∃ state. valid-msg state m
  by (meson assms network-with-constrained-ops.deliver-is-valid network-with-constrained-ops-axioms prefix-elem-to-carriers)

4.4 Dummy network models

interpretation trivial-node-histories: node-histories λm. []
  by standard auto

interpretation trivial-network: network λm. [] id
  by standard auto

interpretation trivial-causal-network: causal-network λm. [] id
  by standard auto

interpretation trivial-network-with-ops: network-with-ops λm. [] (λx y. Some y) 0
  by standard auto

interpretation trivial-network-with-constrained-ops: network-with-constrained-ops λm. [] (λx y. Some y) 0 (λx y. True)
  by standard (simp add: trivial-node-histories.prefix-of-node-history-def)

end

5 Replicated Growable Array

The RGA, introduced by [10], is a replicated ordered list (sequence) datatype that supports insert and delete operations.

theory
  Ordered-List
imports
  Util
begin

  type-synonym ('id, 'v) elt = 'id × 'v × bool

5.1 Insert and delete operations

Insertion operations place the new element after an existing list element with a given ID, or at the head of the list if no ID is given. Deletion operations refer to the ID of the list element
that is to be deleted. However, it is not safe for a deletion operation to completely remove a list element, because then a concurrent insertion after the deleted element would not be able to locate the insertion position. Instead, the list retains so-called tombstones: a deletion operation merely sets a flag on a list element to mark it as deleted, but the element actually remains in the list. A separate garbage collection process can be used to eventually purge tombstones [10], but we do not consider tombstone removal here.

**hide-const** insert

**fun** insert-body :: ('id::{linorder}, 'v) elt list ⇒ ('id, 'v) elt ⇒ ('id, 'v) elt list **where**

insert-body [] e = [e] |
insert-body (x#xs) e =
  (if fst x < fst e then
   e#x#xs
  else x#insert-body xs e)

**fun** insert :: ('id::{linorder}, 'v) elt list ⇒ ('id, 'v) elt ⇒ 'id option ⇒ ('id, 'v) elt list option **where**

insert xs e None = Some (insert-body xs e) |
insert [] e (Some i) = None |
insert (x#xs) e (Some i) =
  (if fst x = i then
    Some (x#insert-body xs e)
  else
einsert xs e (Some i) ≡ (∀t. Some (x#t)))

**fun** delete :: ('id::{linorder}, 'v) elt list ⇒ 'id ⇒ ('id, 'v) elt list option **where**

delete [] i = None |
delete ((i', v, flag)#xs) i =
  (if i' = i then
    Some ((i', v, True)#xs)
  else
delete xs i ≡ (∀t. Some ((i',v,flag)#t)))

5.2 Well-definedness of insert and delete

**lemma** insert-no-failure:

assumes i = None ∨ (∃i'. i = Some i' ∧ i' ∈ fst ' set xs)

shows ∃xs'. insert xs e i = Some xs'

using assms by(induction rule: insert.induct; force)

**lemma** insert-None-index-neq-None [dest]:

assumes insert xs e i = None

shows i ≠ None

using assms by(cases i, auto)

**lemma** insert-Some-None-index-not-in [dest]:

assumes insert xs e (Some i) = None

shows i ∉ fst ' set xs

using assms by(induction xs, auto split: if-split-asm bind-splits)

**lemma** index-not-in-insert-Some-None [simp]:

assumes i ∉ fst ' set xs

shows insert xs e (Some i) = None

using assms by(induction xs, auto)

**lemma** delete-no-failure:

assumes i ∈ fst ' set xs

shows ∃xs'. delete xs i = Some xs'
using assms by(induction xs; force)

lemma delete-None-index-not-in [dest]:
  assumes delete xs i = None
  shows i /∈ fst ' set xs
using assms by(induction xs, auto split: if-split-asm bind-splits simp add: fst-eq-Domain)

lemma index-not-in-delete-None [simp]:
  assumes delete xs i = None
  shows i /∈ fst ' set xs
using assms by(induction xs, auto split: if-split-asm bind-splits simp add: fst-eq-Domain)

5.3 Preservation of element indices

lemma insert-body-preserve-indices [simp]:
  shows fst ' set (insert-body xs e) = fst ' set xs ∪ {fst e}
by(induction xs, auto simp add: insert-commute)

lemma insert-preserve-indices:
  assumes ∃ ys. insert xs e i = Some ys
  shows fst ' set (the (insert xs e i)) = fst ' set xs ∪ {fst e}
using assms by(induction xs; cases i; auto simp add: insert-commute split: bind-splits)

corollary insert-preserve-indices':
  assumes insert xs e i = Some ys
  shows fst ' set (the (insert xs e i)) = fst ' set xs ∪ {fst e}
using assms insert-preserve-indices by blast

lemma delete-preserve-indices:
  assumes delete xs i = Some ys
  shows fst ' set xs = fst ' set ys
using assms by(induction xs arbitrary: ys; simp) (case-tac a; auto split: if-split-asm bind-splits)

5.4 Commutativity of concurrent operations

lemma insert-body-commutes:
  assumes fst e1 ≠ fst e2
  shows insert-body (insert-body xs e1) e2 = insert-body (insert-body xs e2) e1
using assms by(induction xs, auto)

lemma insert-insert-body:
  assumes fst e1 ≠ fst e2
  and i2 ≠ Some (fst e1)
  shows insert (insert-body xs e1) e2 i2 = insert xs e2 i2 ≫ (λys. Some (insert-body ys e1))
using assms by (induction xs; cases i2) (auto split: if-split-asm simp add: insert-body-commutes)

lemma insert-nil-None:
  assumes fst e1 ≠ fst e2
  and i ≠ fst e2
  and i2 ≠ Some (fst e1)
  shows insert [] e2 i2 ≫ (λys. insert ys e1 (Some i)) = None
using assms by (cases i2) clarsimp+

lemma insert-insert-body-commute:
  assumes i ≠ fst e1
  and fst e1 ≠ fst e2
  shows insert (insert-body xs e1) e2 (Some i) =
  insert xs e2 (Some i) ≫ (λy. Some (insert-body y e1))
using assms by (induction xs, auto simp add: insert-body-commutes)

lemma insert-commutes:
  assumes \( \text{fst } e \neq \text{fst } e_2 \)
  \( i_1 = \text{None} \lor i_1 \neq \text{Some } (\text{fst } e_2) \)
  \( i_2 = \text{None} \lor i_2 \neq \text{Some } (\text{fst } e_1) \)
  shows insert xs e1 i1 \( \geq \) (\( \lambda \text{ys. insert } \text{ys } e_2 i_2 \) = insert xs e2 i2 \( \geq \) (\( \lambda \text{ys. insert } \text{ys } e_1 i_1 \))
using assms proof (induction rule: insert.induct)

fix xs and e :: ('a, 'b) elt
  assumes i2 = None \lor i2 \neq Some (fst e) and fst e \neq fst e2
  thus insert xs e None \( \geq \) (\( \lambda \text{ys. insert } \text{ys } e_2 i_2 \) = insert xs e2 i2 \( \geq \) (\( \lambda \text{ys. insert } \text{ys } e \) (Some i))
by (auto simp add: insert-body-commutes intro: insert-insert-body)

next
fix i and e :: ('a, 'b) elt
  assumes \( \text{fst } e \neq \text{fst } e_2 \) and \( i_2 = \text{None} \lor i_2 \neq \text{Some } (\text{fst } e) \) and \( \text{Some } i = \text{None} \lor \text{Some } i \neq \text{Some } (\text{fst } e_2) \)
  thus insert [] e (Some i) \( \geq \) (\( \lambda \text{ys. insert } \text{ys } e_2 i_2 \) = insert [] e2 i2 \( \geq \) (\( \lambda \text{ys. insert } \text{ys } e \) (Some i))
by (auto intro: insert-Nil-None[symmetric])

