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

July 7, 2017

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.
2.1 Kleisli arrow composition

**definition** kleisli :: 
'(b ⇒ 'b option) ⇒ ('b ⇒ 'b option) ⇒ ('b ⇒ 'b option)
(infixr ⊲ 65) where

\[ f ⊲ g \equiv \lambda x. (f x >>= (\lambda y. g y)) \]

**lemma** kleisli-comm-cong:
- assumes \( f ⊲ g = g ⊲ f \)
- shows \( z ⊲ x ⊲ y = z ⊲ y ⊲ x \)
  using assms by(clarsimp simp add: kleisli-def)

**lemma** kleisli-assoc:
- shows \( (z ⊲ x) ⊲ y = z ⊲ (x ⊲ y) \)
  by(auto simp add: kleisli-def)

2.2 Lemmas about sets

**lemma** distinct-set-notin [dest]:
- assumes distinct \((x \# xs)\)
- shows \( x \notin set xs \)
  using assms by(induction xs, auto)

**lemma** set-membership-equality-technicalD [dest]:
- assumes \( \{x\} \cup (set xs) = \{y\} \cup (set ys) \)
- shows \( x = y \lor y \in set xs \)
  using assms by(induction xs, auto)

**lemma** set-equality-technical:
- assumes \( \{x\} \cup (set xs) = \{y\} \cup (set ys) \)
  and \( x \notin set xs \)
  and \( y \notin set ys \)
  and \( y \in set xs \)
- shows \( \{x\} \cup (set xs - \{y\}) = set ys \)
  using assms by (induction xs) auto

**lemma** set-elem-nth:
- assumes \( x \in set xs \)
- shows \( \exists m. m < length xs \land xs ! m = x \)
  using assms by(induction xs, simp) (meson in-set-conv-nth)

2.3 Lemmas about list

**lemma** list-nil-or-snoc:
- shows \( xs = [] \lor (\exists y ys. xs = ys @ [y]) \)
  by (induction xs, auto)

**lemma** suffix-eq-distinct-list:
- assumes distinct \( xs \)
  and \( ys @ suf1 = xs \)
  and \( ys @ suf2 = xs \)
- shows \( suf1 = suf2 \)
  using assms by(induction xs arbitrary: suf1 suf2 rule: rev-induct, simp) (metis append-eq-append-conv)

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

using \( \text{assms} \)
apply\( (\text{induction \( xs \) arbitrary: pre1 pre2 ys, simp})\)
apply\( (\text{case-tac pre1}; \text{case-tac pre2}; \text{clarify})\)
apply\( (\text{metis suffix-eq-distinct-list append-Nil})\)
apply\( (\text{metis Un-iff append-eq-Cons-cons distinct.simps(2) list.set-intros(1) set-append suffix-eq-distinct-list})\)
apply\( (\text{metis Un-iff append-eq-Cons-cons distinct.simps(2) list.set-intros(1) set-append suffix-eq-distinct-list})\)
apply\( (\text{metis distinct.simps(2) hd-append2 list.sel(1) list.sel(3) list.simps(3) tl-append2})\)
done

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

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

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

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

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

lemma \( \text{list-nth-split:} \)
assumes \( m < \text{length} \ \text{cs} \)
and \( n < m \)
and \( 1 < \text{length} \ \text{cs} \)
shows \( \exists x s \ y s \ . \ \text{cs} = x @ (\text{cs} ! n) @ \text{ys} @ (\text{cs} ! m) @ \text{zs} \)
using \( \text{assms} \)
proof\( (\text{induction \( n \) arbitrary: \text{cs} \( m \})\)
\( \text{case 0 thus} \ \exists x s \ y s \ . \ \text{cs} = x @ [x] @ \text{exI}, \text{clarsimp} \)
apply\( (\text{case-tac cs; clarsimp})\)
apply\( (\text{rule-tac x=[] in \text{exI}, clarsimp})\)
4
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 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 clar simp
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*∥*y*, whenever one does not happen before the other: ¬(*x* ≺ *y*) and ¬(*y* ≺ *x*).

definition concurrent :: 'a ⇒ 'a ⇒ bool (infix || 50) where
s1 || s2 ≡ ¬ (s1 ≺ s2) ∧ ¬ (s2 ≺ s1)

lemma concurrentI [intro!, simp]: ¬ (s1 ≺ s2) ⇒ ¬ (s2 ≺ s1) ⇒ s1 || s2
by (auto simp: concurrent-def)

lemma concurrentD1 [dest]: s1 || s2 ⇒ ¬ (s1 ≺ s2)
by (auto simp: concurrent-def)

lemma concurrentD2 [dest]: s1 || s2 ⇒ ¬ (s2 ≺ s1)
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 ≡ ∀ 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 zs; \forall x \in set zs. \neg y < x ] \Rightarrow hb-consistent (xs @ [y])

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

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

lemma consistentI [intro!]:
assumes hb-consistent (xs @ ys)
and \forall z \in set (xs @ ys). \neg z < 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) (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-y-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 {

8
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 \( \exists \text{prefix} \text{ suffix}. \; \text{ys} @ [y] = \text{prefix} @ x \# \text{suffix} \land \text{concurrent-set x suffix} \)
    by force

} ultimately show \( \exists \text{prefix} \text{ suffix}. \; \text{ys} @ [y] = \text{prefix} @ x \# \text{suffix} \land \text{concurrent-set x suffix} \)
    by auto
qed

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

lemma hb-consistent-append-porder:
    assumes hb-consistent \( \text{xs} @ \text{ys} \)
    \( \text{x} \in \text{set xs} \)
    \( \text{y} \in \text{set ys} \)
    shows \( \neg \text{y} < \text{x} \)
    using assms by (induction \( \text{ys} \) arbitrary; \( \text{xs} \) 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 :: \( \text{'} \text{a list} \Rightarrow \text{'} \text{b} \Rightarrow \text{'} \text{b} \)
where
apply-operations es ≡ foldl (op ⊳) (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) ⊳ (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 :: \( \text{'} \text{a list} \Rightarrow \text{ bool } \)
where
    concurrent-ops-commute xs ≡
    \( \forall \text{x y} \; \{x, y\} \subseteq \text{set xs} \longrightarrow \text{concurrent x y} \longrightarrow \langle x \rangle \triangleright \langle y \rangle = \langle y \rangle \triangleright \langle x \rangle \)

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

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

lemma concurrent-ops-commute-concurrent-set:
assumes concurrent-ops-commute (prefix@[suffix@[x]])
  concurrent-set x suffix
  distinct (prefix @ x # suffix)
shows apply-operations (prefix @ suffix @ [x]) = apply-operations (prefix @ x # suffix)
using assms proof (induction suffix arbitrary: rule: rev-induct, force)
fix a xs
assume IH: concurrent-ops-commute (prefix @ xs @ [x]) \implies
  concurrent-set x xs \implies distinct (prefix @ x # xs) \implies
  apply-operations (prefix @ xs @ [x]) = apply-operations (prefix @ x # xs)
assume assms: concurrent-ops-commute (prefix @ (zs @ [a])@[x])
  concurrent-set x (zs @ [a]) distinct (prefix @ x # xs @ [a])
hence ac-comm: (a) \triangleright (x) = (x) \triangleright (a)
  by (clarsimp simp: concurrent-ops-commute-def) blast
have copc: concurrent-ops-commute (prefix @ xs @ [x])
  by (clarsimp simp: concurrent-ops-commute-def) blast
have apply-operations ((prefix @ x # xs) @ [a]) = (apply-operations (prefix @ x # xs)) \triangleright (a)
  by (simp del: append-assoc)
also have ... = (apply-operations (prefix @ xs @ [x])) \triangleright (a)
  using IH assms copc by auto
also have ... = ((apply-operations (prefix @ xs)) \triangleright (x)) \triangleright (a)
  by (simp add: append-assoc[symmetric] del: append-assoc)
also have ... = (apply-operations (prefix @ xs)) \triangleright ((a) \triangleright (x))
  using ac-comm kleisli-comm-cong kleisli-assoc by simp
finally show apply-operations (prefix @ (zs @ [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
  concurrent-ops-commute ys
  distinct xs
  distinct ys
  hb-consistent zs
  hb-consistent ys
shows 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 \land 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 yssplit by simp (meson 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 yssplit 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]) ⇒ apply-operations 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)
theorem sec-progress:
assumes op-history xs
shows apply-operations xs initial-state ≠ None
using assms proof(induction xs rule: rev-induct, simp)
case (snoc x xs)
  have apply-operations xs initial-state ≠ None
    using snoc.IH snoc.prems trunc-history kleisli-def bind-def by blast
  moreover have apply-operations (xs ⊕ [x]) = apply-operations xs ⊢ ⟨x⟩
    by simp
  ultimately show ?case
    using no-failure snoc.prems by (clarsimp simp add: kleisli-def split: bind-splits)
qed

end

end

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.

theory
  Network
imports
  Convergence
begin

4.1 Node histories

We model a distributed system as an unbounded number of communicating nodes. We assume nothing about the communication pattern of nodes—we assume only that each node is uniquely identified by a natural number, and that the flow of execution at each node consists of a finite, totally ordered sequence of execution steps (events). We call that sequence of events at node \( i \) the history of that node. For convenience, we assume that every event or execution step is unique within a node’s history.

