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

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

October 11, 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

5.4 Commutativity of concurrent operations 24
5.5 Alternative definition of insert 25
5.6 Network 27
5.7 Strong eventual consistency 36

6 Increment-Decrement Counter 37

7 Observed-Remove Set 38

1 Introduction

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

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

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

2 Technical Lemmas

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

theory
  Util
imports
  Main
  HOL−Library.Monad-Syntax

begin

2.1 Kleisli arrow composition

definition *kleisli* :: $('b \Rightarrow 'b\ option) \Rightarrow ('b \Rightarrow 'b\ option) \Rightarrow ('b \Rightarrow 'b\ option)$ (**infixr** $\rhd$ *65*) **where**
$f \rhd g \equiv \lambda x.\ (f\ x \gg\!= (\lambda y.\ g\ y))$

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

lemma *kleisli-assoc*:
shows $(z \rhd x) \rhd y = z \rhd (x \rhd y)$
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 $pre1@ys@suf1 = xs$

and *pre2@ys@suf2* = *xs*
shows *pre1* = *pre2* ∧ *suf1* = *suf2*
using *assms*
apply(*induction xs arbitrary*: *pre1 pre2 ys*, *simp*)
apply(*case-tac pre1*; *case-tac pre2*; *clarify*)
apply(*metis suffix-eq-distinct-list append-Nil*)
apply(*metis Un-iff append-eq-Cons-conv distinct.simps*(*2*) *list.set-intros*(*1*) *set-append suffix-eq-distinct-list*)
apply(*metis Un-iff append-eq-Cons-conv distinct.simps*(*2*) *list.set-intros*(*1*) *set-append suffix-eq-distinct-list*)
apply(*metis distinct.simps*(*2*) *hd-append2 list.sel*(*1*) *list.sel*(*3*) *list.simps*(*3*) *tl-append2*)
done

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

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

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

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

lemma *list-nth-split-technical*:
assumes *m* < *length cs*
and *cs* ≠ []
shows ∃ *xs ys*. *cs* = *xs*@(*cs*!*m*)#*ys*
using *assms*
apply(*induction m arbitrary*: *cs*)
apply(*meson in-set-conv-decomp nth-mem*)
apply(*metis in-set-conv-decomp length-list-update set-swap set-update-memI*)
done

lemma *list-nth-split*:
assumes *m* < *length cs*
and *n* < *m*
and *1* < *length cs*
shows ∃ *xs ys zs*. *cs* = *xs*@(*cs*!*n*)#*ys*@(*cs*!*m*)#*zs*
using *assms* **proof**(*induction n arbitrary*: *cs m*)
case *0* **thus** *?case*
apply(*case-tac cs*; *clarsimp*)
apply(*rule-tac x*=[] **in** *exI*, *clarsimp*)

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

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

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

end

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* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\preceq$ *50*)
  and *hb* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\prec$ *50*) +
  fixes *interp* :: $'a \Rightarrow 'b \rightharpoonup 'b$ ($\langle$-$\rangle$ [*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: $\neg(x \prec y)$ and $\neg(y \prec x)$.

definition *concurrent* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\|$ *50*) **where**
  $s1 \parallel s2 \equiv \neg\ (s1 \prec s2) \land \neg\ (s2 \prec s1)$

lemma *concurrentI* [*intro*!]: $\neg\ (s1 \prec s2) \Longrightarrow \neg\ (s2 \prec s1) \Longrightarrow s1 \parallel s2$
  by (*auto simp*: *concurrent-def*)

lemma *concurrentD1* [*dest*]: $s1 \parallel s2 \Longrightarrow \neg\ (s1 \prec s2)$
  by (*auto simp*: *concurrent-def*)

lemma *concurrentD2* [*dest*]: $s1 \parallel s2 \Longrightarrow \neg\ (s2 \prec s1)$
  by (*auto simp*: *concurrent-def*)

lemma *concurrent-refl* [*intro*!, *simp*]: $s \parallel s$
  by (*auto simp*: *concurrent-def*)

lemma *concurrent-comm*: $s1 \parallel s2 \longleftrightarrow s2 \parallel s1$
  by (*auto simp*: *concurrent-def*)

definition *concurrent-set* :: $'a \Rightarrow 'a\ list \Rightarrow bool$ **where**
  concurrent-set x xs $\equiv \forall y \in set\ xs.\ x \parallel y$

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* $\Longrightarrow$ *concurrent x a* $\Longrightarrow$ G
shows G
using *assms* **by** (*auto simp*: *concurrent-set-def*)

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

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

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

3.2 Happens-before consistency

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

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

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

lemma ($x \prec y \vee$ *concurrent x y*) = ($\neg\, y \prec x$)
using *less-asym* **by** *blast*

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

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

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

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

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

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

lemma hb-consistent-remove1 [intro]:
  assumes hb-consistent xs
  shows   hb-consistent (remove1 x xs)
  using assms by (induction rule: hb-consistent.induct) (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 {
```

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

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

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

3.3 Apply operations

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

definition *apply-operations* :: *'a list* $\Rightarrow$ *'b* $\rightharpoonup$ *'b* **where**
apply-operations es $\equiv$ *foldl* (*op* $\rhd$) *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*) $\rhd$ $\langle x\rangle$
by(*auto simp add*: *apply-operations-def kleisli-def*)

3.4 Concurrent operations commute

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

definition *concurrent-ops-commute* :: *'a list* $\Rightarrow$ *bool* **where**
concurrent-ops-commute xs $\equiv$
$\forall$ *x y*. {*x*, *y*} $\subseteq$ *set xs* $\longrightarrow$ *concurrent x y* $\longrightarrow$ $\langle x\rangle\rhd\langle y\rangle = \langle y\rangle\rhd\langle x\rangle$

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

lemma *concurrent-ops-commute-singleton* [*intro*!]: *concurrent-ops-commute* [*x*]

```
  by(auto simp: concurrent-ops-commute-def)

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

lemma concurrent-ops-commute-rearrange:
  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]) ⟹
             concurrent-set x xs ⟹ distinct (prefix @ x # xs) ⟹
             apply-operations (prefix @ xs @ [x]) = apply-operations (prefix @ x # xs)
  assume assms: concurrent-ops-commute (prefix @ (xs @ [a]) @ [x])
               concurrent-set x (xs @ [a]) distinct (prefix @ x # xs @ [a])
  hence ac-comm: ⟨a⟩ ▷ ⟨x⟩ = ⟨x⟩ ▷ ⟨a⟩
    by (clarsimp simp: concurrent-ops-commute-def) blast
  have copc: concurrent-ops-commute (prefix @ xs @ [x])
    using assms by (clarsimp simp: concurrent-ops-commute-def) blast
  have apply-operations ((prefix @ x # xs) @ [a]) = (apply-operations (prefix @ x # xs)) ▷ ⟨a⟩
    by (simp del: append-assoc)
  also have ... = (apply-operations (prefix @ xs @ [x])) ▷ ⟨a⟩
    using IH assms copc by auto
  also have ... = ((apply-operations (prefix @ xs)) ▷ ⟨x⟩) ▷ ⟨a⟩
    by (simp add: append-assoc[symmetric] del: append-assoc)
  also have ... = (apply-operations (prefix @ xs)) ▷ (⟨a⟩ ▷ ⟨x⟩)
    using ac-comm kleisli-comm-cong kleisli-assoc by simp
  finally show apply-operations (prefix @ (xs @ [a]) @ [x]) = apply-operations (prefix @ x # xs @ [a])
    by (metis Cons-eq-appendI append-assoc apply-operations-Snoc kleisli-assoc)
qed
```

3.5 Abstract convergence theorem

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