next
fix xs i and x e :: ('a, 'b) elt
  assumes IH: \( \text{fst } x \neq i \) \( \Rightarrow \)
  \( \text{fst } e \neq \text{fst } e_2 \) \( \Rightarrow \)
  \( \text{Some } i = \text{None} \lor \text{Some } i \neq \text{Some } (\text{fst } e_2) \) \( \Rightarrow \)
  \( i_2 = \text{None} \lor i_2 \neq \text{Some } (\text{fst } e) \) \( \Rightarrow \)
  insert xs e (Some i) \( \geq \) (\( \lambda \text{ys. insert } \text{ys } e_2 i_2 \) = insert xs e2 i2 \( \geq \) (\( \lambda \text{ys. insert } \text{ys } e \) (Some i))
  apply 
  apply (erule disjE, clasimp, simp, rule conjI)
  apply (case_tac i2; force simp add: insert-body-commutes insert-insert-body-commute)
  apply (case_tac i2; clasimp cong: Option.bind-cong simp add: insert-insert-body split: bind-splits)
  apply force
  done
qed

lemma delete-commutes:
  shows delete xs i1 \( \geq \) (\( \lambda \text{ys. delete } \text{ys } i_2 \) = delete xs i2 \( \geq \) (\( \lambda \text{ys. delete } \text{ys } i_1 \))
by (induction xs, auto split: bind-splits if-split-asm)

lemma insert-body-delete-commute:
  assumes i2 \neq \text{fst } e
  shows delete (insert-body xs e) i2 \( \geq \) (\( \lambda t. \text{Some } (x \neq t) \)) = delete xs i2 \( \geq \) (\( \lambda y. \text{Some } (x \neq \text{insert-body } y e) \))
using assms by (induction xs arbitrary: x; cases e, auto split: bind-splits if-split-asm simp add: insert-body-delete-commute)

lemma insert-delete-commute:
  assumes i2 \neq \text{fst } e
  shows insert xs e i1 \( \geq \) (\( \lambda \text{ys. delete } \text{ys } i_2 \) = delete xs i2 \( \geq \) (\( \lambda \text{ys. insert } \text{ys } e \))
using assms by (induction xs; cases e; cases i1, auto split: bind-splits if-split-asm simp add: insert-body-delete-commute)
5.5 Alternative definition of insert

fun insert\' :: ('id::{linorder}, 'v) elt list ⇒ ('id, 'v) elt ⇒ 'id option → ('id::{linorder}, 'v) elt list

where

insert\' [] e None = Some [e] |
insert\' [] e (Some i) = None |
insert\' (x#xs) e None =
  (if fst x < fst e then
  Some (e#x#xs)
  else
case insert\' xs e None of
  None ⇒ None
| Some t ⇒ Some (x#t)) |
insert\' (x#xs) e (Some i) =
  (if fst x = i then
case insert\' xs e None of
  None ⇒ None
| Some t ⇒ Some (x#t)
else
case insert\' xs e (Some i) of
  None ⇒ None
| Some t ⇒ Some (x#t))

lemma [elim!, dest]:
assumes insert\' xs e None = None
shows False
using assms by(induction xs, auto split: if-split asm option.split-asm)

lemma insert-body-insert':
shows insert\' xs e None = Some (insert-body xs e)
by(induction xs, auto)

lemma insert-insert':
shows insert xs e i = insert\' xs e i
by(induction xs; cases e; cases i, auto split: option.split simp add: insert-body-insert')

lemma insert-body-stop-iteration:
assumes fst e > fst x
shows insert-body (x#xs) e = e#x#xs
using assms by simp

lemma insert-body-contains-new-elem:
shows ∃ p s. xs = p @ s ∧ insert-body xs e = p @ e # s
proof (induction xs)
case Nil thus ?case by force
next
case (Cons a xs)
then obtain p s where xs = p @ s ∧ insert-body xs e = p @ e # s by force
thus ?case
  apply clarsimp
  apply (rule conjI; clarsimp)
  apply force
  apply (rule-tac x=a # p in cxI, force)
done
qed

lemma insert-between-elements:
assumes xs = pre@ref#suf
and distinct (map fst xs)
and \( \forall i' \in \text{fst } \text{set} \ x s \implies i' < \text{fst } e \)
shows insert \( x \) \( e \rightarrow (\text{Some } (\text{fst } ref)) = \text{Some } (\text{pre } @ \text{ref } \# e \# \text{suf}) \)
using assms by (induction \( x s \) arbitrary; \text{pre suf } \text{force}) (case-tac \text{pre}; case-tac \text{suf}; \text{force})

lemma insert-preserves-order:
assumes \( \forall x \in \text{set as} \ a \neq \text{fst } x \)
and distinct (map fst xs)
shows \( \exists \text{pre suf }, x s = \text{pre } @ (a, b, c) \# \text{suf } \land (\forall y \in \text{set pre. } \text{fst } y \neq a) \)
using assms proof (induction \( x s \), clarsimp)
case (Cons \( x \) \( x s \))
\{ assume \( x \neq (a, b, c) \)
    hence \( (a, b, c) \in \text{set} \ x s \) distinct (map fst xs)
    using Cons.prems by force+
then obtain \( \text{pre suf } \) where \( x s = \text{pre } @ (a, b, c) \# \text{suf } \land (\forall y \in \text{set pre. } \text{fst } y \neq a) \)
    using Cons.IH by force
    hence \( ?\text{case} \)
    apply (rule-tac \( x = x \# \text{pre } \in \text{exI} \))
    using Cons.prems (2) by auto
\} thus \( ?\text{case} \)
by force
qed

lemma insert-preserves-order-
assumes \( i = \text{None } \lor (\exists i', i = \text{Some } i' \land i' \in \text{fst } \text{set} x s) \)
and distinct (map fst xs)
shows \( \exists \text{pre suf }, x s = \text{pre } @ \text{suf } \land \text{insert } e \# i = \text{Some } (\text{pre } @ e \# \text{suf}) \)
using assms proof 
\{ assume \( i = \text{None} \)
    hence \( ?\text{thesis} \)
    by clarsimp (metis insert-body-contains-new-elem)
\} moreover 
\{ assume \( \exists i', i = \text{Some } i' \land i' \in \text{fst } \text{set} x s \)
then obtain \( j v b \) where \( i = \text{Some } j (j, v, b) \in \text{set} x s \) by force
moreover then obtain \( \text{as bs where} \ x s = \text{as } @ (j, v, b) \# \text{bs } \forall x \in \text{set as. } \text{fst } x \neq j \)
using assms by (metis split-tuple-list-by-id)
moreover then obtain \( \text{cs ds where} \text{insert-body } bs e = \text{cs } @ e \# \text{bs } \text{cs } @ \text{ds } = \text{bs} \)
by (metis insert-body-contains-new-elem)
ultimately have \( ?\text{thesis} \)
by (rule-tac \( x = \text{as } @ (j, v, b) \# \text{cs } \in \text{exI}; \text{clarsimp} \) (metis insert-position-element-technical)
\} ultimately show \( ?\text{thesis} \)
using assms by force
qed

end

5.6 Network

theory RGA
imports
Network
Ordered-List
datatype ('id, 'v) operation =
  Insert ('id, 'v) elt 'id option |
  Delete 'id

fun interpret-opers :: ('id::linorder, 'v) operation ⇒ ('id, 'v) elt list (⟨⟩ [0] 1000)
where
  interpret-opers (Insert e n) xs = insert xs e n |
  interpret-opers (Delete n) xs = delete xs n

definition element-ids :: ('id, 'v) elt list ⇒ 'id set where
  element-ids list ≡ set (map fst list)

definition valid-rga-msg :: ('id, 'v) elt list ⇒ 'id × ('id::linorder, 'v) operation ⇒ bool where
  valid-rga-msg list msg ≡ case msg of
    (i, Insert e None) ⇒ fst e = i |
    (i, Insert e (Some pos)) ⇒ fst e = i ∧ pos ∈ element-ids list |
    (i, Delete pos) ⇒ pos ∈ element-ids list