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
  \( x ⊑^i z \equiv \exists xs ys zs. xs@x#ys@z#zs = history i \)

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 \(*::xs_1 @ e_1 @ ys_1 @ e_2 @ zs_1 = \text{history} \, i\)
  \(xs_2 @ e_2 @ ys_2 @ e_3 @ zs_2 = \text{history} \, i\)
using \text{history-order-def} \text{assms by} \text{auto}
hence \(xs_1 @ e_1 @ ys_1 @ e_2 @ zs_1 = \text{history} \, i\)
  \(ys_2 @ e_3 @ zs_2 = \text{history} \, i\)
by(\text{rule-tac} \, xs=\text{history} \, i \, \text{and} \, ys=\, e_2 \, \text{in pre-suf-eq-distinct-list}) \text{auto}
thus \(?\text{thesis}\)
by(\text{clarsimp simp: history-order-def}) \text{(metis \(*\, (2)\, \text{append.assoc append-Cons)}\)
qed

lemma (in node-histories) \text{local-order-carrier-closed}:
  \text{assumes} \, e_1 \sqsubseteq i \, e_2
  \text{shows} \, \{e_1, \, e_2\} \subseteq \text{set} (\text{history} \, i)
using \text{assms by} \text{(clarsimp simp add: history-order-def)}
\text{(metis set-conv-decomp Un-iff Un-subset-iff list.simps(15) set-append set-subset-Cons)}

lemma (in node-histories) \text{node-total-order-irrefl}:
  \text{shows} \, \neg (e \sqsubseteq i \, e)
by(\text{clarsimp simp add: history-order-def})
\text{(metis Un-iff histories-distinct distinct-append distinct-set-notin list.set-intros(1) set-append)}

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

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

definition (in node-histories) \text{prefix-of-node-history :: 'evt list \Rightarrow nat \Rightarrow bool (infix prefix of 50)} where \(xs\) \text{prefix of} \, i \equiv \exists \, y_0. \, xs@y_0 = \text{history} \, i\)

lemma (in node-histories) \text{carriers-head-ll}:
  \text{assumes} \, y \# y_0 = \text{history} \, i\n  \text{shows} \, \neg (x \sqsubseteq i \, y)
using \text{assms}
apply(\text{clarsimp simp add: history-order-def})
apply(\text{rename-tac} \, xs_1 \, ys_1 \, zs_1)
apply(\text{subgoal-tac} \, xs_1 \, @ \, x \# y_0 = [] \land zs_1 = y_0)
apply \text{clarsimp}
apply(\text{rule-tac} \, xs=\text{history} \, i \, \text{and} \, ys=[y] \text{in pre-suf-eq-distinct-list})
apply \text{auto}
done

lemma (in node-histories) \text{prefix-of-ConsD [dest]}:
  \text{assumes} \, x \neq x_0 \text{ prefix of} \, i
  \text{shows} \, [x] \text{ prefix of} \, i
using \text{assms by (auto simp: prefix-of-node-history-def)}

lemma (in node-histories) \text{prefix-of-appendD [dest]}:
  \text{assumes} \, xs \oplus ys \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 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 - 
have \(\exists \text{idx } \text{idx} < \text{length } (\text{history } i) \wedge (\text{history } i) ![\text{idx}] = x\)
using \(assms\) by (simp add: set-elem-nth)
thus \(?thesis\)
using \(assms\) 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 \(\exists \text{idz. } \text{idz} < \text{length } (\text{history } i) \wedge (\text{history } i) ![\text{idz}] = x\)
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 ⊑ i 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.

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) ] \implies
  \exists j. Broadcast m ∈ set (history j)

  and deliver-locally: [ Broadcast m ∈ set (history i) ] \implies
  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 ] \implies 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 msg-ids
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 \exists j. Broadcast m ⊑ j 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 ∉ set (history j)
Based on the well-known definition by [8], we say that \( m1 \prec m2 \) if any of the following is true:

1. \( m1 \) and \( m2 \) were broadcast by the same node, and \( m1 \) was broadcast before \( m2 \).
2. The node that broadcast \( m2 \) had delivered \( m1 \) before it broadcast \( m2 \).
3. There exists some operation \( m3 \) such that \( m1 \prec m3 \) and \( m3 \prec m2 \).

\[
\begin{align*}
\text{inductive (in network)} & \quad \text{hb} :: \text{msg} \Rightarrow \text{msg} \Rightarrow \text{bool} \\
\text{hb-broadcast:} & \quad \parallel \text{Broadcast} m1 \sqsupset \text{Broadcast} m2 \parallel \Rightarrow \text{hb} m1 m2 \\
\text{hb-deliver:} & \quad \parallel \text{Deliver} m1 \sqsupset \text{Broadcast} m2 \parallel \Rightarrow \text{hb} m1 m2 \\
\text{hb-trans:} & \quad \parallel \text{hb} m1 m2; \text{hb} m2 m3 \parallel \Rightarrow \text{hb} m1 m3
\end{align*}
\]

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

\[
\begin{align*}
\text{definition (in network)} & \quad \text{weak-hb} :: \text{msg} \Rightarrow \text{msg} \Rightarrow \text{bool} \\
\text{weak-hb m1 m2} & \equiv \text{hb} m1 m2 \lor \text{m1} = \text{m2}
\end{align*}
\]

\[
\begin{align*}
\text{locale causal-network} = & \quad \text{network +} \\
\text{assumes causal-delivery: Deliver} m2 \in \text{set (history j)} & \Rightarrow \text{hb} m1 m2 \Rightarrow \text{Deliver} m1 \sqsupset \text{Deliver} m2
\end{align*}
\]

\[
\begin{align*}
\text{lemma (in causal-network) causal-broadcast:} & \quad \text{assumes Deliver} m2 \in \text{set (history j)} \\
& \quad \text{and} \quad \text{Deliver} m1 \sqsupset \text{Broadcast} m2 \\
& \quad \text{shows} \quad \text{Deliver} m1 \sqsupset \text{Deliver} m2 \\
& \quad \text{using} \quad \text{assms causal-delivery hb.intros(2) by blast}
\end{align*}
\]

\[
\begin{align*}
\text{lemma (in network) hb-broadcast-exists1:} & \quad \text{assumes hb m1 m2} \\
& \quad \text{shows} \exists i. \text{Broadcast} m1 \in \text{set (history i)} \\
& \quad \text{using} \quad \text{assms} \\
& \quad \text{apply (induction rule: hb.induct)}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{apply (meson insert-subset node-histories.local-order-carrier-closed node-histories-axioms)} \\
& \quad \text{apply (meson delivery-has-a-cause insert-subset local-order-carrier-closed)} \\
& \quad \text{apply simp} \\
& \quad \text{done}
\end{align*}
\]