```
theorem convergence:
  assumes set xs = set ys
          concurrent-ops-commute xs
          concurrent-ops-commute ys
          distinct xs
          distinct ys
          hb-consistent xs
          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 ∧ concurrent-set x suffix
    using hb-consistent-prefix-suffix-exists by fastforce
```

```
  moreover hence *: distinct (prefix @ suffix) hb-consistent xs
    using assms by auto
  moreover {
    have hb-consistent prefix hb-consistent suffix
      using ys-split assms hb-consistent-append-D2 hb-consistent-append-elim-ConsD by blast+
    hence hb-consistent (prefix @ suffix)
      by (metis assms(8) hb-consistent-append hb-consistent-append-porder list.set-intros(2) ys-split)
  }
  moreover have **: concurrent-ops-commute (prefix @ suffix @ [x])
    using assms ys-split by (clarsimp simp: concurrent-ops-commute-def)
  moreover hence concurrent-ops-commute (prefix @ suffix)
    by (force simp del: append-assoc simp add: append-assoc[symmetric])
  ultimately have apply-operations xs = apply-operations (prefix@suffix)
   using assms by simp (metis Diff-insert-absorb Un-iff * concurrent-ops-commute-appendD set-append)
  moreover have apply-operations (prefix@suffix @ [x]) = apply-operations (prefix@x # suffix)
    using ys-split assms ** concurrent-ops-commute-concurrent-set by force
  ultimately show ?case
    using ys-split by (force simp: append-assoc[symmetric] simp del: append-assoc)
qed

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

3.6 Convergence and progress

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

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

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

theorem *sec-progress*:
  assumes *op-history xs*
  shows *apply-operations xs initial-state* $\neq$ *None*
using *assms* **proof**(*induction xs rule*: *rev-induct*, *simp*)
  case (*snoc x xs*)
  have *apply-operations xs initial-state* $\neq$ *None*
    using *snoc.IH snoc.prems trunc-history kleisli-def bind-def* **by** *blast*
  moreover have *apply-operations* (*xs* @ [*x*]) = *apply-operations xs* $\triangleright$ $\langle x \rangle$
    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* $\Rightarrow$ *'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* $\Rightarrow$ *nat* $\Rightarrow$ *'evt* $\Rightarrow$ *bool* (-/ $\sqsubset^{-}$/ - [*50*,*1000*,*50*]*50*)
where
  $x \sqsubset^i z \equiv \exists\, xs\ ys\ zs.\ xs@x\#ys@z\#zs = history\ i$

lemma (**in** *node-histories*) *node-total-order-trans*:
  assumes *e1* $\sqsubset^i$ *e2*
    and *e2* $\sqsubset^i$ *e3*
  shows *e1* $\sqsubset^i$ *e3*

proof −
  obtain *xs1 xs2 ys1 ys2 zs1 zs2* **where** ∗: *xs1* @ *e1* # *ys1* @ *e2* # *zs1* = *history i*
    xs2 @ *e2* # *ys2* @ *e3* # *zs2* = *history i*
   using *history-order-def assms* **by** *auto*
  hence *xs1* @ *e1* # *ys1* = *xs2* ∧ *zs1* = *ys2* @ *e3* # *zs2*
   by(*rule-tac xs=history i* **and** *ys*=[*e2*] **in** *pre-suf-eq-distinct-list*) *auto*
  thus *?thesis*
   by(*clarsimp simp*: *history-order-def*) (*metis* ∗(*2*) *append.assoc append-Cons*)
qed

lemma (**in** *node-histories*) *local-order-carrier-closed*:
  assumes $e1 \sqsubset^i e2$
   shows {*e1*,*e2*} ⊆ *set* (*history i*)
  using *assms* **by** (*clarsimp simp add*: *history-order-def*)
   (*metis in-set-conv-decomp Un-iff Un-subset-iff insert-subset list.simps*(*15*)
    set-append set-subset-Cons)+

lemma (**in** *node-histories*) *node-total-order-irrefl*:
  shows ¬ ($e \sqsubset^i e$)
  by(*clarsimp simp add*: *history-order-def*)
   (*metis Un-iff histories-distinct distinct-append distinct-set-notin*
    list.set-intros(*1*) *set-append*)

lemma (**in** *node-histories*) *node-total-order-antisym*:
  assumes $e1 \sqsubset^i e2$
    and $e2 \sqsubset^i e1$
   shows *False*
  using *assms node-total-order-irrefl node-total-order-trans* **by** *blast*

lemma (**in** *node-histories*) *node-order-is-total*:
  assumes *e1* ∈ *set* (*history i*)
    and *e2* ∈ *set* (*history i*)
    and *e1* ≠ *e2*
   shows $e1 \sqsubset^i e2 \vee e2 \sqsubset^i e1$
  using *assms* **unfolding** *history-order-def* **by**(*metis list-split-two-elems histories-distinct*)

definition (**in** *node-histories*) *prefix-of-node-history* :: ′*evt list* ⇒ *nat* ⇒ *bool* (**infix** *prefix of 50*) **where**
  xs prefix of i ≡ ∃ *ys*. *xs*@*ys* = *history i*

lemma (**in** *node-histories*) *carriers-head-lt*:
  assumes *y*#*ys* = *history i*
  shows ¬($x \sqsubset^i y$)
using *assms*
  apply(*clarsimp simp add*: *history-order-def*)
  apply(*rename-tac xs1 ys1 zs1*)
   apply (*subgoal-tac xs1* @ *x* # *ys1* = [] ∧ *zs1* = *ys*)
  apply *clarsimp*
  apply (*rule-tac xs=history i* **and** *ys*=[*y*] **in** *pre-suf-eq-distinct-list*)
   apply *auto*
  done

lemma (**in** *node-histories*) *prefix-of-ConsD* [*dest*]:
  assumes *x* # *xs prefix of i*
   shows [*x*] *prefix of i*
  using *assms* **by**(*auto simp*: *prefix-of-node-history-def*)

lemma (**in** *node-histories*) *prefix-of-appendD* [*dest*]:
  assumes *xs* @ *ys 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 *set xs* $\subseteq$ *set* (*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 $x \in$ *set xs*
shows $x \in$ *set* (*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 $x \sqsubset^i y$
and *xs prefix of i*
and $y \in$ *set xs*
shows $x \in$ *set xs*
proof −
obtain *ys* **where** *xs* @ *ys* = *history i*
using *assms prefix-of-node-history-def* **by** *blast*
moreover obtain *as bs cs* **where** *as* @ *x* # *bs* @ *y* # *cs* = *history i*
using *assms history-order-def* **by** *blast*
moreover obtain *pre suf* **where** ∗: *xs* = *pre* @ *y* # *suf*
using *assms split-list* **by** *fastforce*
ultimately have *pre* = *as* @ *x* # *bs* $\wedge$ *suf* @ *ys* = *cs*
by (*rule-tac xs=history i* **and** *ys*=[*y*] **in** *pre-suf-eq-distinct-list*) *auto*
thus *?thesis*
using *assms* ∗ **by** *clarsimp*
qed

lemma (**in** *node-histories*) *local-order-prefix-closed-last*:
assumes $x \sqsubset^i y$
and *xs*@[*y*] *prefix of i*
shows $x \in$ *set xs*
proof −
have $x \in$ *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 $x \in$ *set* (*history i*)
shows ∃ *pre*. *pre* @ [*x*] *prefix of i*
proof −
have ∃ *idx*. *idx* < *length* (*history i*) $\wedge$ (*history i*) ! *idx* = *x*
using *assms* **by**(*simp add*: *set-elem-nth*)
thus *?thesis*
by(*metis append-take-drop-id take-Suc-conv-app-nth prefix-of-node-history-def*)
qed