locale rga = network-with-constrained-ops - interpret-opers [] valid-rga-msg

definition indices :: ('id × ('id, 'v) operation) event list ⇒ 'id list where
  indices xs ≡ List.map-filter (λx. case x of Deliver (i, Insert e n) ⇒ Some (fst e) | - ⇒ None) xs

lemma indices-Nil [simp]:
  shows indices [] = []
by(auto simp: indices-def map-filter-def)

lemma indices-append [simp]:
  shows indices (xs@ys) = indices xs @ indices ys
by(auto simp: indices-def map-filter-def)

lemma indices-Broadcast-singleton [simp]:
  shows indices [Broadcast b] = []
by(auto simp: indices-def map-filter-def)

lemma indices-Deliver-Insert [simp]:
  shows indices [Deliver (i, Insert e n)] = [fst e]
by(auto simp: indices-def map-filter-def)

lemma indices-Deliver-Delete [simp]:
  shows indices [Deliver (i, Delete n)] = []
by(auto simp: indices-def map-filter-def)

lemma (in rga) idx-in-elem-inserted [intro]:
  assumes Deliver (i, Insert e n) ∈ set xs
  shows fst e ∈ set (indices xs)
using assms by(induction xs, auto simp add: indices-def map-filter-def)

lemma (in rga) apply-opers-idx-elems:
  assumes es prefix of i
  and apply-operations es = Some xs
  shows element-ids xs = set (indices es)
using assms unfolding element-ids-def
proof(induction es arbitrary: xs rule: rev-induct, clarsimp)
case (snoc x xs) thus ?case

proof (cases x, clarsimp, blast)
case (Deliver e)
  moreover obtain a b where e = (a, b) by force
  ultimately show ?thesis
    using snoc assms apply (cases b; clarsimp split: bind-splits simp add: interp-msg-def)
    apply (metis Un-insert-right append.right-neutral insert-preserve-indices' list.set(1)
      option.sel prefix-of-appendD prod.sel(1) set-append)
    apply (metis delete-preserve-indices prefix-of-appendD)
    qed

qed

lemma (in rga) delete-does-not-change-element-ids:
assumes es @ [Deliver (i, Delete n)] prefix of j
and apply-operations es = Some xs1
and apply-operations (es @ [Deliver (i, Delete n)]) = Some xs2
shows element-ids xs1 = element-ids xs2

proof
  have indices es = indices (es @ [Deliver (i, Delete n)])
    by simp
  then show ?thesis
    using snoc assms prefix-of-appendD rga.apply-opers-idx-elems rga-axioms
    by (metis no-types)

qed

lemma (in rga) someone-inserted-id:
assumes es @ [Deliver (i, Insert (k, v, f) n)] prefix of j
and apply-operations es = Some xs1
and apply-operations (es @ [Deliver (i, Insert (k, v, f) n)]) = Some xs2
and a ∈ element-ids xs2
and a ≠ k
shows a ∈ element-ids xs1

using assms apply-opers-idx-elems by auto

lemma (in rga) deliver-insert-exists:
assumes es prefix of j
and apply-operations es = Some xs
and a ∈ element-ids xs
shows ∃ i v f n. Deliver (i, Insert (a, v, f) n) ∈ set es

using assms unfolding element-ids-def

proof (induction es arbitrary: xs rule: rev-induct, clarsimp)
case (snoc x xs ys) thus ?case

proof (cases x)
case (Broadcast e) thus ?thesis
  using snoc by(clarsimp, metis image-eqI prefix-of-appendD prod.sel(1))

next

case (Deliver e)
moreover then obtain xs' where *: apply-operations xs = Some xs'
using snoc by fastforce
moreover obtain k v where **: e = (k, v) by force
ultimately show ?thesis
using assms snoc proof (cases v)
case (Insert el -)
  thus ?thesis
  using snoc Deliver * **
  apply (cases el; cases fst el = a; clarsimp)
  apply (blast, metis (no-types, lifting) element-ids-def prefix-of-appendD set-map snoc.prems(2)
    snoc.prems(3) someone-inserted-id)
done

next
case (Delete -) thus \( \text{thesis} \)
  using snoc Deliver ** apply clarsimp
  apply\( (\text{drule prefix-of-appendD, clarsimp simp add: } \text{bind-eq-\text{Some-conv}} \text{ interp-msg-def}) \)
  apply\( (\text{metis delete-preserve-indices image-eqI prod.sel(1)}) \)
  done
qed

lemma (in rga) insert-in-apply-set:
  assumes es @ [\text{Broadcast} (i, Insert e (\text{Some a}))] prefix of j
  and \text{Broadcast} (i', Insert e' n) \in set es
  and apply-operations es = \text{Some s}
  shows \text{fst e' \in element-ids s}
using assms apply-ops-idz-elems idx-in-elem-inserted prefix-of-appendD by blast

lemma (in rga) insert-msg-id:
  assumes Broadcast (i, Insert e n) \in set (\text{history} j)
  shows \text{fst e} = i
proof --
  obtain state where 1: valid-rga-msg state (i, Insert e n)
    using assms broadcast-is-valid by blast
  thus \text{fst e} = i
  by (clarsimp simp add: valid-rga-msg-def split: option.split-asm)
qed

lemma (in rga) allowed-insert:
  assumes Broadcast (i, Insert e n) \in set (\text{history} j)
  shows \text{n = Some \( \vee (\exists i' e' n'. \text{n = Some (fst e')} \wedge \text{Broadcast} (i', Insert e' n') \sqsubseteq j\) Broadcast (i, Insert e n)}
proof --
  obtain pre where 1: pre @ [\text{Broadcast} (i, Insert e n)] prefix of j
    using assms events-before-exist by blast
  from this obtain state where 2: apply-operations pre = Some state and 3: valid-rga-msg state (i, Insert e n)
    using broadcast-only-valid-msgs by blast
  show \text{n = Some \( \vee (\exists i' e' n'. \text{n = Some (fst e')} \wedge \text{Broadcast} (i', Insert e' n') \sqsubseteq j\) Broadcast (i, Insert e n)}
  proof (cases n)
    fix a
    assume 4: \text{n = Some a}
    hence \text{a \in element-ids state and 5: fst e} = i
    using 3 by (clarsimp simp add: valid-rga-msg-def)+
    from this have \( \exists i' v' f' n'. \text{Deliver (i', Insert (a, v', f') n') \in set pre} \)
    using deliver-insert-exists 2 1 by blast
    thus \text{n = Some \( \vee (\exists i' e' n'. \text{n = Some (fst e')} \wedge \text{Broadcast} (i', Insert e' n') \sqsubseteq j\) Broadcast (i, Insert e n)}
  using events-in-local-order 1 4 5 by (metis fst-conv)
qed simp

lemma (in rga) allowed-delete:
  assumes Broadcast (i, Delete e) \in set (\text{history} j)
  shows \( \exists i' n' \in b. \text{Deliver (i', Insert (x, v, b) n') \sqsubseteq j\) Broadcast (i, Delete e)
proof --
  obtain pre where 1: pre @ [\text{Broadcast (i, Delete e)}] prefix of j
    using assms events-before-exist by blast
  from this obtain state where 2: apply-operations pre = Some state
lemma (in rga) insert-id-unique:
assumes \( \text{fst } e_1 = \text{fst } e_2 \)
and Broadcast \((i_1, \text{Insert } e_1 n_1)\) \(\in\) set \(\text{history } i\)
and Broadcast \((i_2, \text{Insert } e_2 n_2)\) \(\in\) set \(\text{history } j\)
shows \(\text{Insert } e_1 n_1 = \text{Insert } e_2 n_2\)
using assms insert-msg-id msg-id-unique Pair-inject fst-conv by metis