\[
\begin{align*}
\text{lemma (in network) hb-broadcast-exists2:} & \quad \text{assumes hb m1 m2} \\
& \quad \text{shows} \exists i. \text{Broadcast} m2 \in \text{set (history i)} \\
& \quad \text{using} \quad \text{assms} \\
& \quad \text{apply (induction rule: hb.induct)}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{apply (meson insert-subset node-histories.local-order-carrier-closed node-histories-axioms)} \\
& \quad \text{apply (meson delivery-has-a-cause insert-subset local-order-carrier-closed)} \\
& \quad \text{apply simp} \\
& \quad \text{done}
\end{align*}
\]

\[
\begin{align*}
\text{lemma (in causal-network) hb-has-a-reason:} & \quad \text{assumes hb m1 m2} \\
& \quad \text{and} \quad \text{Broadcast} m2 \in \text{set (history i)} \\
& \quad \text{shows} \quad \text{Deliver} m1 \in \text{set (history i)} \lor \text{Broadcast} m1 \in \text{set (history i)} \\
& \quad \text{using} \quad \text{assms} \\
& \quad \text{apply (induction rule: hb.induct)}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{apply (metis insert-subset local-order-carrier-closed network.broadcasts-unique network-axioms)} \\
& \quad \text{apply (metis insert-subset local-order-carrier-closed network.broadcasts-unique network-axioms)} \\
& \quad \text{using hb-trans causal-delivery local-order-carrier-closed apply blast}
\end{align*}
\]

4.3 Causal networks

\[
\begin{align*}
\text{lemma (in causal-network) hb-has-a-reason:} & \quad \text{assumes hb m1 m2} \\
& \quad \text{and} \quad \text{Broadcast} m2 \in \text{set (history i)} \\
& \quad \text{shows} \quad \text{Deliver} m1 \in \text{set (history i)} \lor \text{Broadcast} m1 \in \text{set (history i)} \\
& \quad \text{using} \quad \text{assms} \\
& \quad \text{apply (induction rule: hb.induct)}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{apply (metis insert-subset local-order-carrier-closed network.broadcasts-unique network-axioms)} \\
& \quad \text{apply (metis insert-subset local-order-carrier-closed network.broadcasts-unique network-axioms)} \\
& \quad \text{using hb-trans causal-delivery local-order-carrier-closed apply blast}
\end{align*}
\]

\[
\begin{align*}
\text{lemma (in causal-network) hb-has-a-reason:} & \quad \text{assumes hb m1 m2} \\
& \quad \text{and} \quad \text{Broadcast} m2 \in \text{set (history i)} \\
& \quad \text{shows} \quad \text{Deliver} m1 \in \text{set (history i)} \lor \text{Broadcast} m1 \in \text{set (history i)} \\
& \quad \text{using} \quad \text{assms} \\
& \quad \text{apply (induction rule: hb.induct)}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{apply (metis insert-subset local-order-carrier-closed network.broadcasts-unique network-axioms)} \\
& \quad \text{apply (metis insert-subset local-order-carrier-closed network.broadcasts-unique network-axioms)} \\
& \quad \text{using hb-trans causal-delivery local-order-carrier-closed apply blast}
\end{align*}
\]
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.intro(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.intro(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.intro(1) hb-irrefl network hb.intro(3) network-axioms)
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
(erule hb-elim)
show (\A ia. Broadcast m1 ⊑ i Broadcast m2 \implies Broadcast m2 ⊑' i a Broadcast m1 \implies False
by (metis broadcasts-unique insert-subset local-order-carrier-closed node-total-order-irrefl node-total-order-trans)
next
show (\A ia. Broadcast m1 ⊑ i Broadcast m2 \implies Deliver m2 ⊑' i a Broadcast m1 \implies False
by (metis broadcast-before-delivery broadcasts-unique insert-subset local-order-carrier-closed node-total-order-irrefl
node-total-order-trans)
next
show (\A m2a. Broadcast m1 ⊑ i Broadcast m2 \implies hb m2 m2a \implies hb m2a m1 \implies 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
(erule hb-elim)
show (\A ia. Deliver m1 ⊑ i Broadcast m2 \implies Deliver m2 ⊑' i a Broadcast m1 \implies False
by (metis broadcast-before-delivery broadcasts-unique insert-subset local-order-carrier-closed node-total-order-irrefl
node-total-order-trans)
next
show (\A ia. Deliver m1 ⊑ i Broadcast m2 \implies Deliver m2 ⊑' i a Broadcast m1 \implies 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 (\A m2a. Deliver m1 ⊑ i Broadcast m2 \implies hb m2 m2a \implies hb m2a m1 \implies 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 \implies False) (hb m3 m2 \implies False)
thus False
using hb.intros(3) by blast
qed

definition (in network) node-deliver-messages :: 'msg event list \Rightarrow 'msg list
where
node-deliver-messages cs \equiv List.map-filter (\A e. case Deliver m \Rightarrow Some m | - \Rightarrow 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 (auto simp add: node-deliver-messages-def map-filter-def)

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

lemma (in network) node-deliver-messages-Broadcast [simp]:
  shows node-deliver-messages [Broadcast m] = []
by (clarsimp simp: node-deliver-messages-def map-filter-def)

lemma (in network) node-deliver-messages-Deliver [simp]:
  shows node-deliver-messages [Deliver m] = [m]
by (clarsimp simp: node-deliver-messages-def map-filter-def)

lemma (in network) prefix-msg-in-history:
  assumes es prefix of i
  and m ∈ set (node-deliver-messages es)
  shows Deliver m ∈ set (history i)
using assms prefix-to-carriers by (fastforce simp: node-deliver-messages-def map-filter-def split: event.split_asm)

lemma (in network) prefix-contains-msg:
  assumes es prefix of i
  and m ∈ set (node-deliver-messages es)
  shows Deliver m ∈ set es
using assms by (auto simp: node-deliver-messages-def map-filter-def split: event.split_asm)

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

lemma (in network) drop-last-message:
  assumes evts prefix of i
  and node-deliver-messages evts = msgs @ [last-msg]
  shows ∃pre. pre prefix of i ∧ node-deliver-messages pre = msgs
proof
  have Deliver last-msg ∈ set evts
    using assms network.prefix-contains-msg network-axioms by force
  then obtain idx where ∗: idx < length evts evts ! idx = Deliver last-msg
  by (meson set-elem-nth)
  then obtain pre suf where evts = pre @ (evts ! idx) # suf
  using id-take-nth-drop by blast
  hence ∗∗: evts = pre @ (Deliver last-msg) # suf
  using assms ∗ by auto
  moreover hence distinct (node-deliver-messages [(Deliver last-msg) @ suf])
  by (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 cs ≡ map snd (node-deliver-messages cs)

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 (zs @ [Broadcast m]) = apply-operations zs
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 (zs @ [Deliver m]) = (apply-operations zs ⇒ 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 \( n \cdot m < \text{length} \ cs \implies n < m \implies \text{cs ! n} \sqsubseteq \text{cs ! m} \)
shows \( \text{hb.hh-consistent} (\text{node-deliver-messages cs}) \)
using \textit{assms proof} (induction cs rule: rev-induct)

\textbf{case} Nil \textbf{thus} \texttt{?case}
  \textbf{by} (simp add: node-deliver-messages-def hb hb-consistent_intros(1) map-filter-simps(2))

\textbf{next} case (snoc \texttt{x} \texttt{xs})
  \textbf{hence} \texttt{∗}: \(∀ m n. m < \text{length} \texttt{xs} \implies n < m \implies \texttt{xs} ! n \sqsubseteq \text{xs} ! m\)
  \textbf{by} (-, erule-tac \texttt{x} = \texttt{m} in meta-allE, erule-tac \texttt{x} = \texttt{n} in meta-allE, clarsimp simp add: nth-append)

\textbf{then show} \texttt{?case}
  \textbf{proof} (cases \texttt{x})
  case (Broadcast \texttt{x}1)
  \textbf{thus} \texttt{?thesis}
    \textbf{using} \texttt{∗} by (clarsimp simp add: node-deliver-messages-append)

\textbf{next}
  case (Deliver \texttt{x}2)
  \textbf{thus} \texttt{?thesis}
    \textbf{using} \texttt{∗} apply (clarsimp simp add: node-deliver-messages-def map-filter-def map-filter-append)