lemma (**in** *node-histories*) *events-in-local-order*:
assumes *pre* @ [*e2*] *prefix of i*
and $e1 \in set\ pre$
shows $e1 \sqsubset^i e2$
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*: $[\![\ Deliver\ m \in set\ (history\ i)\ ]\!] \Longrightarrow \exists j.\ Broadcast\ m \in set\ (history\ j)$
and *deliver-locally*: $[\![\ Broadcast\ m \in set\ (history\ i)\ ]\!] \Longrightarrow Broadcast\ m \sqsubset^i Deliver\ m$
and *msg-id-unique*: $[\![\ Broadcast\ m1 \in set\ (history\ i);\ Broadcast\ m2 \in set\ (history\ j);\ msg\text{-}id\ m1 = msg\text{-}id\ m2\ ]\!] \Longrightarrow i = j \wedge m1 = m2$

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-id*s are the same, then they were in fact broadcast by the same node and the two messages are identical. In practice, these globally unique IDs can by implemented using unique node identifiers, sequence numbers or timestamps.

lemma (**in** *network*) *broadcast-before-delivery*:
assumes $Deliver\ m \in set\ (history\ i)$
shows $\exists j.\ Broadcast\ m \sqsubset^j Deliver\ m$
using *assms deliver-locally delivery-has-a-cause* **by** *blast*

lemma (**in** *network*) *broadcasts-unique*:
assumes $i \neq j$
and $Broadcast\ m \in set\ (history\ i)$
shows $Broadcast\ m \notin set\ (history\ j)$

using *assms msg-id-unique* **by** *blast*

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 $m3 \prec m2$.

inductive (**in** *network*) *hb* :: $'msg \Rightarrow 'msg \Rightarrow bool$ **where**
hb-broadcast: $[\![$ *Broadcast m1* $\sqsubset^i$ *Broadcast m2* $]\!] \Longrightarrow$ *hb m1 m2* |
hb-deliver: $[\![$ *Deliver m1* $\sqsubset^i$ *Broadcast m2* $]\!] \Longrightarrow$ *hb m1 m2* |
hb-trans: $[\![$ *hb m1 m2*; *hb m2 m3* $]\!] \Longrightarrow$ *hb m1 m3*

inductive-cases (**in** *network*) *hb-elim*: *hb x y*

definition (**in** *network*) *weak-hb* :: $'msg \Rightarrow 'msg \Rightarrow bool$ **where**
weak-hb m1 m2 $\equiv$ *hb m1 m2* $\vee$ $m1 = m2$

locale *causal-network* = *network* +
assumes *causal-delivery*: *Deliver m2* $\in$ *set* (*history j*) $\Longrightarrow$ *hb m1 m2* $\Longrightarrow$ *Deliver m1* $\sqsubset^j$ *Deliver m2*

lemma (**in** *causal-network*) *causal-broadcast*:
assumes *Deliver m2* $\in$ *set* (*history j*)
and *Deliver m1* $\sqsubset^i$ *Broadcast m2*
shows *Deliver m1* $\sqsubset^j$ *Deliver m2*
using *assms causal-delivery hb.intros*(*2*) **by** *blast*

lemma (**in** *network*) *hb-broadcast-exists1*:
assumes *hb m1 m2*
shows $\exists i$. *Broadcast m1* $\in$ *set* (*history i*)
using *assms*
apply(*induction rule*: *hb.induct*)
apply(*meson insert-subset node-histories.local-order-carrier-closed node-histories-axioms*)
apply(*meson delivery-has-a-cause insert-subset local-order-carrier-closed*)
apply *simp*
done

lemma (**in** *network*) *hb-broadcast-exists2*:
assumes *hb m1 m2*
shows $\exists i$. *Broadcast m2* $\in$ *set* (*history i*)
using *assms*
apply(*induction rule*: *hb.induct*)
apply(*meson insert-subset node-histories.local-order-carrier-closed node-histories-axioms*)
apply(*meson delivery-has-a-cause insert-subset local-order-carrier-closed*)
apply *simp*
done

4.3 Causal networks

lemma (**in** *causal-network*) *hb-has-a-reason*:
assumes *hb m1 m2*
and *Broadcast m2* $\in$ *set* (*history i*)
shows *Deliver m1* $\in$ *set* (*history i*) $\vee$ *Broadcast m1* $\in$ *set* (*history i*)
using *assms* **apply** (*induction rule*: *hb.induct*)
apply(*metis insert-subset local-order-carrier-closed network.broadcasts-unique network-axioms*)
apply(*metis insert-subset local-order-carrier-closed network.broadcasts-unique network-axioms*)
using *hb-trans causal-delivery local-order-carrier-closed* **apply** *blast*

```
  done

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

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

lemma (in causal-network) hb-broadcast-broadcast-order:
  assumes hb m1 m2
    and Broadcast m1 ∈ set (history i)
    and Broadcast m2 ∈ set (history i)
  shows Broadcast m1 ⊏^i Broadcast m2
using assms proof(induction rule: hb.induct)
  case (hb-broadcast m1 i m2) thus ?case
    by(metis insertI1 local-order-carrier-closed network.broadcasts-unique network-axioms subsetCE)
next
  case (hb-deliver m1 i m2) thus ?case
    by(metis broadcasts-unique insert-subset local-order-carrier-closed
          network.broadcast-before-delivery network-axioms node-total-order-trans)
next
  case (hb-trans m1 m2 m3)
  then show ?case
  proof (cases Broadcast m2 ∈ set (history i))
    case True thus ?thesis
      using hb-trans node-total-order-trans by blast
  next
    case False hence Deliver m2 ∈ set (history i) m1 ≠ m2 m2 ≠ m3
      using hb-has-a-reason hb-trans by auto
    thus ?thesis
        by(metis hb-trans event.inject(1) hb.intros(1) hb-irrefl network.hb.intros(3) network-axioms
```

node-order-is-total hb-irrefl)
  qed
qed

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* $\sqsubset^i$ *Broadcast m2*
  thus *False*
    apply − **proof**(*erule hb-elim*)
    show $\bigwedge$*ia. Broadcast m1* $\sqsubset^i$ *Broadcast m2* $\Longrightarrow$ *Broadcast m2* $\sqsubset^i$*a Broadcast m1* $\Longrightarrow$ *False*
     by(*metis broadcasts-unique insert-subset local-order-carrier-closed node-total-order-irrefl node-total-order-trans*)
  next
    show $\bigwedge$*ia. Broadcast m1* $\sqsubset^i$ *Broadcast m2* $\Longrightarrow$ *Deliver m2* $\sqsubset^i$*a Broadcast m1* $\Longrightarrow$ *False*
     by(*metis broadcast-before-delivery broadcasts-unique insert-subset local-order-carrier-closed node-total-order-irrefl node-total-order-trans*)
  next
    show $\bigwedge$*m2a. Broadcast m1* $\sqsubset^i$ *Broadcast m2* $\Longrightarrow$ *hb m2 m2a* $\Longrightarrow$ *hb m2a m1* $\Longrightarrow$ *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* $\sqsubset^i$ *Broadcast m2*
  thus *False*
    apply − **proof**(*erule hb-elim*)
    show $\bigwedge$*ia. Deliver m1* $\sqsubset^i$ *Broadcast m2* $\Longrightarrow$ *Broadcast m2* $\sqsubset^i$*a Broadcast m1* $\Longrightarrow$ *False*
     by (*metis broadcast-before-delivery broadcasts-unique insert-subset local-order-carrier-closed node-total-order-irrefl node-total-order-trans*)
  next
    show $\bigwedge$*ia. Deliver m1* $\sqsubset^i$ *Broadcast m2* $\Longrightarrow$ *Deliver m2* $\sqsubset^i$*a Broadcast m1* $\Longrightarrow$ *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 $\bigwedge$*m2a. Deliver m1* $\sqsubset^i$ *Broadcast m2* $\Longrightarrow$ *hb m2 m2a* $\Longrightarrow$ *hb m2a m1* $\Longrightarrow$ *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* $\Longrightarrow$ *False*) (*hb m3 m2* $\Longrightarrow$ *False*)
  thus *False*
    using *hb.intros*(*3*) **by** *blast*
qed