lemma (in rga) allowed-delete-deliver:
assumes \(\text{Deliver } (i, \text{Delete } x)\) \(\in\) set \(\text{history } j\)
shows \(\exists i' n' v b. \text{Deliver } (i', \text{Insert } (x, v, b) n') \sqsupset_1 \text{Deliver } (i, \text{Delete } x)\)
using assms by (meson \(\text{allowed-insert-deliver bot-least causal-broadcast delivery-has-a-cause insert-subset}\))

lemma (in rga) allowed-delete-deliver-in-set:
assumes \((\text{es@}[\text{Deliver } (i, \text{Delete } m)])\) \(\text{prefix of } j\)
shows \(\exists i' n' v b. \text{Deliver } (i', \text{Insert } (m, v, b) n') \sqsupset_1 \text{Deliver } (i, \text{Delete } x)\)
using assms by (local-order-prefix-closed-last \(\text{rga.allowed-delete-deliver \text{rga-axioms set-append subsetCE prefix-to-carriers}\}))

lemma (in rga) allowed-insert-deliver:
assumes \(\text{Deliver } (i, \text{Insert } e n)\) \(\in\) set \(\text{history } j\)
shows \(n = \text{None } \lor (\exists i' n'' v b. n = \text{Some } n' \land \text{Deliver } (i', \text{Insert } (n', v, b) n'') \sqsupset_1 \text{Deliver } (i, \text{Insert } e n))\)
proof –
obtain \(ja\) where 1: Broadcast \((i, \text{Insert } e n)\) \(\in\) set \(\text{history } ja\)
using assms delivery-has-a-cause by blast
show \(n = \text{None } \lor (\exists i' n'' v b. n = \text{Some } n' \land \text{Deliver } (i', \text{Insert } (n', v, b) n'') \sqsupset_1 \text{Deliver } (i, \text{Insert } e n))\)
proof (cases \(n\))
  fix \(a\)
  assume 3: \(n = \text{Some } a\)
  from this obtain \(i' e' n'\) where 4: \(\text{Some } a = \text{Some } (\text{fst } e')\) and
  2: \(\text{Deliver } (i', \text{Insert } e' n') \sqsupset a \text{Broadcast } (i, \text{Insert } e \text{(Some } a))\)
using allowed-insert 1 by blast
hence \(\text{Deliver } (i', \text{Insert } e' n') \in\) set \(\text{history } ja\) and Broadcast \((i, \text{Insert } e \text{(Some } a))\) \(\in\) set \(\text{history } ja\)
  using local-order-carrier-closed by simp+
  from this obtain \(ja\) where Broadcast \((i, \text{Insert } e \text{(Some } a))\) \(\in\) set \(\text{history } ja\)
  using delivery-has-a-cause by simp
  have \(\exists i'' n'' v b. n = \text{Some } n' \land \text{Deliver } (i', \text{Insert } (n', v, b) n'') \sqsupset_1 \text{Deliver } (i, \text{Insert } e n)\)
  using 2 3 4 by (metis assms causal-broadcast prod.collapse)
  thus \(n = \text{None } \lor (\exists i'' n'' v b. n = \text{Some } n' \land \text{Deliver } (i', \text{Insert } (n', v, b) n'') \sqsupset_1 \text{Deliver } (i, \text{Insert } e n))\)
    by auto
qed simp
qed

lemma (in rga) allowed-insert-deliver-in-set:
assumes \((\text{es@}[\text{Deliver } (i, \text{Insert } e m)])\) \(\text{prefix of } j\)
shows  \( m = \text{None} \lor (\exists i' \ m' \ n \ v \ b. \ m = \text{Some} \ m' \land \text{Deliver} (i', \text{Insert} (m', v, b) \ n) \in \text{set} \ es) \)

by(\text{metis} \ \text{assms} \ \text{Un-insert-right} \ \text{insert-subset} \ \text{list} \ \text{set} \ \text{append} \ \text{prefix-to-carriers} \ \text{allowed-insert-deliver} \ \text{local-order-prefix-failure} \ \text{prefix-closed-last})

\text{lemma (in rga) Insert-no-failure:}
\begin{align*}
\text{assumes} & \quad es \@ [\text{Deliver} (i, \text{Insert} e n)] \text{ prefix of } j \\
\text{and} & \quad \text{apply-operations } es = \text{Some } s \\
\text{shows} & \quad \exists ys. \text{ insert } s \ e \ n = \text{Some } y s
\end{align*}

by(\text{metis} ((\text{no-types}, \text{lifting}) \ \text{element-ids-def} \ \text{allowed-insert-deliver-in-set} \ \text{assms} \ \text{fst-conv} \ \text{insert-in-apply-set} \ \text{insert-no-failure} \ \text{set-map})

\text{lemma (in rga) delete-no-failure:}
\begin{align*}
\text{assumes} & \quad es \@ [\text{Deliver} (i, \text{Delete} n)] \text{ prefix of } j \\
\text{and} & \quad \text{apply-operations } es = \text{Some } s \\
\text{shows} & \quad \exists ys. \text{ delete } s \ n = \text{Some } y s
\end{align*}

\text{proof} –
\begin{align*}
\text{obtain} & \quad i' na v b \text{ where } 1: \text{Deliver} (i', \text{Insert} (n, v, b) \ na) \in \text{set} \ es \\
\text{using} & \quad \text{assms allowed-delete-deliver-in-set by blast} \\
\text{also have} & \quad \text{fst} (n, v, b) \in \text{set} (\text{indices} \ es) \\
\text{using} & \quad \text{assms idxs-in-elem-inserted calculation by blast} \\
\text{from} & \quad \text{this assms and 1 show } \exists ys. \text{ delete } s \ n = \text{Some } y s \\
\text{apply} – \\
\text{apply} (\text{rule delete-no-failure}) \\
\text{apply} (\text{metis apply-opers-idx-elems element-ids-def prefix-of-appendD prod.sel(1) set-map})
\end{align*}

\text{done}

\text{qed}

\text{lemma (in rga) Insert-equal:}
\begin{align*}
\text{assumes} & \quad \text{fst } e 1 = \text{fst } e 2 \\
\text{and} & \quad \text{Broadcast} (i 1, \text{Insert } e 1 \ n 1) \in \text{set} (\text{history} i) \\
\text{and} & \quad \text{Broadcast} (i 2, \text{Insert } e 2 \ n 2) \in \text{set} (\text{history} j) \\
\text{shows} & \quad \text{Insert } e 1 \ n 1 = \text{Insert } e 2 \ n 2
\end{align*}

\text{using} \ \text{insert-id-unique} \ \text{assms by simp}

\text{lemma (in rga) same-insert:}
\begin{align*}
\text{assumes} & \quad \text{fst } e 1 = \text{fst } e 2 \\
\text{and} & \quad \text{xs prefix of } i \\
\text{and} & \quad (i 1, \text{Insert } e 1 \ n 1) \in \text{set} (\text{node-deliver-messages} \ \text{xs}) \\
\text{and} & \quad (i 2, \text{Insert } e 2 \ n 2) \in \text{set} (\text{node-deliver-messages} \ \text{xs}) \\
\text{shows} & \quad \text{Insert } e 1 \ n 1 = \text{Insert } e 2 \ n 2
\end{align*}

\text{proof} –
\begin{align*}
\text{have} & \quad \text{Deliver} (i 1, \text{Insert } e 1 \ n 1) \in \text{set} (\text{history} i) \\
\text{using} & \quad \text{assms by(\text{auto simp add: node-deliver-messages-def prefix-msg-in-history})} \\
\text{from} & \quad \text{this obtain } j \text{ where 1: } \text{Broadcast} (i 1, \text{Insert } e 1 \ n 1) \in \text{set} (\text{history} j) \\
\text{using} & \quad \text{delivery-has-a-cause by blast} \\
\text{have} & \quad \text{Deliver} (i 2, \text{Insert } e 2 \ n 2) \in \text{set} (\text{history} i) \\
\text{using} & \quad \text{assms by(\text{auto simp add: node-deliver-messages-def prefix-msg-in-history})} \\
\text{from} & \quad \text{this obtain } k \text{ where 2: } \text{Broadcast} (i 2, \text{Insert } e 2 \ n 2) \in \text{set} (\text{history} k) \\
\text{using} & \quad \text{delivery-has-a-cause by blast} \\
\text{show} & \quad \text{Insert } e 1 \ n 1 = \text{Insert } e 2 \ n 2 \\
\text{by(\text{rule Insert-equal}; \text{force simp add: assms intro: 1 2})}
\end{align*}