  \textbf{apply} (rename-tac \texttt{m} \texttt{m1} \texttt{m2})
  \textbf{apply} (case-tac \texttt{m}; clarsimp)
  \textbf{apply} (drule set-elem-nth, erule exE, erule conjE)
  \textbf{apply} (clarsimp simp add: nth-append)
  \textbf{by} (metis causal-delivery insert-subset node-histories.local-order-carrier-closed
  node-histories-axioms node-total-order-antisym)

\textbf{q}e\textbf{d}
\textbf{q}\textbf{e}\textbf{d}

\textbf{corollary} (in \textit{network-with-ops})
  \textbf{shows} \texttt{hb hb-consistent (node-deliver-messages (history \textit{i}))}
  \textbf{by} (metis hb-consistent-technical history-order-def less-one linorder-neqE-nat list-nth-split zero-order(3))

\textbf{lemma} (in \textit{network-with-ops}) \textbf{hb-consistent-prefix}:
  \textbf{assumes} \texttt{xs prefix of \textit{i}}
  \textbf{shows} \texttt{hb hb-consistent (node-deliver-messages \texttt{xs})}
  \textbf{using} \textit{assms proof} (clarsimp simp: prefix-of-node-history-def, rule-tac \texttt{i} = \texttt{i} in hb-consistent-technical)

  \textbf{fix} \texttt{m n ys \textbf{assume} \texttt{∗}: \texttt{xs @ ys = history \textit{i} m < \text{length} \texttt{xs} n < m}}

  \textbf{consider} (a) \texttt{xs = []} | (b) \texttt{∃ c. xs = [c]} | (c) \texttt{Suc 0 < \text{length} \texttt{xs}}

  \textbf{by} (metis Suc-pred length-Suc-conv length-greater-0-conv zero-less-diff)

  \textbf{thus} \texttt{xs ! n \sqsubseteq \text{xs} ! m}

  \textbf{proof} (cases)
    \textbf{case a thus} \texttt{?thesis}
      \textbf{using} \texttt{∗} by clarsimp

  \textbf{next}
    \textbf{case b thus} \texttt{?thesis}
      \textbf{using} \textit{assms} \texttt{∗} by clarsimp

  \textbf{next}
    \textbf{case c thus} \texttt{?thesis}
      \textbf{using} \textit{assms} \texttt{∗} apply clarsimp

      \textbf{apply} (drule list-nth-split, assumption, clarsimp simp: c)

      \textbf{apply} (metis append.assoc append.simps(2) history-order-def)

  \textbf{done}

\textbf{q}e\textbf{d}
\textbf{q}e\textbf{d}

\textbf{locale} \textit{network-with-constrained-ops} = \textit{network-with-ops} +
\textbf{fixes} \textit{valid-msg} :: \texttt{′c ⇒ ′a × ′b ⇒ bool}
\textbf{assumes} broadcast-only-valid-msgs: \texttt{pre @ [Broadcast m]} prefix of \texttt{i} \implies
  \exists \textit{state}. apply-operations \texttt{pre} = \texttt{Some state ∧ valid-msg state m}

\textbf{lemma} (in \textit{network-with-constrained-ops}) broadcast-is-valid:
  \textbf{assumes} Broadcast \texttt{m} \in \textit{set} (history \textit{i})

21
shows \( \exists \) 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 \( \in \) set (history i)
shows \( \exists \) j pre state. pre \( \parallel \) [Broadcast m] prefix of j \( \land \) apply-operations pre = Some state \( \land \) 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 \( \in \) set xs
shows \( \exists \) 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 \( \lambda \) m. []
by standard auto

interpretation trivial-network: network \( \lambda \) m. [] id
by standard auto

interpretation trivial-causal-network: causal-network \( \lambda \) m. [] id
by standard auto

interpretation trivial-network-with-ops: network-with-ops \( \lambda \) m. [] (\( \lambda \) x y. Some y) 0
by standard auto

interpretation trivial-network-with-constrained-ops: network-with-constrained-ops \( \lambda \) m. [] (\( \lambda \) x y. Some y) 0 \( \lambda \) x 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 \times \ 'v \times 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
   insert xs e (Some i) >>= (λt. 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 >>= (λt. 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-None)

lemma index-not-in-delete-None [simp]:
  assumes i /∈ set xs
  shows delete xs i = None
using assms by (induction xs)

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)

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 delete-preserve-indices by blast

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)

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)

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 \( \text{fst } e_1 \neq \text{fst } e_2 \)
  \( i_1 = \text{None } \lor \ i_1 \neq \text{Some } (\text{fst } e_2) \)
  \( i_2 = \text{None } \lor \ i_2 \neq \text{Some } (\text{fst } e_1) \)
shows  \( \text{insert } x s e_1 i_1 \gg (\lambda y s. \text{insert } y s e_2 i_2) = \text{insert } x s e_2 i_2 \gg (\lambda y s. \text{insert } y s e_1 i_1) \)
using assms proof(induction rule: insert.induct)
fix \( xs \) and \( e :: (a, 'b) elt \)
assume \( i_2 = \text{None } \lor \ i_2 \neq \text{Some } (\text{fst } e) \) and \( \text{fst } e \neq \text{fst } e_2 \)
thus \( \text{insert } x s e \text{ None } \gg (\lambda y s. \text{insert } y s e_2 i_2) = \text{insert } x s e_2 i_2 \gg (\lambda y s. \text{insert } y s e \text{ None }) \)
  by (auto simp add: insert-body-commutes intro: insert-insert-body)
next
fix \( i \) and \( e :: (a, 'b) elt \)
assume \( \text{fst } e \neq \text{fst } e_2 \) and \( i_2 = \text{None } \lor \ i_2 \neq \text{Some } (\text{fst } e) \) and \( \text{Some } i = \text{None } \lor \ i \neq \text{Some } (\text{fst } e_2) \)
thus \( \text{insert } [] e \text{ (Some } i) \gg (\lambda y s. \text{insert } y s e_2 i_2) = \text{insert } [] e_2 i_2 \gg (\lambda y s. \text{insert } y s e \text{ (Some } i)) \)
  by (auto intro: insert-nil-none[symmetric])
next
fix \( xs \ i \) and \( x e :: (a, 'b) elt \)
assume \( \text{IH}: (\text{fst } x \neq i \implies \text{fst } e \neq \text{fst } e_2 \implies \text{Some } i = \text{None } \lor \ i \neq \text{Some } (\text{fst } e_2) \implies i_2 = \text{None } \lor \ i_2 \neq \text{Some } (\text{fst } e)) \implies \text{insert } x s e \text{ (Some } i) \gg (\lambda y s. \text{insert } y s e_2 i_2) = \text{insert } x s e_2 i_2 \gg (\lambda y s. \text{insert } y s e \text{ (Some } i)) \)
  apply -
  apply(rule disjE, clarsimp, simp, rule conjI)
  apply(case-tac \( i_2 \); force simp add: insert-body-commutes insert-insert-body-commute)
  apply(case-tac \( i_2 \); clarsimp cong: Option.bind-cong simp add: insert-insert-body split: bind-splits)
  apply force
  done
qed

lemma delete-commutes:
shows \( \text{delete } x s i_1 \gg (\lambda y s. \text{delete } y s i_2) = \text{delete } x s i_2 \gg (\lambda y s. \text{delete } y s i_1) \)
by(induction \( xs \), auto split: bind-splits if-split-asm)

lemma insert-body-delete-commute:
assumes \( i_2 \neq \text{fst } e \)
shows  \( \text{delete } (\text{insert-body } x s e) i_2 \gg (\lambda t. \text{Some } (x \# t)) = \text{delete } x s i_2 \gg (\lambda y. \text{Some } (x \# \text{insert-body } y e)) \)
using assms by (induction \( xs \); cases \( e \); cases \( i_1 \); auto split: bind-splits if-split-asm simp add: insert-body-delete-commute)