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

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

lemma (**in** *network*) *node-deliver-messages-Cons*:
  shows *node-deliver-messages* (*x#xs*) = (*node-deliver-messages* [*x*])@(*node-deliver-messages xs*)

by(*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 $m \in$ *set* (*node-deliver-messages es*)
shows *Deliver m* $\in$ *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 $m \in$ *set* (*node-deliver-messages es*)
shows *Deliver m* $\in$ *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** $*$: $y \in$ *set* (*node-deliver-messages xs*) $y \in$ *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 $\exists$ *pre*. *pre prefix of i* $\wedge$ *node-deliver-messages pre* = *msgs*
proof −
have *Deliver last-msg* $\in$ *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-conv 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* (*xs* @ [*Broadcast m*]) = *apply-operations xs*
by(*auto simp add*: *apply-operations-def node-deliver-messages-def map-filter-def*)

lemma (**in** *network-with-ops*) *apply-operations-Deliver* [*simp*]:
shows *apply-operations* (*xs* @ [*Deliver m*]) = (*apply-operations xs* ⤜ *interp-msg m*)
by(*auto simp add*: *apply-operations-def node-deliver-messages-def map-filter-def kleisli-def*)

lemma (**in** *network-with-ops*) *hb-consistent-technical*:
assumes $\bigwedge m\ n.\ m < \mathit{length}\ cs \Longrightarrow n < m \Longrightarrow cs\ !\ n \sqsubset^i cs\ !\ m$
shows *hb.hb-consistent* (*node-deliver-messages cs*)

using *assms* **proof** (*induction cs rule*: *rev-induct*)
case *Nil* **thus** *?case*
by(*simp add*: *node-deliver-messages-def hb.hb-consistent.intros*(*1*) *map-filter-simps*(*2*))
next
case (*snoc x xs*)
hence ∗: ($\bigwedge m\ n.\ m < length\ xs \Longrightarrow n < m \Longrightarrow xs\ !\ n \sqsubset^i xs\ !\ m$)
by(−, *erule-tac x*=*m* **in** *meta-allE*, *erule-tac x*=*n* **in** *meta-allE*, *clarsimp simp add*: *nth-append*)
then show *?case*
proof (*cases x*)
case (*Broadcast x1*) **thus** *?thesis*
using *snoc* ∗ **by** (*simp add*: *node-deliver-messages-append*)
next
case (*Deliver x2*) **thus** *?thesis*
using *snoc* ∗ **apply**(*clarsimp simp add*: *node-deliver-messages-def map-filter-def map-filter-append*)
apply (*rename-tac m m1 m2*)
apply (*case-tac m*; *clarsimp*)
apply(*drule set-elem-nth*, *erule exE*, *erule conjE*)
apply(*erule-tac x*=*length xs* **in** *meta-allE*)
apply(*clarsimp simp add*: *nth-append*)
by (*metis causal-delivery insert-subset node-histories.local-order-carrier-closed*
node-histories-axioms node-total-order-antisym)
qed
qed

corollary (**in** *network-with-ops*)
shows *hb.hb-consistent* (*node-deliver-messages* (*history i*))
by (*metis hb-consistent-technical history-order-def less-one linorder-neqE-nat list-nth-split zero-order*(*3*))

lemma (**in** *network-with-ops*) *hb-consistent-prefix*:
assumes *xs prefix of i*
shows *hb.hb-consistent* (*node-deliver-messages xs*)
using *assms* **proof** (*clarsimp simp*: *prefix-of-node-history-def*, *rule-tac i*=*i* **in** *hb-consistent-technical*)
fix *m n ys* **assume** ∗: $xs\ @\ ys = history\ i\ m < length\ xs\ n < m$
consider (*a*) $xs = []$ | (*b*) $\exists c.\ xs = [c]$ | (*c*) $Suc\ 0 < length\ (xs)$
by (*metis Suc-pred length-Suc-conv length-greater-0-conv zero-less-diff*)
thus $xs\ !\ n \sqsubset^i xs\ !\ m$
proof (*cases*)
case *a* **thus** *?thesis*
using ∗ **by** *clarsimp*
next
case *b* **thus** *?thesis*
using *assms* ∗ **by** *clarsimp*
next
case *c* **thus** *?thesis*
using *assms* ∗ **apply** *clarsimp*
apply(*drule list-nth-split*, *assumption*, *clarsimp simp*: *c*)
apply (*metis append.assoc append.simps*(*2*) *history-order-def*)
done
qed
qed

locale *network-with-constrained-ops* = *network-with-ops* +
fixes *valid-msg* :: $'c \Rightarrow ('a \times 'b) \Rightarrow bool$
assumes *broadcast-only-valid-msgs*: $pre\ @\ [Broadcast\ m]\ prefix\ of\ i \Longrightarrow$
$\exists\, state.\ apply\text{-}operations\ pre = Some\ state \wedge valid\text{-}msg\ state\ m$

lemma (**in** *network-with-constrained-ops*) *broadcast-is-valid*:
assumes $Broadcast\ m \in set\ (history\ i)$

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* @ [*Broadcast m*] *prefix of j* $\wedge$ *apply-operations pre* = *Some state* $\wedge$ *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* $\Rightarrow$ $('id,\ 'v)$ *elt* $\Rightarrow$ $('id,\ 'v)$ *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* $\Rightarrow$ $('id,\ 'v)$ *elt* $\Rightarrow$ *'id option* $\Rightarrow$ $('id,\ 'v)$ *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) $\gg=$ (λt. *Some* (x#t)))

fun *delete* :: $('id::\{linorder\},\ 'v)$ *elt list* $\Rightarrow$ *'id* $\Rightarrow$ $('id,\ 'v)$ *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 $\gg=$ (λt. *Some* ((i',v,flag)#t)))

5.2 Well-definedness of insert and delete

lemma *insert-no-failure*:
  assumes $i = None \vee (\exists i'.\ i = Some\ i' \wedge i' \in fst\ `\ set\ xs)$
  shows $\exists xs'.\ insert\ xs\ e\ i = Some\ xs'$
using *assms* **by**(*induction rule*: *insert.induct*; *force*)

lemma *insert-None-index-neq-None* [*dest*]:
  assumes *insert xs e i* = *None*
  shows $i \neq None$
using *assms* **by**(*cases i*, *auto*)

lemma *insert-Some-None-index-not-in* [*dest*]:
  assumes *insert xs e* (*Some i*) = *None*
  shows $i \notin fst\ `\ set\ xs$
using *assms* **by**(*induction xs*, *auto split*: *if-split-asm bind-splits*)

lemma *index-not-in-insert-Some-None* [*simp*]:
  assumes $i \notin fst\ `\ set\ xs$
  shows *insert xs e* (*Some i*) = *None*
using *assms* **by**(*induction xs*, *auto*)

lemma *delete-no-failure*:
  assumes $i \in fst\ `\ set\ xs$
  shows $\exists xs'.\ delete\ xs\ i = Some\ xs'$
using *assms* **by**(*induction xs*; *force*)