\text{qed}

\text{lemma (in rga) insert-commute-assms:}
\begin{align*}
\text{assumes} & \quad \{\text{Deliver} (i, \text{Insert } e n), \text{Deliver} (i', \text{Insert } e' n')\} \subseteq \text{set} (\text{history} j) \\
\text{and} & \quad \text{hb.concurrent} (i, \text{Insert } e n) (i', \text{Insert } e' n') \\
\text{shows} & \quad n = \text{None} \lor n \neq \text{Some} (\text{fst } e')
\end{align*}

\text{using} \ \text{assms}
apply(clarsimp simp: hb.concurrent-def)
apply(cases e')
apply clarsimp
apply(frule delivery-has-a-cause)
apply(frule delivery-has-a-cause, clarsimp)
apply(frule allowed-insert)
apply clarsimp
apply(metis Insert-equal delivery-has-a-cause fst-conv hb.intros(2) insert-subset
  local-order-carrier-closed insert-msg-id)
done

lemma subset-reorder:
  assumes {a, b} ⊆ c
  shows {b, a} ⊆ c
using assms by simp

lemma (in rga) Insert-Insert-concurrent:
  assumes {Deliver (i, Insert e n), Deliver (i', Insert e' (Some m))} ⊆ set (history j)
  and hb.concurrent (i, Insert e k) (i', Insert e' (Some m))
  shows fst e ≠ m
by (metis assms subset-reorder hb.concurrent-comm insert-commute-assms option.simps(3))

lemma (in rga) insert-valid-assms:
  assumes Deliver (i, Insert e n) ∈ set (history j)
  shows n = None ∨ n ≠ Some (fst e)
using assms by (meson allowed-insert-deliver hb.concurrent-def hb.less-asym insert-subset
  local-order-carrier-closed rga.insert-commute-assms rga-axioms)

lemma (in rga) Insert-Delete-concurrent:
  assumes {Deliver (i, Insert e n), Deliver (i', Delete n')} ⊆ set (history j)
  and hb.concurrent (i, Insert e n) (i', Delete n')
  shows n' ≠ fst e
by (metis assms Insert-equal allowed-delete delivery-has-a-cause fst-conv hb.concurrent-def
  hb.intros(2) insert-subset local-order-carrier-closed rga.insert-msg-id rga-axioms)

lemma (in rga) concurrent-operations-commute:
  assumes xs prefix of i
  shows hb.concurrent-ops-commute (node-deliver-messages xs)
proof -
  have ∀x y. {x, y} ⊆ set (node-deliver-messages xs) ⇒ hb.concurrent x y ⇒ interp-msg x ⊢
    interp-msg y = interp-msg y ⊢ interp-msg x
proof
  fix x y ii
  assume {x, y} ⊆ set (node-deliver-messages xs)
  and C: hb.concurrent x y
  hence X: x ∈ set (node-deliver-messages xs) and Y: y ∈ set (node-deliver-messages xs)
  by auto
  obtain x1 x2 y1 y2 where 1: x = (x1, x2) and 2: y = (y1, y2)
  by fastforce
  have (interp-msg (x1, x2) ⊢ interp-msg (y1, y2)) ii = (interp-msg (y1, y2) ⊢ interp-msg (x1, x2))
  by
  proof(cases x2; cases y2)
    fix ix1 ix2 iy1 iy2
    assume X2: x2 = Insert ix1 ix2 and Y2: y2 = Insert iy1 iy2
    show (interp-msg (x1, x2) ⊢ interp-msg (y1, y2)) ii = (interp-msg (y1, y2) ⊢ interp-msg (x1, x2))
  by
  proof(cases fst ix1 = fst iy1)
    assume fst ix1 = fst iy1
hence \( \text{Insert } ix1 \ ix2 = \text{Insert } iy1 \ iy2 \)

apply (rule same-insert)
using \( 1 \ 2 \ X \ Y \ X2 \ Y2 \) asms apply auto
done

hence \( ix1 = iy1 \) and \( ix2 = iy2 \)
by auto

from this and \( X2 \ Y2 \) show \( (\text{interp-msg } (x1, x2) \triangleright \text{interp-msg } (y1, y2)) \) \( ii = (\text{interp-msg } (y1, y2)) \) \( ii \)
by (clarsimp simp add: kleisli-def interp-msg-def)

next
assume NEQ: \( \text{fst } ix1 \neq \text{fst } iy1 \)
have \( ix2 = \text{None} \lor ix2 \neq \text{Some} (\text{fst } iy1) \)
apply (rule insert-commute-assms)
using prefix-msg-in-history[OF asms] \( X \ Y \ X2 \ Y2 \) \( 1 \ 2 \)
apply (clarsimp, blast)
using \( C \ 1 \ 2 \ X2 \ Y2 \) apply blast
done

also have \( iy2 = \text{None} \lor iy2 \neq \text{Some} (\text{fst } ix1) \)
apply (rule insert-commute-assms)
using prefix-msg-in-history[OF asms] \( X \ Y \ X2 \ Y2 \) \( 1 \ 2 \)
apply (clarsimp, blast)
using \( 1 \ 2 \ C \ X2 \ Y2 \) apply blast
done

ultimately have \( \text{insert } ii \ ix1 \ ix2 \gg (\lambda x. \text{insert } x \ iy1 \ iy2) = \text{insert } ii \ iy1 \ iy2 \gg (\lambda x. \text{insert } x \ ix1 \ ix2) \)

using NEQ insert-commutes by blast

thus \( (\text{interp-msg } (x1, x2) \triangleright \text{interp-msg } (y1, y2)) \) \( ii = (\text{interp-msg } (y1, y2) \triangleright \text{interp-msg } (x1, x2)) \) \( ii \)
by (clarsimp simp add: interp-msg-def X2 Y2 kleisli-def)

qed

next
fix \( ix1 \ ix2 \ \text{yd} \)
assume \( X2: x2 = \text{Insert } ix1 \ ix2 \) and \( Y2: y2 = \text{Delete } \text{yd} \)

have \( \text{hb.concurrent } (x1, \text{Insert } ix1 \ ix2) \) \( (y1, \text{Delete } yd) \)
using \( C \ X2 \ Y2 \) \( 1 \ 2 \) by simp

also have \( \{\text{Deliver } (x1, \text{Insert } ix1 \ ix2), \text{Deliver } (y1, \text{Delete } yd)\} \subseteq \text{set } (\text{history } i) \)
using prefix-msg-in-history asms \( X2 \ Y2 \) \( X \ Y \) \( 1 \ 2 \) by blast

ultimately have \( \text{yd} \neq \text{fst } ix1 \)

apply

apply (rule Insert-Delete-concurrent; force)
done

hence \( \text{insert } ii \ ix1 \ ix2 \gg (\lambda x. \text{delete } x \ \text{yd}) = \text{delete } ii \ \text{yd} \gg (\lambda x. \text{insert } x \ ix1 \ ix2) \)
by (rule insert-delete-commute)

thus \( (\text{interp-msg } (x1, x2) \triangleright \text{interp-msg } (y1, y2)) \) \( ii = (\text{interp-msg } (y1, y2) \triangleright \text{interp-msg } (x1, x2)) \) \( ii \)
by (clarsimp simp add: interp-msg-def kleisli-def X2 Y2)

next
fix \( xd \ iy1 \ iy2 \)
assume \( X2: x2 = \text{Delete } xd \) and \( Y2: y2 = \text{Insert } iy1 \ iy2 \)