lemma insert-delete-commute:
assumes \( i_2 \neq \text{fst } e \)
shows \( \text{insert } x s e i_1 \gg (\lambda y s. \text{delete } y s i_2) = \text{delete } x s i_2 \gg (\lambda y s. \text{insert } y s e i_1) \)
using assms by(induction \( xs \); cases \( e \); cases \( i_1 \); 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] |
\[
\begin{align*}
\text{insert'} \; [] \; e & \quad (\text{Some } i) = \text{None} \\
\text{insert'} \; (x \# xs) \; e \; \text{None} & = \\
& \quad (\text{if } \text{fst} \; x < \text{fst} \; e \; \text{then} \\
& \quad \quad \text{Some} \; (e \# x \# xs) \\
& \quad \text{else} \\
& \quad \quad \text{case } \text{insert'} \; xs \; e \; \text{None} \; \text{of} \\
& \quad \quad \quad \text{None} \quad \Rightarrow \quad \text{None} \\
& \quad \quad \quad \text{Some} \; t \quad \Rightarrow \quad \text{Some} \; (x \# t) \\
\text{insert'} \; (x \# xs) \; e \; (\text{Some } i) & = \\
& \quad (\text{if } \text{fst} \; x = i \; \text{then} \\
& \quad \quad \text{case } \text{insert'} \; xs \; e \; \text{None} \; \text{of} \\
& \quad \quad \quad \text{None} \quad \Rightarrow \quad \text{None} \\
& \quad \quad \quad \text{Some} \; t \quad \Rightarrow \quad \text{Some} \; (x \# t) \\
& \quad \text{else} \\
& \quad \quad \text{case } \text{insert'} \; xs \; e \; (\text{Some } i) \; \text{of} \\
& \quad \quad \quad \text{None} \quad \Rightarrow \quad \text{None} \\
& \quad \quad \quad \text{Some} \; t \quad \Rightarrow \quad \text{Some} \; (x \# t)
\end{align*}
\]

\textbf{lemma [elim!, dest]:} \\
\textbf{assumes} \text{insert'} \; xs \; e \; \text{None} = \text{None} \\
\textbf{shows} \quad \text{False} \\
\textbf{using} \; \text{assms by} \; (\text{induction } xs, \; \text{auto} \; \text{split}: \; \text{if-split-asn} \; \text{option.split-asn})

\textbf{lemma} \; \text{insert-body-insert'}: \\
\textbf{shows} \; \text{insert'} \; xs \; e \; \text{None} = \text{Some} \; (\text{insert-body } xs \; e) \\
\textbf{by} \; (\text{induction } xs, \; \text{auto})

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

\textbf{lemma} \; \text{insert-body-stop-iteration}: \\
\textbf{assumes} \; \text{fst} \; e > \text{fst} \; x \\
\textbf{shows} \; \text{insert-body} \; (x \# xs) \; e = e \# x \# xs \\
\textbf{using} \; \text{assms by} \; \text{simp}

\textbf{lemma} \; \text{insert-body-contains-new-elem}: \\
\textbf{shows} \; \exists \; p \; s. \; \text{xs} = p \; @ \; s \land \; \text{insert-body} \; xs \; e = p \; @ \; e \; \# \; s \\
\textbf{proof} \; (\text{induction } xs) \\
\textbf{case Nil} \; \text{thus} \; ?\text{case} \; \text{by force} \\
\textbf{next} \\
\textbf{case} \; (\text{Cons } a \; xs) \\
\textbf{then obtain} \; p \; s \; \text{where} \; \text{xs} = p \; @ \; s \land \; \text{insert-body} \; xs \; e = p \; @ \; e \; \# \; s \; \text{by force} \\
\textbf{thus} \; ?\text{case} \\
\quad \text{apply} \; \text{clarsimp} \\
\quad \text{apply} \; (\text{rule conjI}; \; \text{clarsimp}) \\
\quad \text{apply} \; \text{force} \\
\quad \text{apply} \; (\text{rule-tac } x=a \; \# \; p \; \text{in} \; \text{exI}, \; \text{force}) \\
\textbf{done}

\textbf{qed}

\textbf{lemma} \; \text{insert-between-elements}: \\
\textbf{assumes} \; \text{xs} = \text{pre}@\text{ref} \; \# \; \text{suf} \\
\quad \text{and} \; \text{distinct} \; (\text{map} \; \text{fst} \; \text{xs}) \\
\quad \quad \text{and} \; \forall \; i', \; i' \in \text{fst} \; \text{set} \; \text{xs} \; \Rightarrow \; i' < \text{fst} \; e \\
\textbf{shows} \; \text{insert} \; \text{xs} \; e \; (\text{Some} \; (\text{fst} \; \text{ref})) = \text{Some} \; (\text{pre} \; @ \; \text{ref} \; \# \; e \; \# \; \text{suf}) \\
\textbf{using} \; \text{assms by} \; (\text{induction } \text{xs} \; \text{arbitrary: pre ref suf, force}) \; (\text{case-tac pre}; \; \text{case-tac suf}; \; \text{force})
lemma insert-position-element-technical:
assumes \( \forall x \in \text{set as}. \ a \neq \text{fst} \ x \)
and insert-body \((\text{as} \oplus \text{ds}) \ e = \text{cs} \oplus e \# ds\)
shows \( \text{insert} \ (\text{as} \oplus (a, aa, b) \# \text{cs} \oplus \text{ds}) \ e \ (\text{Some} \ a) = \text{Some} \ (\text{as} \oplus (a, aa, b) \# \text{cs} \oplus e \# ds) \)
using \text{assms by} \ (\text{induction as arbitrary: cs ds; clarsimp})

lemma split-tuple-list-by-id:
assumes \((a, b, c) \in \text{set xs}\)
and distinct \((\text{map} \ \text{fst} \ \text{xs})\)
shows \( \exists \text{pre suf. xs} = \text{pre} \oplus (a, b, c) \# \text{suf} \wedge (\forall y \in \text{set pre}. \ \text{fst} \ y \neq a) \)
using \text{assms proof}(\text{induction xs, clarsimp})
\begin{itemize}
\item case \((\text{Cons x xs})\)
\begin{itemize}
\item assume \( x \neq (a, b, c) \)
\item hence \((a, b, c) \in \text{set xs} \) distinct \((\text{map} \ \text{fst} \ \text{xs})\)
\item using \text{Cons.prem by force+}
\item then obtain \text{pre suf where xs} = \text{pre} \oplus (a, b, c) \# \text{suf} \wedge (\forall y \in \text{set pre}. \ \text{fst} \ y \neq a)
\item using \text{Cons.IH by force}
\item hence case
\item apply\((\text{rule-tac x=x#pre in exI})\)
\item using \text{Cons.prem(2) by auto}
\end{itemize}
\end{itemize}
\} thus case
by force
qed

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

5.6 Network

theory
RGA
imports
Network
Ordered-List
begin

datatype \('id, 'v) operation = 
Insert ('id, 'v) elt 'id option | 
Delete 'id
fun interpret-opers :: ('id::linorder, 'v) operation ⇒ ('id, 'v) elt list ⇒ ('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 Delver (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

28
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)
by (metis delete-preserve-indices prefix-of-appendD)
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
    by (metis (no-types) assms prefix-of-appendD rga.apply-ops-idx-elems rga-axioms)
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
proof (induction es arbitrary: xs rule: rev-induct)
  case (snoc x xs ys)
  thus ?case
    proof (cases x)
      case (Broadcast e)
      thus ?thesis
        using snoc by (clarsimp, blast)
    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)
      next
      case (Delete e)
      moreover then obtain xs' where *: apply-operations xs = Some xs'
        using snoc by (clarsimp, blast, simp add: interp-msg-def)
      ultimately show ?thesis
        using assms snoc proof (cases cl)
      next
    next
  qed

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
  next
    case (Broadcast e)
    thus ?thesis
      using snoc by (clarsimp, blast)
  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)
    next
    case (Insert cl -)
      thus ?thesis
        using snoc proof (cases cl)
    next
    case (Delete -)
      thus ?thesis
        using snoc proof (clarsimp)
  qed

29
lemma (in rga) insert-in-apply-set:
  assumes es @ [Deliver (i, Insert e (Some a))] prefix of j
  and Deliver (i', Insert e' n) ∈ set es
  and apply-operations es = Some s
  shows fst e' ∈ element-ids s
using assms apply-ops-idx-elems idx-in-elem-inserted prefix-of-appendD by blast