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

lemma *index-not-in-delete-None* [*simp*]:
  assumes $i \notin$ *fst* ‘ *set xs*
  shows *delete xs i* = *None*
using *assms* **by**(*induction xs*, *auto*)

5.3 Preservation of element indices

lemma *insert-body-preserve-indices* [*simp*]:
  shows *fst* ‘ *set* (*insert-body xs e*) = *fst* ‘ *set xs* $\cup$ {*fst e*}
by(*induction xs*, *auto simp add*: *insert-commute*)

lemma *insert-preserve-indices*:
  assumes $\exists$ *ys*. *insert xs e i* = *Some ys*
  shows *fst* ‘ *set* (*the* (*insert xs e i*)) = *fst* ‘ *set xs* $\cup$ {*fst e*}
using *assms* **by**(*induction xs*; *cases i*; *auto simp add*: *insert-commute split*: *bind-splits*)

corollary *insert-preserve-indices′*:
  assumes *insert xs e i* = *Some ys*
  shows *fst* ‘ *set* (*the* (*insert xs e i*)) = *fst* ‘ *set xs* $\cup$ {*fst e*}
using *assms insert-preserve-indices* **by** *blast*

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

5.4 Commutativity of concurrent operations

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

lemma *insert-insert-body*:
  assumes *fst e1* $\neq$ *fst e2*
    and *i2* $\neq$ *Some* (*fst e1*)
  shows *insert* (*insert-body xs e1*) *e2 i2* = *insert xs e2 i2* $\gg=$ (λ*ys*. *Some* (*insert-body ys e1*))
using *assms* **by** (*induction xs*; *cases i2*) (*auto split*: *if-split-asm simp add*: *insert-body-commutes*)

lemma *insert-Nil-None*:
  assumes *fst e1* $\neq$ *fst e2*
    and *i* $\neq$ *fst e2*
    and *i2* $\neq$ *Some* (*fst e1*)
  shows *insert* [] *e2 i2* $\gg=$ (λ*ys*. *insert ys e1* (*Some i*)) = *None*
using *assms* **by** (*cases i2*) *clarsimp+*

lemma *insert-insert-body-commute*:
  assumes *i* $\neq$ *fst e1*
    and *fst e1* $\neq$ *fst e2*
  shows *insert* (*insert-body xs e1*) *e2* (*Some i*) =
      insert xs e2 (*Some i*) $\gg=$ (λ*y*. *Some* (*insert-body y e1*))
using *assms* **by**(*induction xs*, *auto simp add*: *insert-body-commutes*)

lemma *insert-commutes*:
  assumes *fst e1* $\neq$ *fst e2*
    i1 = *None* $\vee$ *i1* $\neq$ *Some* (*fst e2*)
    i2 = *None* $\vee$ *i2* $\neq$ *Some* (*fst e1*)
  shows *insert xs e1 i1* $\gg=$ (λ*ys. insert ys e2 i2*) =
    insert xs e2 i2 $\gg=$ (λ*ys. insert ys e1 i1*)
using *assms* **proof**(*induction rule*: *insert.induct*)
  fix *xs* **and** *e* :: ('*a*, '*b*) *elt*
  assume *i2* = *None* $\vee$ *i2* $\neq$ *Some* (*fst e*) **and** *fst e* $\neq$ *fst e2*
  thus *insert xs e None* $\gg=$ (λ*ys. insert ys e2 i2*) = *insert xs e2 i2* $\gg=$ (λ*ys. insert ys e None*)
    by(*auto simp add*: *insert-body-commutes intro*: *insert-insert-body*)
next
  fix *i* **and** *e* :: ('*a*, '*b*) *elt*
  assume *fst e* $\neq$ *fst e2* **and** *i2* = *None* $\vee$ *i2* $\neq$ *Some* (*fst e*) **and** *Some i* = *None* $\vee$ *Some i* $\neq$ *Some* (*fst e2*)
  thus *insert* [] *e* (*Some i*) $\gg=$ (λ*ys. insert ys e2 i2*) = *insert* [] *e2 i2* $\gg=$ (λ*ys. insert ys e* (*Some i*))
    by (*auto intro*: *insert-Nil-None*[*symmetric*])
next
  fix *xs i* **and** *x e* :: ('*a*, '*b*) *elt*
  assume *IH*: (*fst x* $\neq$ *i* $\Longrightarrow$
      fst e $\neq$ *fst e2* $\Longrightarrow$
      Some i = *None* $\vee$ *Some i* $\neq$ *Some* (*fst e2*) $\Longrightarrow$
      i2 = *None* $\vee$ *i2* $\neq$ *Some* (*fst e*) $\Longrightarrow$
      insert xs e (*Some i*) $\gg=$ (λ*ys. insert ys e2 i2*) = *insert xs e2 i2* $\gg=$ (λ*ys. insert ys e* (*Some i*)))
    and *fst e* $\neq$ *fst e2*
    and *Some i* = *None* $\vee$ *Some i* $\neq$ *Some* (*fst e2*)
    and *i2* = *None* $\vee$ *i2* $\neq$ *Some* (*fst e*)
  thus *insert* (*x* # *xs*) *e* (*Some i*) $\gg=$ (λ*ys. insert ys e2 i2*) = *insert* (*x* # *xs*) *e2 i2* $\gg=$ (λ*ys. insert ys e* (*Some i*))
    apply −
    apply(*erule disjE*, *clarsimp*, *simp*, *rule conjI*)
    apply(*case-tac i2*; *force simp add*: *insert-body-commutes insert-insert-body-commute*)
    apply(*case-tac i2*; *clarsimp cong*: *Option.bind-cong simp add*: *insert-insert-body split*: *bind-splits*)
    apply *force*
    done
qed

lemma *delete-commutes*:
  shows *delete xs i1* $\gg=$ (λ*ys. delete ys i2*) = *delete xs i2* $\gg=$ (λ*ys. delete ys i1*)
by(*induction xs*, *auto split*: *bind-splits if-split-asm*)

lemma *insert-body-delete-commute*:
  assumes *i2* $\neq$ *fst e*
  shows *delete* (*insert-body xs e*) *i2* $\gg=$ (λ*t. Some* (*x*#*t*)) =
    delete xs i2 $\gg=$ (λ*y. Some* (*x*#*insert-body y e*))
using *assms* **by** (*induction xs arbitrary*: *x*; *cases e*, *auto split*: *bind-splits if-split-asm*)

lemma *insert-delete-commute*:
  assumes *i2* $\neq$ *fst e*
  shows *insert xs e i1* $\gg=$ (λ*ys. delete ys i2*) = *delete xs i2* $\gg=$ (λ*ys. insert ys e i1*)
using *assms* **by**(*induction xs*; *cases e*; *cases i1*, *auto split*: *bind-splits if-split-asm simp add*: *insert-body-delete-commute*)