have \( \text{hb.concurrent } (x1, \text{Delete } xd) \) \( (y1, \text{Insert } iy1 \ iy2) \)
using \( C \ X2 \ Y2 \) \( 1 \ 2 \) by simp

also have \( \{\text{Deliver } (x1, \text{Delete } xd), \text{Deliver } (y1, \text{Insert } iy1 \ iy2)\} \subseteq \text{set } (\text{history } i) \)
using prefix-msg-in-history asms \( X2 \ Y2 \) \( X \ Y \) \( 1 \ 2 \) by blast

ultimately have \( xd \neq \text{fst } iy1 \)

apply

apply (rule Insert-Delete-concurrent; force)
done

hence \( \text{delete } ii \ xd \gg (\lambda x. \text{insert } x \ iy1 \ iy2) = \text{insert } ii \ iy1 \ iy2 \gg (\lambda x. \text{delete } x \ xd) \)
by (rule insert-delete-commute[symmetric])
thus \((\text{interp-msg } (x1, x2) \triangleright \text{interp-msg } (y1, y2)) \ i_\text{ii} = (\text{interp-msg } (y1, y2) \triangleright \text{interp-msg } (x1, x2)) \ i_\text{ii}\)
by (clarsimp simp add: interp-msg-def kleisli-def X2 Y2)

next
fix \(x\ d\ y\ d\)
assume \(X2: x2 = \text{Delete } xd\) and \(Y2: y2 = \text{Delete } yd\)
have \(\text{delete } ii \ x d \cong (\lambda x. \text{delete } x \ y d) = \text{delete } ii \ y d \cong (\lambda x. \text{delete } x \ xd)\)
by (rule delete-commutes)
thus \((\text{interp-msg } (x1, x2) \triangleright \text{interp-msg } (y1, y2)) \ i_\text{ii} = (\text{interp-msg } (y1, y2) \triangleright \text{interp-msg } (x1, x2)) \ i_\text{ii}\)
by (clarsimp simp add: interp-msg-def kleisli-def X2 Y2)

qed

thus \((\text{interp-msg } x \triangleright \text{interp-msg } y) \ i_\text{ii} = (\text{interp-msg } y \triangleright \text{interp-msg } x) \ i_\text{ii}\)
using 1 2 by auto

qed

thus \(\text{hh.concurrent-ops-commute } (\text{node-deliver-messages } xs)\)
by (auto simp add: \(\text{hh.concurrent-ops-commute-def}\))

qed

\begin{corollary}
(in \(\text{rga}\)) \(\text{concurrent-operations-commute}'\):
\begin{quote}
shows \(\text{hb.concurrent-ops-commute } (\text{node-deliver-messages } (\text{history } i))\)
by (meson \(\text{concurrent-operations-commute-append}\), \(\text{right-neutral prefix-of-node-history-def}\))
\end{quote}
\end{corollary}

\begin{lemma}
(in \(\text{rga}\)) \(\text{apply-operations-never-fails}\):
\begin{quote}
assumes \(xs \text{ prefix of } i\)
shows \(\text{apply-operations } xs \neq \text{None}\)
using \(\text{assms}\) proof (induction \(xs\) rule: rev-induct)
show \(\text{apply-operations } [] \neq \text{None}\)
by clarsimp
\end{quote}
\end{lemma}

next
fix \(x\ xs\)
assume 1: \(xs \text{ prefix of } i \Rightarrow \text{apply-operations } xs \neq \text{None}\)
and 2: \(xs @ [x] \text{ prefix of } i\)
hence 3: \(xs \text{ prefix of } i\)
by auto
show \(\text{apply-operations } (xs @ [x]) \neq \text{None}\)
proof (cases \(x\))
fix \(b\)
assume \(x = \text{Broadcast } b\)
thus \(\text{apply-operations } (xs @ [x]) \neq \text{None}\)
using 1 3 by clarsimp

next
fix \(d\)
assume 4: \(x = \text{Deliver } d\)
thus \(\text{apply-operations } (xs @ [x]) \neq \text{None}\)
proof (cases \(d\); clarify)
fix \(a\ b\)
assume 5: \(x = \text{Deliver } (a, b)\)
show \(\exists y. \text{apply-operations } (xs @ [\text{Deliver } (a, b)]) = \text{Some } y\)
proof (cases \(b\); clarify)
fix \(aa\ aaa\ ba\ x12\)
assume 6: \(b = \text{Insert } (aa, aaa, ba) \ x12\)
show \(\exists y. \text{apply-operations } (xs @ [\text{Deliver } (a, \text{Insert } (aa, aaa, ba) \ x12)]) = \text{Some } y\)
apply (clarsimp simp add: 1 interp-msg-def split!: bind-splits)
apply (simp add: 1 3)
apply (rule rga.Insert-no-failure, rule rga-axioms)
using 4 5 6 2 apply force+
done
next
  fix \( x_2 \)
  \( \text{assume } 6: \ b = \text{Delete } x_2 \)
  \( \text{show } \exists \ y. \ \text{apply-operations } (xs @ [\text{Deliver } (a, \text{Delete } x_2)]) = \text{Some } y \)
  \( \text{apply(clarsimp simp add: interp-msg-def split!: bind-splits)} \)
  \( \text{apply(simp add: } 1 \ 3) \)
  \( \text{apply(rule delete-no-failure)} \)
  \( \text{using } 4 \ 5 \ 6 \ 2 \ \text{apply force+} \)
next
qed
qed
qed

\begin{enumerate}
\item \textbf{lemma (in rga) apply-operations-never-fails':}
  \begin{itemize}
  \item \textbf{shows} \( \text{apply-operations } (\text{history } i) \neq \text{None} \)
  \item \textbf{by}(meson apply-operations-never-fails append.right-neutral prefix-of-node-history-def)
  \end{itemize}
\end{enumerate}

\begin{enumerate}
\item \textbf{corollary (in rga) rga-convergence:}
  \begin{itemize}
  \item \textbf{assumes} \( \text{set } (\text{node-deliver-messages } xs) = \text{set } (\text{node-deliver-messages } ys) \)
  \item \( \text{and } xs \text{ prefix of } i \)
  \item \( \text{and } ys \text{ prefix of } j \)
  \item \textbf{shows} \( \text{apply-operations } xs = \text{apply-operations } ys \)
  \item \textbf{using} \( \text{assms by}(\text{auto simp add: apply-operations-def intro: hb.convergence-ext} \)
  \item \( \text{concurrent-operations-commute node-deliver-messages-distinct hb-consistent-prefix} \)
  \end{itemize}
\end{enumerate}

\section{5.7 Strong eventual consistency}
\begin{enumerate}
\item \textbf{context rga begin}
\item \textbf{sublocale sec: strong-eventual-consistency weak-hb hb interp-msg}
  \( \lambda \text{ops. } \exists xs \ i. \ xs \text{ prefix of } i \land \text{node-deliver-messages } xs = \text{ops } [] \)
\item \textbf{proof(standard; clarsimp)}
  \( \text{fix } xsa \ i \)
  \( \text{assume } xsa \text{ prefix of } i \)
  \( \text{thus } \text{hb.hb-consistent } (\text{node-deliver-messages } xsa) \)
  \( \text{by}(\text{auto simp add: hb-consistent-prefix}) \)
\item \textbf{next}
  \( \text{fix } xsa \ i \)
  \( \text{assume } xsa \text{ prefix of } i \)
  \( \text{thus } \text{distinct } (\text{node-deliver-messages } xsa) \)
  \( \text{by}(\text{auto simp add: node-deliver-messages-distinct}) \)
\item \textbf{next}
  \( \text{fix } xsa \ i \)
  \( \text{assume } xsa \text{ prefix of } i \)
  \( \text{thus } \text{hb.concurrent-ops-commute } (\text{node-deliver-messages } xsa) \)
  \( \text{by}(\text{auto simp add: concurrent-operations-commute}) \)
\item \textbf{next}
  \( \text{fix } xsa \ i \)
  \( \text{assume } xsa \text{ prefix of } i \)
  \( \text{thus } \exists y. \ \text{interp-msg } (a, b) \text{ state } = \text{Some } y \)
  \( \text{by}(\text{metis (no-types, lifting) apply-operations-def bind.bind-lunit not-None-eq} \)
  \( \text{hb.apply-operations-Snoc kleisli-def apply-operations-never-fails interp-msg-def}) \)
\item \textbf{next}
  \( \text{fix } xsa \ i \)
\end{enumerate}
assume node-deliver-messages xsa = xs @ [(a, b)]
and xsa prefix of x
thus ∃xsa. (∃x. xsa prefix of x) ∧ node-deliver-messages xsa = xs
using drop-last-message by blast
qed
end

interpretation trivial-rga-implementation: rga λx. []
by (standard, auto simp add: trivial-node-histories.history-order-def
trivial-node-histories.prefix-of-node-history-def)
end