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

lemma (in rga) allowed-insert:
  assumes Broadcast (i, Insert e n) ∈ set (history j)
  shows n = None ∨ (∃i' e' n'. n = Some (fst e') ∧ Deliver (i', Insert e' n') ⊑ Broadcast (i, Insert e n))
proof –
  obtain pre where 1: pre @ [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 n = None ∨ (∃i' e' n'. n = Some (fst e') ∧ Deliver (i', Insert e' n') ⊑ Broadcast (i, Insert e n))
  proof(cases n)
    fix a
    assume 4: n = Some a
    hence a ∈ element-ids state and 5: fst e = i
    using 3 by(clarsimp simp add: valid-rga-msg-def)+
    from this have ∃i' e' f' n'. Deliver (i', Insert (a, v', f') n') ∈ set pre
    using deliver-insert-exists 2 1 by blast
    thus n = None ∨ (∃i' e' n'. n = Some (fst e') ∧ Deliver (i', Insert e' n') ⊑ Broadcast (i, Insert e n))
    using events-in-local-order 1 4 5 by(metis fst-conv)
  qed simp
qed

lemma (in rga) allowed-delete:
  assumes Broadcast (i, Delete x) ∈ set (history j)
  shows ∃i' n' v b. Deliver (i', Insert (x, v, b) n') ⊑ Broadcast (i, Delete x)
proof –
  obtain pre where 1: pre @ [Broadcast (i, Delete x)] prefix of j
  using assms events-before-exist by blast
  from this obtain state where 2: apply-operations pre = Some state
  and valid-rga-msg state (i, Delete x)
  using broadcast-only-valid-msgs by blast
  hence x ∈ element-ids state
  using apply-ops-idx-elems by(simp add: valid-rga-msg-def)
  hence ∃i' v' f' n'. Deliver (i', Insert (x, v', f') n') ∈ set pre
using deliver-insert-exists 1 2 by blast
thus \( \exists i', n' \in b. \text{Deliver} (i', \text{Insert} (x, v, b) n') \sqsubseteq \text{Broadcast} (i, \text{Delete} x) \)
using events-in-local-order 1 by blast
qed

lemma (in rga) insert-id-unique:
  assumes fst e1 = fst e2
  and \text{Broadcast} (i1, \text{Insert} e1 n1) \in set \text{(history} i)
  and \text{Broadcast} (i2, \text{Insert} e2 n2) \in set \text{(history} j)
  shows \text{Insert} e1 n1 = \text{Insert} e2 n2
using assms insert-msg-id msg-id-unique Pair-inject fst-conv

by (meson assms Un-insert-right insert-subset list.simps)

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

lemma (in rga) allowed-delete-deliver-in-set:
  assumes \text{(es} 3\{\text{Deliver} (i, \text{Delete} m)\}) \text{prefix of} j
  shows \( \exists i' \in b. \text{Deliver} (i', \text{Insert} (m, v, b) n) \in set \text{es} \)
by (metis (no-types, lifting) Un-insert-right insert-iff list.simps(15) assms
  local-order-prefix-closed-last rga.allowed-prefix-closed-last 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 \( n = \text{None} \lor (\exists i' \in n' \sqsubseteq \text{v} b. n = \text{Some} n' \land \text{Deliver} (i', \text{Insert} (n', v, b) n') \sqsubseteq \text{Broadcast} (i, \text{Delete} x) \)
proof (cases \text{n})
  fix a
  assume 3: \( n = \text{Some} a \)
  from this obtain \( i' \in e' n' \text{ where} 4: \text{Some} a = \text{Some} \{\text{fst} e'\} \text{ and} \)
  2: \( \text{Deliver} (i', \text{Insert} e' n') \sqsubseteq \text{Broadcast} (i, \text{Insert} e (\text{Some} a)) \)
using allowed-insert 1 by blast
  hence \text{Deliver} (i', \text{Insert} e' n') \in set \text{(history} ja) \and \text{Broadcast} (i, \text{Insert} e (\text{Some} a)) \in set \text{(history} ja)
  using local-order-carrier-closed by simp+
  from this obtain \text{ja} where \text{Broadcast} (i, \text{Insert} e (\text{Some} a)) \in set \text{(history} ja)
  using delivery-has-a-cause by simp
  have \( \exists i' \in n' n'' \sqsubseteq v b. n = \text{Some} n' \land \text{Deliver} (i', \text{Insert} (n', v, b) n'') \sqsubseteq \text{Broadcast} (i, \text{Insert} e n) \)
using 2 3 \And \text{by}(metis assms causal-broadcast prod.collapse)
  thus \( n = \text{None} \lor (\exists i' \in n' n'' \sqsubseteq v b. n = \text{Some} n' \land \text{Deliver} (i', \text{Insert} (n', v, b) n'') \sqsubseteq \text{Broadcast} (i, \text{Insert} e n) \)
  by auto
  qed simp

qed

lemma (in rga) allowed-insert-deliver-in-set:
  assumes \text{(es} 3\{\text{Deliver} (i, \text{Insert} e m)\}) \text{prefix of} j
  shows \( m = \text{None} \lor (\exists i' \in m' n v b. m = \text{Some} m' \land \text{Deliver} (i', \text{Insert} (m', v, b) n) \in set \text{es}) \)
by (metis assms Un-insert-right insert-subset list.simps(15) set-append prefix-to-carriers
  allowed-insert-deliver local-order-prefix-closed-last)

lemma (in rga) Insert-no-failure:
assumes es @ [Deliver (i, Insert e n)] prefix of j and apply-operations es = Some s shows ∃ ys. insert s e n = Some ys by (metis (no-types, lifting) element-ids-def allowed-insert-deliver-in-set assms fst-conv insert-in-apply-set insert-no-failure set-map)

lemma (in rga) delete-no-failure: assumes es @ [Deliver (i, Delete n)] prefix of j and apply-operations es = Some s shows ∃ ys. delete s n = Some ys by obtain i' na v b where 1: Deliver (i', Insert (n, v, b) na) ∈ set es using assms allowed-delete-deliver-in-set by blast also have fst (n, v, b) ∈ set (indices es) using assms idx-in-elem-inserted calculation by blast from this assms and 1 show ∃ ys. delete s n = Some ys apply − apply (rule delete-no-failure) apply (metis apply-opers-idx-elems element-ids-def prefix-of-appendD prod.sel(1) set-map) done qed

lemma (in rga) Insert-equal: assumes fst e1 = fst e2 and Broadcast (i1, Insert e1 n1) ∈ set (history i) and Broadcast (i2, Insert e2 n2) ∈ set (history j) shows Insert e1 n1 = Insert e2 n2 using insert-id-unique assms by simp

lemma (in rga) same-insert: assumes fst e1 = fst e2 and xs prefix of i and (i1, Insert e1 n1) ∈ set (node-deliver-messages xs) and (i2, Insert e2 n2) ∈ set (node-deliver-messages xs) shows Insert e1 n1 = Insert e2 n2 proof − have Deliver (i1, Insert e1 n1) ∈ set (history i) using assms by (auto simp add: node-deliver-messages-def prefix-msg-in-history) from this obtain j where 1: Broadcast (i1, Insert e1 n1) ∈ set (history j) using delivery-has-a-cause by blast have Deliver (i2, Insert e2 n2) ∈ set (history i) using assms by (auto simp add: node-deliver-messages-def prefix-msg-in-history) from this obtain k where 2: Broadcast (i2, Insert e2 n2) ∈ set (history k) using delivery-has-a-cause by blast show Insert e1 n1 = Insert e2 n2 by (rule Insert-equal; force simp add: assms intro: 1 2)

qed

lemma (in rga) insert-commute-assms: assumes {Deliver (i, Insert e n), Deliver (i', Insert e' n')} ⊆ set (history j) and hb.concurrent (i, Insert e n) (i', Insert e' n') shows n = None ∨ n ≠ Some (fst e') using assms
apply (clarsimp simp: hb.concurrent-def)
apply (cases e')
apply clarsimp
apply (frule delivery-has-a-cause)
apply (clarsimp)

qed
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\} \subseteq c
  shows \{b, a\} \subseteq c
using assms by (simp)