5.5 Alternative definition of insert

fun *insert'* :: ('*id*::{*linorder*}, '*v*) *elt list* $\Rightarrow$ ('*id*, '*v*) *elt* $\Rightarrow$ '*id option* $\rightharpoonup$ ('*id*::{*linorder*}, '*v*) *elt list*
where
  insert' [] *e None* = *Some* [*e*] |

```
  insert′ [] e      (Some i) = None |
  insert′ (x#xs) e None      =
     (if fst x < fst e then
        Some (e#x#xs)
      else
        case insert′ xs e None of
          None   ⇒ None
        | Some t ⇒ Some (x#t)) |
  insert′ (x#xs) e (Some i) =
     (if fst x = i then
        case insert′ xs e None of
          None   ⇒ None
        | Some t ⇒ Some (x#t)
      else
        case insert′ xs e (Some i) of
          None   ⇒ None
        | Some t ⇒ Some (x#t))

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

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

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

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

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

lemma insert-between-elements:
  assumes xs = pre@ref#suf
      and distinct (map fst xs)
      and ⋀i′. i′ ∈ fst ‘ set xs ⟹ i′ < fst e
    shows insert xs e (Some (fst ref)) = Some (pre @ ref # e # suf)
  using assms by(induction xs arbitrary: pre ref suf, force) (case-tac pre; case-tac suf; force)
```

```
lemma insert-position-element-technical:
  assumes ∀x∈set as. a ≠ fst x
    and insert-body (cs @ ds) e = cs @ e # ds
  shows insert (as @ (a, aa, b) # cs @ ds) e (Some a) = Some (as @ (a, aa, b) # cs @ e # ds)
using assms by (induction as arbitrary: cs ds; clarsimp)

lemma split-tuple-list-by-id:
  assumes (a,b,c) ∈ set xs
    and distinct (map fst xs)
  shows ∃pre suf. xs = pre @ (a,b,c) # suf ∧ (∀y ∈ set pre. fst y ≠ a)
using assms proof(induction xs, clarsimp)
  case (Cons x xs)
  { assume x ≠ (a, b, c)
    hence (a, b, c) ∈ set xs distinct (map fst xs)
      using Cons.prems by force+
    then obtain pre suf where xs = pre @ (a, b, c) # suf ∧ (∀y∈set pre. fst y ≠ a)
      using Cons.IH by force
    hence ?case
      apply(rule-tac x=x#pre in exI)
      using Cons.prems(2) by auto
  } thus ?case
    by force
qed

lemma insert-preserves-order:
  assumes i = None ∨ (∃i'. i = Some i' ∧ i' ∈ fst ` set xs)
      and distinct (map fst xs)
    shows ∃pre suf. xs = pre@suf ∧ insert xs e i = Some (pre @ e # suf)
  using assms proof −
  { assume i = None
    hence ?thesis
      by clarsimp (metis insert-body-contains-new-elem)
  } moreover {
    assume ∃i'. i = Some i' ∧ i' ∈ fst ` set xs
    then obtain j v b where i = Some j (j, v, b) ∈ set xs by force
    moreover then obtain as bs where xs = as@(j,v,b)#bs ∀x ∈ set as. fst x ≠ j
      using assms by (metis split-tuple-list-by-id)
    moreover then obtain cs ds where insert-body bs e = cs@e#ds cs@ds = bs
      by(metis insert-body-contains-new-elem)
    ultimately have ?thesis
      by(rule-tac x=as@(j,v,b)#cs in exI; clarsimp)(metis insert-position-element-technical)
  } ultimately show ?thesis
    using 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 Deliver (i, Insert e n) ∈ set xs
  shows   fst e ∈ set (indices xs)
using assms by(induction xs, auto simp add: indices-def map-filter-def)

lemma (in rga) apply-opers-idx-elems:
  assumes es prefix of i
      and apply-operations es = Some xs
    shows element-ids xs = set (indices es)
using assms unfolding element-ids-def
proof(induction es arbitrary: xs rule: rev-induct, clarsimp)
  case (snoc x xs) thus ?case
  proof (cases x, clarsimp, blast)
    case (Deliver e)
    moreover obtain a b where e = (a, b) by force
    ultimately show ?thesis
```

```
      using snoc assms apply (cases b; clarsimp split: bind-splits simp add: interp-msg-def)
       apply (metis Un-insert-right append.right-neutral insert-preserve-indices' list.set(1)
             option.sel prefix-of-appendD prod.sel(1) set-append)
      by (metis delete-preserve-indices prefix-of-appendD)
  qed
qed

lemma (in rga) delete-does-not-change-element-ids:
  assumes es @ [Deliver (i, Delete n)] prefix of j
  and apply-operations es = Some xs1
  and apply-operations (es @ [Deliver (i, Delete n)]) = Some xs2
  shows element-ids xs1 = element-ids xs2
proof -
  have indices es = indices (es @ [Deliver (i, Delete n)])
    by simp
  then show ?thesis
    by (metis (no-types) assms prefix-of-appendD rga.apply-opers-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 apply-opers-idx-elems by auto

lemma (in rga) deliver-insert-exists:
  assumes es prefix of j
      and apply-operations es = Some xs
      and a ∈ element-ids xs
    shows ∃ i v f n. Deliver (i, Insert (a, v, f) n) ∈ set es
using assms unfolding element-ids-def
proof(induction es arbitrary: xs rule: rev-induct, clarsimp)
  case (snoc x xs ys) thus ?case
  proof (cases x)
    case (Broadcast e) thus ?thesis
      using snoc by(clarsimp, metis image-eqI prefix-of-appendD prod.sel(1))
  next
    case (Deliver e)
    moreover then obtain xs' where *: apply-operations xs = Some xs'
        using snoc by fastforce
    moreover obtain k v where **: e = (k, v) by force
    ultimately show ?thesis
      using assms snoc proof (cases v)
      case (Insert el -) thus ?thesis
        using snoc Deliver * **
        apply (cases el; cases fst el = a; clarsimp)
        apply (blast, metis (no-types, lifting) element-ids-def prefix-of-appendD set-map snoc.prems(2)
                  snoc.prems(3) someone-inserted-id)
        done
      next
      case (Delete -) thus ?thesis
        using snoc Deliver ** apply clarsimp
        apply(drule prefix-of-appendD, clarsimp simp add: bind-eq-Some-conv interp-msg-def)
        apply(metis delete-preserve-indices image-eqI prod.sel(1))
        done
```

qed
qed
qed

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

lemma (**in** *rga*) *insert-msg-id*:
assumes *Broadcast* (*i*, *Insert e n*) $\in$ *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-asm*)
qed

lemma (**in** *rga*) *allowed-insert*:
assumes *Broadcast* (*i*, *Insert e n*) $\in$ *set* (*history j*)
shows *n* = *None* $\vee$ ($\exists\, i'\ e'\ n'$. *n* = *Some* (*fst* e') $\wedge$ *Deliver* (i', *Insert* e' n') $\sqsubset^j$ *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* $\vee$ ($\exists\, i'\ e'\ n'$. *n* = *Some* (*fst* e') $\wedge$ *Deliver* (i', *Insert* e' n') $\sqsubset^j$ *Broadcast* (*i*, *Insert e n*))
proof(*cases n*)
fix *a*
assume *4*: *n* = *Some a*
hence *a* $\in$ *element-ids state* **and** *5*: *fst e* = *i*
using *3* **by**(*clarsimp simp add*: *valid-rga-msg-def*)+
from *this* **have** $\exists\, i'\ v'\ f'\ n'$. *Deliver* (i', *Insert* (*a*, v', f') n') $\in$ *set pre*
using *deliver-insert-exists 2 1* **by** *blast*
thus *n* = *None* $\vee$ ($\exists\, i'\ e'\ n'$. *n* = *Some* (*fst* e') $\wedge$ *Deliver* (i', *Insert* e' n') $\sqsubset^j$ *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*) $\in$ *set* (*history j*)
shows $\exists\, i'\ n'\ v\ b$. *Deliver* (i', *Insert* (*x*, *v*, *b*) n') $\sqsubset^j$ *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* $\in$ *element-ids state*
using *apply-opers-idx-elems* **by**(*simp add*: *valid-rga-msg-def*)
hence $\exists\, i'\ v'\ f'\ n'$. *Deliver* (i', *Insert* (*x*, v', f') n') $\in$ *set pre*