6 Increment-Decrement Counter

The Increment-Decrement Counter is perhaps the simplest CRDT, and a paradigmatic example of a replicated data structure with commutative operations.

theory Counter
imports Network
begin

datatype operation = Increment | Decrement

fun counter-op :: operation ⇒ int ⇒ int where
counter-op Increment x = Some (x + 1) |
counter-op Decrement x = Some (x - 1)

locale counter = network-with-ops - counter-op 0

lemma (in counter) counter-op x ▷ counter-op y = counter-op y ▷ counter-op x
by (case-tac x; case-tac y; auto simp: kleisli-def)

lemma (in counter) concurrent-operations-commute:
assumes xs prefix of i
shows hb.concurrent-ops-commute (node-deliver-messages xs)
using assms
apply(clarsimp simp: hb.concurrent-ops-commute-def)
apply(rename-tac a b x y)
apply(case-tac b; case-tac y; force simp add: interp-msg-def kleisli-def)
done

corollary (in counter) counter-convergence:
assumes set (node-deliver-messages xs) = set (node-deliver-messages ys)
and xs prefix of i
and ys prefix of j
shows apply-operations xs = apply-operations ys
using assms by (auto simp add: apply-operations-def intro: hb.convergence-ext
concurrent-operations-commute node-deliver-messages-distinct hb-consistent-prefix)

context counter begin

sublocale sec: strong-eventual-consistency weak-hb hb interp-msg
λops. ∃xs i. xs prefix of i ∧ node-deliver-messages xs = ops 0
apply (standard;clarsimp simp add: hb-consistent-prefix drop-last-message)
node-deliver-messages-distinct concurrent-operations-commute
apply (metis (full-types) interp-msg-def counter-op.elims)
using drop-last-message apply blast
done

end
end

7 Observed-Remove Set

The ORSet is a well-known CRDT for implementing replicated sets, supporting two operations: the insertion and deletion of an arbitrary element in the shared set.

theory
  ORSet
imports
  Network
begin

datatype ('id, 'a) operation = Add 'id 'a | Rem 'id set 'a
type-synonym ('id, 'a) state = 'a ⇒ 'id set

definition op-elem :: ('id, 'a) operation ⇒ 'a where
  op-elem oper ≡ case oper of Add i e ⇒ e | Rem is e ⇒ e

definition interpret-op :: ('id, 'a) operation ⇒ ('id, 'a) state ⇒ ('id, 'a) state where
  interpret-op oper state ≡ let before = state (op-elem oper);
                         after = case oper of Add i e ⇒ before ∪ {i} | Rem is e ⇒ before − is
                         in Some (state ((op-elem oper) := after))

definition valid-behaviours :: ('id, 'a) state ⇒ ('id, 'id, 'a) operation ⇒ bool where
  valid-behaviours state msg ≡
  case msg of
  (i, Add j e) ⇒ i = j |
  (i, Rem is e) ⇒ is = state e

locale orset = network-with-constrained-ops - interpret-op λx. { } valid-behaviours

lemma (in orset) add-add-commute:
  shows (Add i1 e1) ▷ (Add i2 e2) = (Add i2 e2) ▷ (Add i1 e1)
by (auto simp add: interpret-op-def op-elem-def kleisli-def, fastforce)

lemma (in orset) add-rem-commute:
  assumes i ∉ is
  shows (Add i e1) ▷ (Rem is e2) = (Rem is e2) ▷ (Add i e1)
using assms by (auto simp add: interpret-op-def kleisli-def op-elem-def, fastforce)

lemma (in orset) apply-operations-never-fails:
  assumes xs prefix of i
  shows apply-operations xs ≠ None
using assms proof (induction xs rule: rev-induct, clarsimp)
case (snoc x xs) thus ?case
proof (cases x)
case (Broadcast e) thus ?thesis
  using snoc by force
next
case (Deliver e) thus thesis
  using snoc by (clarsimp, metis interpret-op-def interp-msg-def bind.bind-lunit prefix-of-appendD)
qed
qed

lemma (in orset) add-id-valid:
  assumes xs prefix of j
  and Deliver (i1, Add i2 e) ∈ set xs
  shows i1 = i2
proof –
  have ∃s. valid-behaviours s (i1, Add i2 e)
  using assms deliver-in-prefix-is-valid by blast
  thus thesis
  by (simp add: valid-behaviours-def)
qed

definition (in orset) added-ids :: ('id × ('id, 'b) operation) event list ⇒ 'id list where
  added-ids es p ≡ List.map-filter (λx. case x of Deliver (i, Add j e) ⇒ if e = p then Some j else None
  | - ⇒ None) es

lemma (in orset) [simp]:
  shows added-ids [] e = []
  by (auto simp: added-ids-def map-filter-def)

lemma (in orset) [simp]:
  shows added-ids (xs @ ys) e = added-ids xs e @ added-ids ys e
  by (auto simp: added-ids-def map-filter-append)

lemma (in orset) added-ids-Broadcast-collapse [simp]:
  shows added-ids ([Broadcast e]) e' = []
  by (auto simp: added-ids-def map-filter-append map-filter-def)

lemma (in orset) added-ids-Deliver-Rem-collapse [simp]:
  shows added-ids ([Deliver (i, Rem is e)]) e' = []
  by (auto simp: added-ids-def map-filter-append map-filter-def)

lemma (in orset) added-ids-Deliver-Add-diff-collapse [simp]:
  shows e ≠ e' ⇒ added-ids ([Deliver (i, Add j e)]) e' = []
  by (auto simp: added-ids-def map-filter-append map-filter-def)

lemma (in orset) added-ids-Deliver-Add-same-collapse [simp]:
  shows added-ids ([Deliver (i, Add j e)]) e = [j]
  by (auto simp: added-ids-def map-filter-append map-filter-def)

lemma (in orset) added-id-not-in-set:
  assumes i1 ∉ set (added-ids [Deliver (i, Add i2 e)] e)
  shows i1 ≠ i2
  using assms by simp

lemma (in orset) apply-operations-added-ids:
  assumes es prefix of j
  and apply-operations es = Some f
  shows f x ⊆ set (added-ids es x)
  using assms proof (induct es arbitrary: f rule: rev-induct, force)
  case (snoc x xs) thus ?case
  proof (cases x, force)
    case (Deliver e)
    moreover obtain a b where e = (a, b) by force
ultimately show \( \text{thesis} \)
using \text{snoc} by (\text{case-tac \( b \); clarsimp simp; interp-msg-def split; bind-splits,}
force split; if-split-asm simp add: op-elem-def interpret-op-def)
qed