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

lemma (in rga) Insert-Delete-concurrent:
  assumes \{Deliver (i, Insert e n), Deliver (i', Delete n')\} \subseteq set (history j)
          and hb.concurrent (i, Insert e n) (i', Delete n')
  shows n' \neq \text{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-commute-assms rga-axioms)

lemma (in rga) concurrent-operations-commute:
  assumes xs prefix of i
  shows hb.concurrent-ops-commute (node-deliver-messages xs)
proof –
  have \(\forall x\ y. \{x, y\} \subseteq \text{set (node-deliver-messages xs)} \rightarrow hb.concurrent x y \rightarrow interp-msg x \triangleright interp-msg y = interp-msg y \triangleright interp-msg x\)
proof
  fix x y ii
  assume \(\{x, y\} \subseteq \text{set (node-deliver-messages xs)}\)
  and C: \(hb.concurrent x y\)
  hence X: \(x \in \text{set (node-deliver-messages xs)}\) and Y: \(y \in \text{set (node-deliver-messages xs)}\)
  by auto
  obtain x1 x2 y1 y2 where 1: \(x = (x1, x2)\) and 2: \(y = (y1, y2)\)
  by fastforce
  have \((interp-msg (x1, x2) \triangleright interp-msg (y1, y2)) ii = (interp-msg (y1, y2) \triangleright interp-msg (x1, x2))\) ii
  proof (cases x2; cases y2)
    fix ix1 ix2 iy1 iy2
    assume X2: \(x2 = \text{Insert } ix1 ix2\) and Y2: \(y2 = \text{Insert } iy1 iy2\)
    show \((interp-msg (x1, x2) \triangleright interp-msg (y1, y2)) ii = (interp-msg (y1, y2) \triangleright interp-msg (x1, x2))\) ii
      proof (cases fst ix1 = fst iy1)
        assume \(\text{fst } ix1 = \text{fst } iy1\)
        hence IX1: \(ix1 = \text{Insert } iy1 iy2\)
          apply (rule same-insert)
        using 1 2 X Y X2 Y2 assms apply auto
        done
        hence \(ix1 = iy1\) and \(ix2 = iy2\)
by auto

from this and $X2 \ Y2$ show \((\interpmsg \ (x1, x2) \imp \interpmsg \ (y1, y2)) \ iI = (\interpmsg \ (y1, y2) \imp \interpmsg \ (x1, x2)) \ iI\)
  by (clarsimp simp add: kleisli_def interp-msg-def)

next

assume \(NEQ: \fst \ iI1 \neq \fst \ iy1\)

have \(iI2 = \text{None} \lor iI2 \neq \text{Some} (\fst \ iy1)\)
  apply (rule insert-commute-assms)
  using prefix-msg-in-history \(\text{OF} \ \assms\) \(X Y X2 Y2 1 2\)
  apply (clarsimp, blast)
  using \(C 1 \ 2 \ X2 \ Y2\) apply blast
  done

also have \(iy2 = \text{None} \lor iy2 \neq \text{Some} (\fst \ iy1)\)
  apply (rule insert-commute-assms)
  using prefix-msg-in-history \(\text{OF} \ \assms\) \(X Y X2 Y2 1 2\)
  apply (clarsimp, blast)
  using \(1 \ 2 \ C \ X2 \ Y2\) apply blast
  done

ultimately have \(\text{insert} \ iI \ iI1 \ iI2 \equiv (\lambda x. \text{insert} \ x \ iy1 \ iy2) = \text{insert} \ iI \ iy1 \ iy2 \equiv (\lambda x. \text{insert} \ x \ iI1 \ iI2)\)
  using \(NEQ\) insert-commutes by blast

thus \((\interpmsg \ (x1, x2) \imp \interpmsg \ (y1, y2)) \ iI = (\interpmsg \ (y1, y2) \imp \interpmsg \ (x1, x2)) \ iI\)
  by (clarsimp simp add: interp-msg-def X2 Y2 kleisli_def)

qed

next

fix \(iI1 iI2 yd\)

assume \(X2: x2 = \text{Insert} \ iI1 iI2\) and \(Y2: y2 = \text{Delete} \ yd\)
  thm insert-delete-commute
  thm Insert-Delete-concurrent

have \(hb.\text{concurrent} (x1, \text{Insert} \ iI1 iI2) (y1, \text{Delete} \ yd)\)
  using \(C X2 Y2 1 2\) by simp

also have \(\{\text{Deliver} (x1, \text{Insert} \ iI1 iI2), \text{Deliver} (y1, \text{Delete} \ yd)\} \subseteq \text{set} \ \text{history} \ i\)
  using prefix-msg-in-history \(\assms\) \(X2 Y2 X Y 1 2\) by blast

ultimately have \(yd \neq \fst \ iI1\)
  apply --
  apply (rule Insert-Delete-concurrent; force)
  done

hence \(\text{insert} \ iI \ iI1 iI2 \equiv (\lambda x. \text{delete} \ x \ yd) = \text{delete} \ iI iI2 \equiv (\lambda x. \text{insert} \ x \ iI1 iI2)\)
  by (rule insert-delete-commute)

thus \((\interpmsg \ (x1, x2) \imp \interpmsg \ (y1, y2)) \ iI = (\interpmsg \ (y1, y2) \imp \interpmsg \ (x1, x2)) \ iI\)
  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 \(hb.\text{concurrent} (x1, \text{Delete} \ xd) (y1, \text{Insert} \ iy1 iy2)\)
    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 \(\assms\) \(X2 Y2 X Y 1 2\) by blast

ultimately have \(xd \neq \fst \ iy1\)
  apply --
  apply (rule Insert-Delete-concurrent; force)
  done

hence \(\text{delete} \ iI \ xd \equiv (\lambda x. \text{insert} \ x \ iy1 iy2) = \text{insert} \ iI \ iy1 iy2 \equiv (\lambda x. \text{delete} \ x \ xd)\)
  by (rule insert-delete-commute[symmetric])

thus \((\interpmsg \ (x1, x2) \imp \interpmsg \ (y1, y2)) \ iI = (\interpmsg \ (y1, y2) \imp \interpmsg \ (x1, x2)) \ iI\)
by (clarsimp simp add: interp-msg-def kleisli-def X2 Y2)

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

thus (interp-msg x \triangleright interp-msg y) ii = (interp-msg y \triangleright interp-msg x) ii
  using 1 2 by auto
qed

thus hh.concurrent-ops-commute (node-deliver-messages xs)
by (auto simp add: hh.concurrent-ops-commute-def)
qed

**corollary** (in rga) concurrent-operations-commute':
shows hh.concurrent-ops-commute (node-deliver-messages (history i))
by (meson concurrent-operations-commute append.right-neutral prefix-of-node-history-def)

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

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

next
fix d
assume 4: x = Deliver d
thus apply-operations (xs \@[x]) \neq None
proof (cases d; clarify)
fix a b
assume 5: x = Deliver (a, b)
show \Ex y. apply-operations (xs \@[Deliver (a, b)]) = Some y
proof (cases b; clarify)
fix aa aaa ba x12
assume 6: b = Insert (aa, aaa, ba) x12
show \Ex y. apply-operations (xs \@[Deliver (a, Insert (aa, aaa, ba) x12)]) = 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 x2
assume 6: \( b = \text{Delete } x2 \)

show \( \exists y. \text{apply-operations} (xs @ [\text{Deliver} (a, \text{Delete } x2)]) = \text{Some } y \)

apply (clarsimp simp add: interp-msg-def split!: bind-splits)
apply (simp add: 1 3)
apply (rule delete-no-failure)
using 4 5 6 2 apply force+
done
qed
qed
qed
qed