using *deliver-insert-exists 1 2* **by** *blast*
thus $\exists\, i'\ n'\ v\ b.\ \mathit{Deliver}\ (i',\ \mathit{Insert}\ (x,\ v,\ b)\ n') \sqsubset^j \mathit{Broadcast}\ (i,\ \mathit{Delete}\ x)$
using *events-in-local-order 1* **by** *blast*
qed

lemma (**in** *rga*) *insert-id-unique*:
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 *assms insert-msg-id msg-id-unique Pair-inject fst-conv* **by** *metis*

lemma (**in** *rga*) *allowed-delete-deliver*:
assumes *Deliver* (*i*, *Delete x*) ∈ *set* (*history j*)
shows $\exists\, i'\ n'\ v\ b.\ \mathit{Deliver}\ (i',\ \mathit{Insert}\ (x,\ v,\ b)\ n') \sqsubset^j \mathit{Deliver}\ (i,\ \mathit{Delete}\ x)$
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 (*es*@[*Deliver* (*i*, *Delete m*)]) *prefix of j*
shows $\exists\, i'\ n\ v\ b.\ \mathit{Deliver}\ (i',\ \mathit{Insert}\ (m,\ v,\ b)\ n) \in \mathit{set}\ \mathit{es}$
by(*metis* (*no-types*, *lifting*) *Un-insert-right insert-iff list.simps*(*15*) *assms*
local-order-prefix-closed-last rga.allowed-delete-deliver rga-axioms set-append subsetCE prefix-to-carriers)

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

lemma (**in** *rga*) *allowed-insert-deliver-in-set*:
assumes (*es*@[*Deliver* (*i*, *Insert e m*)]) *prefix of j*
shows $m = \mathit{None} \lor (\exists\, i'\ m'\ n\ v\ b.\ m = \mathit{Some}\ m' \land \mathit{Deliver}\ (i',\ \mathit{Insert}\ (m',\ v,\ b)\ n) \in \mathit{set}\ \mathit{es})$
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*
proof −
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(*frule delivery-has-a-cause*, *clarsimp*)

apply(*frule allowed-insert*)
apply *clarsimp*
apply(*metis Insert-equal delivery-has-a-cause fst-conv hb.intros*(*2*) *insert-subset local-order-carrier-closed insert-msg-id*)
done

lemma *subset-reorder*:
assumes $\{a, b\} \subseteq c$
shows $\{b, a\} \subseteq c$
using *assms* **by** *simp*

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

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

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

lemma (**in** *rga*) *concurrent-operations-commute*:
assumes *xs prefix of i*
shows *hb.concurrent-ops-commute* (*node-deliver-messages xs*)
proof −
have $\bigwedge x\ y.\ \{x, y\} \subseteq set\ (node\text{-}deliver\text{-}messages\ xs) \Longrightarrow hb.concurrent\ x\ y \Longrightarrow interp\text{-}msg\ x \rhd interp\text{-}msg\ y = interp\text{-}msg\ y \rhd interp\text{-}msg\ x$
proof
fix *x y ii*
assume $\{x, y\} \subseteq set\ (node\text{-}deliver\text{-}messages\ xs)$
and C: *hb.concurrent x y*
hence X: $x \in set\ (node\text{-}deliver\text{-}messages\ xs)$ **and** Y: $y \in set\ (node\text{-}deliver\text{-}messages\ xs)$
by *auto*
obtain *x1 x2 y1 y2* **where** *1*: $x = (x1, x2)$ **and** *2*: $y = (y1, y2)$
by *fastforce*
have $(interp\text{-}msg\ (x1, x2) \rhd interp\text{-}msg\ (y1, y2))\ ii = (interp\text{-}msg\ (y1, y2) \rhd interp\text{-}msg\ (x1, x2))\ ii$
proof(*cases x2*; *cases y2*)
fix *ix1 ix2 iy1 iy2*
assume $X2$: $x2 = Insert\ ix1\ ix2$ **and** $Y2$: $y2 = Insert\ iy1\ iy2$
show $(interp\text{-}msg\ (x1, x2) \rhd interp\text{-}msg\ (y1, y2))\ ii = (interp\text{-}msg\ (y1, y2) \rhd interp\text{-}msg\ (x1, x2))\ ii$
proof(*cases fst ix1 = fst iy1*)
assume *fst ix1 = fst iy1*
hence *Insert ix1 ix2 = 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** (*interp-msg* (*x1*, *x2*) ▷ *interp-msg* (*y1*, *y2*)) *ii* = (*interp-msg* (*y1*, *y2*) ▷ *interp-msg* (*x1*, *x2*)) *ii*
by(*clarsimp simp add*: *kleisli-def interp-msg-def*)
next
assume *NEQ*: *fst ix1* ≠ *fst iy1*
have *ix2* = *None* ∨ *ix2* ≠ *Some* (*fst iy1*)
apply(*rule insert-commute-assms*)
using *prefix-msg-in-history*[*OF assms*] *X Y X2 Y2 1 2*
apply(*clarsimp*, *blast*)
using *C 1 2 X2 Y2* **apply** *blast*
done
also have *iy2* = *None* ∨ *iy2* ≠ *Some* (*fst ix1*)
apply(*rule insert-commute-assms*)
using *prefix-msg-in-history*[*OF assms*] *X Y X2 Y2 1 2*
apply(*clarsimp*, *blast*)
using *1 2 C X2 Y2* **apply** *blast*
done
ultimately have *insert ii ix1 ix2* ≫= (λ*x*. *insert x iy1 iy2*) = *insert ii iy1 iy2* ≫= (λ*x*. *insert x ix1 ix2*)
using *NEQ insert-commutes* **by** *blast*
thus (*interp-msg* (*x1*, *x2*) ▷ *interp-msg* (*y1*, *y2*)) *ii* = (*interp-msg* (*y1*, *y2*) ▷ *interp-msg* (*x1*, *x2*)) *ii*
by(*clarsimp simp add*: *interp-msg-def X2 Y2 kleisli-def*)
qed
next
fix *ix1 ix2 yd*
assume *X2*: *x2* = *Insert ix1 ix2* **and** *Y2*: *y2* = *Delete yd*
thm *insert-delete-commute*
thm *Insert-Delete-concurrent*
have *hb.concurrent* (*x1*, *Insert ix1 ix2*) (*y1*, *Delete yd*)
using *C X2 Y2 1 2* **by** *simp*
also have {*Deliver* (*x1*, *Insert ix1 ix2*), *Deliver* (*y1*, *Delete yd*)} ⊆ *set* (*history i*)
using *prefix-msg-in-history assms X2 Y2 X Y 1 2* **by** *blast*
ultimately have *yd* ≠ *fst ix1*
apply −
apply(*rule Insert-Delete-concurrent*; *force*)
done
hence *insert ii ix1 ix2* ≫= (λ*x*. *delete x yd*) = *delete ii yd* ≫= (λ*x*. *insert x ix1 ix2*)
by(*rule insert-delete-commute*)
thus (*interp-msg* (*x1*, *x2*) ▷ *interp-msg* (*y1*, *y2*)) *ii* = (*interp-msg* (*y1*, *y2*) ▷ *interp-msg* (*x1*, *x2*)) *ii*
by(*clarsimp simp add*: *interp-msg-def kleisli-def X2 Y2*)
next
fix *xd iy1 iy2*
assume *X2*: *x2* = *Delete xd* **and** *Y2*: *y2* = *Insert iy1 iy2*
have *hb.concurrent* (*x1*, *Delete xd*) (*y1*, *Insert iy1 iy2*)
using *C X2 Y2 1 2* **by** *simp*
also have {*Deliver* (*x1*, *Delete xd*), *Deliver* (*y1*, *Insert iy1 iy2*)} ⊆ *set* (*history i*)
using *prefix-msg-in-history assms X2 Y2 X Y 1 2* **by** *blast*
ultimately have *xd* ≠ *fst iy1*
apply −
apply(*rule Insert-Delete-concurrent*; *force*)
done
hence *delete ii xd* ≫= (λ*x*. *insert x iy1 iy2*) = *insert ii iy1 iy2* ≫= (λ*x*. *delete x xd*)
by(*rule insert-delete-commute*[*symmetric*])
thus (*interp-msg* (*x1*, *x2*) ▷ *interp-msg* (*y1*, *y2*)) *ii* = (*interp-msg* (*y1*, *y2*) ▷ *interp-msg* (*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* $\gg=$ (λx. *delete x yd*) = *delete ii yd* $\gg=$ (λx. *delete x xd*)
        by(*rule delete-commutes*)
      thus (*interp-msg* (*x1*, *x2*) $\rhd$ *interp-msg* (*y1*, *y2*)) *ii* = (*interp-msg* (*y1*, *y2*) $\rhd$ *interp-msg* (*x1*, *x2*)) *ii*
        by(*clarsimp simp add*: *interp-msg-def kleisli-def X2 Y2*)
    qed
    thus (*interp-msg x* $\rhd$ *interp-msg y*) *ii* = (*interp-msg y* $\rhd$ *interp-msg x*) *ii*
      using *1 2* by *auto*
  qed
  thus *hb.concurrent-ops-commute* (*node-deliver-messages xs*)
    by(*auto simp add*: *hb.concurrent-ops-commute-def*)
qed