lemma \text{(in orset)} \text{Deliver-added-ids}:
assumes \( xs \) prefix of \( j \)
and \( i \in \text{set} \) \( \text{added-ids} \) \( xs \) \( e \)
shows \text{Deliver} \( (i, \text{Add} \ i \ e) \in \text{set} \) \( xs \)
using \text{assms} \text{proof} (\text{induct} \) \( xs \) \text{rule: rev-induct, clarsimp)
case \( (\text{snoc} \ x \ xs) \) thus \( ?\text{case} \)
proof (cases \( x \), force)
moreover obtain \( a \ b \) where \( e' = (a, b) \) by force
ultimately show \( ?\text{thesis} \)
using \text{snoc apply (case-tac \( b \); clarsimp)}
apply \( \) (\text{metis added-ids-Deliver-Add-diff-collapse added-ids-Deliver-Add-same-collapse}
empty-iff list.set(1) set-ConsD add-id-valid in-set-conv-decomp prefix-of-appendD)
apply force
done
qed

lemma \text{(in orset)} \text{Broadcast-Deliver-prefix-closed}:
assumes \( xs \) \( \oplus \) \( [\text{Broadcast} \ (r, \text{Rem} \ ix \ e)] \) prefix of \( j \)
and \( i \in ix \)
sells \text{Deliver} \( (i, \text{Add} \ i \ e) \in \text{set} \) \( xs \)
proof –
obtain \( y \) where \text{apply-operations} \( xs \) \( = \) \text{Some} \( y \)
using \text{assms} broadcast-only-valid-msgs \text{by blast}
morerover hence \( ix = y \ e \)
by \( \) (\text{metis mono-tags, lifting} \) \text{assms}(1) broadcast-only-valid-msgs operation.case(2) \text{option.simps(1)}
valid-behaviours-def case-prodD)
ultimately show \( ?\text{thesis} \)
using \text{assms Deliver-added-ids apply-operations-added-ids by blast}
qed

lemma \text{(in orset)} \text{Broadcast-Deliver-prefix-closed2}:
assumes \( xs \) prefix of \( j \)
and \text{Broadcast} \( (r, \text{Rem} \ ix \ e) \in \text{set} \) \( xs \)
and \( i \in ix \)
sells \text{Deliver} \( (i, \text{Add} \ i \ e) \in \text{set} \) \( xs \)
using \text{assms Broadcast-Deliver-prefix-closed by (induction} \) \( xs \) \text{rule: rev-induct; force)

lemma \text{(in orset)} \text{concurrent-add-remove-independent-technical}:
assumes \( i \in is \)
and \( xs \) prefix of \( j \)
and \( (i, \text{Add} \ i \ e) \in \text{set} \) \( \text{node-deliver-messages} \) \( xs \) \( \) and \( (ir, \text{Rem} \ ir \ e) \in \text{set} \) \( \text{node-deliver-messages} \) \( xs \)
sells \text{hb} \( (i, \text{Add} \ i \ e) \) \( (ir, \text{Rem} \ ir \ e) \)
proof –
obtain \( \text{pre} \ k \) where \text{pre}(\text{Broadcast} \ (ir, \text{Rem } is \ e)] \) prefix of \( k \)
using \text{assms delivery-has-a-cause events-before-exist prefix-msg-in-history by blast}
morerover hence \text{Deliver} \( (i, \text{Add} \ i \ e) \in \text{set} \) \text{pre}
using \text{Broadcast-Deliver-prefix-closed assms(1) by auto}
ultimately show \( ?\text{thesis} \)
using \text{hb.intro}(2) \text{events-in-local-order by blast}
qed

lemma (in orset) Deliver-Add-same-id-same-message:
  assumes Deliver (i, Add i e1) ∈ set (history j) and Deliver (i, Add i e2) ∈ set (history j)
  shows e1 = e2
proof -
  obtain pre1 pre2 k1 k2 where *: pre1 @[Broadcast (i, Add i e1)] prefix of k1 pre2 @[Broadcast (i, Add i e2)] prefix of k2
    using assms delivery-has-a-cause events-before-exist by meson
  moreover hence Broadcast (i, Add i e1) ∈ set (history k1) Broadcast (i, Add i e2) ∈ set (history k2)
    using node-histories.prefix-to-carriers node-histories-axioms by force+
  ultimately show ?thesis
    using msg-id-unique by fastforce
qed

lemma (in orset) ids-imply-messages-same:
  assumes i ∈ is
    and xs prefix of j
    and (i, Add i e1) ∈ set (node-deliver-messages xs) and (ir, Rem is e2) ∈ set (node-deliver-messages xs)
  shows e1 = e2
proof -
  obtain pre k where pre @[Broadcast (ir, Rem is e2)] prefix of k
    using assms delivery-has-a-cause events-before-exist prefix-msg-in-history by blast
  moreover hence Deliver (i, Add i e2) ∈ set pre
    using Broadcast-Deliver-prefix-closed assms(1) by blast
  moreover have Deliver (i, Add i e1) ∈ set (history j)
    using assms(2) assms(3) prefix-msg-in-history by blast
  ultimately show ?thesis
    by (metis fst-conv msg-id-unique network.delivery-has-a-cause network-axioms operation.inject(1)
      prefix-elem-to-carriers prefix-of-appendD prod.inject)
qed

corollary (in orset) concurrent-add-remove-independent:
  assumes ¬ hb (i, Add i e1) (ir, Rem is e2) and ¬ hb (ir, Rem is e2) (i, Add i e1)
    and xs prefix of j
    and (i, Add i e1) ∈ set (node-deliver-messages xs) and (ir, Rem is e2) ∈ set (node-deliver-messages xs)
  shows i ∉ is
    using assms ids-imply-messages-same concurrent-add-remove-independent-technical by fastforce

lemma (in orset) rem-rem-commute:
  shows (Rem i1 eI1) ▷ (Rem i2 eI2) = (Rem i2 eI2) ▷ (Rem i1 eI1)
  by (unfold interpret-op-def op-elem-def kleisli-def, fastforce)

lemma (in orset) concurrent-operations-commute:
  assumes xs prefix of i
  shows hb.concurrent-ops-commute (node-deliver-messages xs)
proof -
  { fix a b x y
    assume (a, b) ∈ set (node-deliver-messages xs)
    (x, y) ∈ set (node-deliver-messages xs)
    hb.concurrent (a, b) (x, y)
    hence interp-msg (a, b) ▷ interp-msg (x, y) = interp-msg (x, y) ▷ interp-msg (a, b)
    apply (unfold interp-msg-def, case-tac b; case-tac y; simp add: add-add-commute rem-rem-commute
      hb.concurrent-def)
    apply (metis add-id-valid add-rem-commute assms concurrent-add-remove-independent hb.concurrentD1
    meson
}

hb.concurrentD2 prefix-contains-msg)
  thus \textit{thesis}
  by (fastforce simp: hb.concurrent-ops-commute-def)
qed

\textbf{theorem (in orset) convergence:}
\begin{itemize}
  \item \textbf{assumes} set (node-deliver-messages xs) = set (node-deliver-messages ys)
  \item and \hspace{1em} xs prefix of i and \hspace{1em} ys prefix of j
\end{itemize}
\textbf{shows} apply-operations xs = apply-operations ys
\textbf{using} assms by (auto simp add: apply-operations-def intro: hb.convergence-ext concurrent-operations-commute node-deliver-messages-distinct hb-consistent-prefix)

\textbf{context} orset \textbf{begin}

\textbf{sublocale} sec: strong-eventual-consistency weak-hb hb interp-msg
\lambda ops. \exists \, xs \, i. \, xs \, prefix \, of \, i \, \land \, node-deliver-messages \, xs \, = \, ops \, \lambda x.\{\}
\textbf{apply} (standard; clarsimp simp add: hb-consistent-prefix node-deliver-messages-distinct concurrent-operations-commute)
\textbf{apply} (metis (no-types, lifting) apply-operations-def bind.bind-lunit not-None-eq hb.apply-operations-Snoc kleisli-def apply-operations-never-fails interp-msg-def)
\textbf{using} drop-last-message \textbf{apply} blast
\textbf{done}

\textbf{end}

\textbf{end}

\textbf{References}