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

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

5.7 Strong eventual consistency

context rga begin

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

proof (standard; clarsimp)
fix \( xs a \)
assume \( xs \text{ prefix of } i \)
thus \( \text{hb.hb-consistent} (\text{node-deliver-messages } xsa) \)
by (auto simp add: hb-consistent-prefix)

next
fix \( xs a \)
assume \( xs \text{ prefix of } i \)
thus \( \text{distinct} (\text{node-deliver-messages } xsa) \)
by (auto simp add: node-deliver-messages-distinct)

next
fix \( xs a \)
assume \( xs \text{ prefix of } i \)
thus \( \text{hb.concurrent-ops-commute} (\text{node-deliver-messages } xsa) \)
by (auto simp add: concurrent-operations-commute)

next
fix \( xs a \ state \ xsa x \)
assume \( \text{hb.apply-operations} \ (xs [] = \text{Some } state \land \text{node-deliver-messages } xsa = xs @ [(a, b)] \land \text{xs prefix of } x \)
thus \( \exists y. \text{interp-msg} (a, b) \text{ state } = \text{Some } y \)
by (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})

next
fix \( xs a \ state \ xsa x \)
assume \( \text{node-deliver-messages } xsa = xs @ [(a, b)] \land \text{xs prefix of } x \)
thus \( \exists xsa. \text{Ex } (\text{op prefix of } xsa) \land \text{node-deliver-messages } xsa = xs \)
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 add: 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 (clarsimp simp: hb-conconsistent-prefix drop-last-message)
apply (clarsimp simp add: node-deliver-messages-distinct concurrent-operations-commute)
apply (metis (full-types) interp-msg-def counter-op.elims)
using drop-last-message apply blast
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 (Ị [0] 1000) 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-append)
  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 ⇒ ′b ⇒ ′id list where
  added-ids es p ≡ List.map-filter (λ x. 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 ?thesis
    using snoc by (case_tac b; clarsimp simp: interp-msg-def split: bind-splits,
                         force split: if-split-asm simp add: op-elem-def interpret-op-def)
**lemma** (in orset) Deliver-added-ids:

**assumes** xs prefix of j

and i ∈ set (added-ids xs e)

**shows** Deliver (i, Add i e) ∈ set xs

**using** assms proof (induct xs rule: rev-induct, clarsimp)

**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 ?thesis

using snoc apply (case-tac b; clarsimp)

apply (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** (in orset) Broadcast-Deliver-prefix-closed:

**assumes** xs @ [Broadcast (r, Rem ix e)] prefix of j

and i ∈ ix

**shows** Deliver (i, Add i e) ∈ set xs

**proof** –

obtain y where apply-operations xs = Some y

using assms broadcast-only-valid-msgs by blast

moreover hence ix = y e

by (metis (mono-tags, lifting) assms(1) broadcast-only-valid-msgs operation.case(2) option.simps(1)

valid-behaviours-def case-prodD)

ultimately show ?thesis

using assms Deliver-added-ids apply-operations-added-ids by blast

qed

**lemma** (in orset) Broadcast-Deliver-prefix-closed2:

**assumes** xs prefix of j

and Broadcast (r, Rem ix e) ∈ set xs

and i ∈ ix

**shows** Deliver (i, Add i e) ∈ set xs

**using** assms Broadcast-Deliver-prefix-closed by (induction xs rule: rev-induct; force)

**lemma** (in orset) concurrent-add-remove-independent-technical:

**assumes** i ∈ is

and xs prefix of j

and (i, Add i e) ∈ set (node-deliver-messages xs) and (ir, Rem is e) ∈ set (node-deliver-messages xs)

**shows** hb (i, Add i e) (ir, Rem is e)

**proof** –

obtain pre k where pre@[Broadcast (ir, Rem is e)] prefix of k

using assms delivery-has-a-cause events-before-exist prefix-msg-in-history by blast

moreover hence Deliver (i, Add i e) ∈ set pre

using Broadcast-Deliver-prefix-closed assms(1) by auto

ultimately show ?thesis

using hb.intro(2) events-in-local-order by blast

qed

**lemma** (in orset) Deliver-Add-same-id-same-message:
assumes \( \text{Deliver} (i, \text{Add} i e1) \in \text{set} \ (\text{history} \ j) \) and \( \text{Deliver} (i, \text{Add} i e2) \in \text{set} \ (\text{history} \ j) \)

shows \( e1 = e2 \)

proof –

obtain \( \text{pre1} \ \text{pre2} \ k1 \ k2 \) where \( \star: \text{pre1}@[\text{Broadcast} (i, \text{Add} i e1)] \ \text{prefix of} \ k1 \ \text{pre2}@[\text{Broadcast} (i, \text{Add} i e2)] \ \text{prefix of} \ k2 \)

using assms delivery-has-a-cause events-before-exist by meson

moreover hence \( \text{Broadcast} (i, \text{Add} i e1) \in \text{set} \ (\text{history} \ k1) \ \text{Broadcast} (i, \text{Add} i e2) \in \text{set} \ (\text{history} \ k2) \)

using node-histories.prefix-to-carriers node-histories-axioms by force+

ultimately show \( \text{thesis} \)

using msg-id-unique by fastforce

qed

lemma (in orset) ids-imply-messages-same:

assumes \( i \in \text{is} \)

and \( \text{xs prefix of} \ j \)

and \( (i, \text{Add} i e1) \in \text{set} \ \text{(node-deliver-messages} \ \text{xs}) \) and \( (ir, \text{Rem} \ is \ e2) \in \text{set} \ \text{(node-deliver-messages} \ \text{xs}) \)

shows \( e1 = e2 \)

proof –

obtain \( \text{pre} \ k \) where \( \text{pre}@[\text{Broadcast} (ir, \text{Rem} \ is \ e2)] \ \text{prefix of} \ k \)

using assms delivery-has-a-cause events-before-exist prefix-msg-in-history by blast

moreover hence \( \text{Deliver} (i, \text{Add} i e1) \in \text{set} \ \text{pre} \)

using Broadcast-Deliver-prefix-closed assms(1) by blast

moreover have \( \text{Deliver} (i, \text{Add} i e1) \in \text{set} \ (\text{history} \ j) \)

using assms(2) assms(3) prefix-msg-in-history by blast

ultimately show \( \text{thesis} \)

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 \( \neg \text{hb} \ (i, \text{Add} i e1) \ (ir, \text{Rem} \ is \ e2) \) and \( \neg \text{hb} \ (ir, \text{Rem} \ is \ e2) \ (i, \text{Add} i e1) \)

and \( \text{xs prefix of} \ j \)

and \( (i, \text{Add} i e1) \in \text{set} \ \text{(node-deliver-messages} \ \text{xs}) \) and \( (ir, \text{Rem} \ is \ e2) \in \text{set} \ \text{(node-deliver-messages} \ \text{xs}) \)

shows \( i \notin \text{is} \)

using assms ids-imply-messages-same concurrent-add-remove-independent-technical by fastforce

lemma (in orset) rem-rem-commute:

shows \( \langle \text{Rem} \ i1 \ e1 \rangle \triangleright \langle \text{Rem} \ i2 \ e2 \rangle = \langle \text{Rem} \ i2 \ e2 \rangle \triangleright \langle \text{Rem} \ i1 \ e1 \rangle \)

by (unfold interpret-op-def op-elem-def kleisli-def, fastforce)

lemma (in orset) concurrent-operations-commute:

assumes \( \text{xs prefix of} \ i \)

shows \( \text{hb.concurrent-ops-commute} \ \text{(node-deliver-messages} \ \text{xs}) \)

proof –

\{ fix \ a \ b \ x \ y \}

assume \( a, b \in \text{set} \ \text{(node-deliver-messages} \ \text{xs}) \)

\( (x, y) \in \text{set} \ \text{(node-deliver-messages} \ \text{xs}) \)

\( \text{hb.concurrent} \ (a, b) \ (x, y) \)

hence \( \text{interp-msg} (a, b) \triangleright \text{interp-msg} (x, y) = \text{interp-msg} (x, y) \triangleright \text{interp-msg} (a, b) \)

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 hb.concurrentD2 prefix-contains-msg)+

done

\} thus \( \text{thesis} \)
by (fastforce simp: hb.concurrent-ops-commute-def)

qed

theorem (in orset) 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 orset begin

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.{}

apply (standard; clarsimp simp add: hb-consistent-prefix node-deliver-messages-distinct
current-operations-commute)

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)
using drop-last-message apply blast

end

References