corollary (in *rga*) *concurrent-operations-commute′*:
  shows *hb.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 *x xs*
  assume *1*: *xs prefix of i* $\Longrightarrow$ *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 $\exists$ *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 $\exists$ *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 = Delete x2
      show ∃ y. apply-operations (xs @ [Deliver (a, Delete x2)]) = 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 (history i) ≠ None
by(meson apply-operations-never-fails append.right-neutral prefix-of-node-history-def)

corollary (in rga) rga-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)
```

5.7 Strong eventual consistency

```
context rga begin

sublocale sec: strong-eventual-consistency weak-hb hb interp-msg
 λops.∃ xs i. xs prefix of i ∧ node-deliver-messages xs = ops []
proof(standard; clarsimp)
 fix xsa i
 assume xsa prefix of i
 thus hb.hb-consistent (node-deliver-messages xsa)
  by(auto simp add: hb-consistent-prefix)
next
 fix xsa i
 assume xsa prefix of i
 thus distinct (node-deliver-messages xsa)
  by(auto simp add: node-deliver-messages-distinct)
next
 fix xsa i
 assume xsa prefix of i
 thus hb.concurrent-ops-commute (node-deliver-messages xsa)
  by(auto simp add: concurrent-operations-commute)
next
 fix xs a b state xsa x
 assume hb.apply-operations xs [] = Some state
  and node-deliver-messages xsa = xs @ [(a, b)]
  and xsa prefix of x
 thus ∃ y. interp-msg (a, b) state = Some y
  by(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)
next
 fix xs a b xsa x
 assume node-deliver-messages xsa = xs @ [(a, b)]
  and xsa prefix of x
 thus ∃ xsa. Ex (op prefix of xsa) ∧ node-deliver-messages xsa = xs
```

  using *drop-last-message* **by** *blast*
qed

end

interpretation *trivial-rga-implementation*: *rga* $\lambda x.$ []
  by(*standard*, *auto simp add*: *trivial-node-histories.history-order-def*
    trivial-node-histories.prefix-of-node-history-def)

end

6 Increment-Decrement Counter

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

theory
  Counter
imports
  Network
begin

datatype *operation* = *Increment* | *Decrement*

fun *counter-op* :: *operation* $\Rightarrow$ *int* $\rightharpoonup$ *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* $\rhd$ *counter-op y* = *counter-op y* $\rhd$ *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*
  λ*ops*. $\exists$*xs i*. *xs prefix of i* $\wedge$ *node-deliver-messages xs* = *ops 0*
  apply(*standard*; *clarsimp simp add*: *hb-consistent-prefix drop-last-message*
    node-deliver-messages-distinct concurrent-operations-commute)
   apply(*metis* (*full-types*) *interp-msg-def counter-op.elims*)
  using *drop-last-message* **apply** *blast*

done

end
end

7 Observed-Remove Set

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

theory
  ORSet
imports
  Network
begin

datatype $('id, 'a)$ *operation* = *Add 'id 'a* | *Rem 'id set 'a*

type-synonym $('id, 'a)$ *state* = *'a* $\Rightarrow$ *'id set*

definition *op-elem* :: $('id, 'a)$ *operation* $\Rightarrow$ *'a* **where**
  op-elem oper $\equiv$ *case oper of Add i e* $\Rightarrow$ *e* | *Rem is e* $\Rightarrow$ *e*

definition *interpret-op* :: $('id, 'a)$ *operation* $\Rightarrow$ $('id, 'a)$ *state* $\rightharpoonup$ $('id, 'a)$ *state* $(\langle$-$\rangle$ [0] 1000) **where**
  interpret-op oper state $\equiv$
    let before = state (op-elem oper);
        after = case oper of Add i e $\Rightarrow$ *before* $\cup$ $\{i\}$ | *Rem is e* $\Rightarrow$ *before* $-$ *is*
    in Some (state ((op-elem oper) := after))

definition *valid-behaviours* :: $('id, 'a)$ *state* $\Rightarrow$ *'id* $\times$ $('id, 'a)$ *operation* $\Rightarrow$ *bool* **where**
  valid-behaviours state msg $\equiv$
    case msg of
      (i, Add j e) $\Rightarrow$ *i = j* |
      (i, Rem is e) $\Rightarrow$ *is = state e*

locale *orset* = *network-with-constrained-ops - interpret-op* $\lambda x.$ $\{\}$ *valid-behaviours*

lemma (**in** *orset*) *add-add-commute*:
  shows $\langle$*Add i1 e1*$\rangle$ $\rhd$ $\langle$*Add i2 e2*$\rangle$ = $\langle$*Add i2 e2*$\rangle$ $\rhd$ $\langle$*Add i1 e1*$\rangle$
  by(*auto simp add*: *interpret-op-def op-elem-def kleisli-def*, *fastforce*)

lemma (**in** *orset*) *add-rem-commute*:
  assumes *i* $\notin$ *is*
  shows $\langle$*Add i e1*$\rangle$ $\rhd$ $\langle$*Rem is e2*$\rangle$ = $\langle$*Rem is e2*$\rangle$ $\rhd$ $\langle$*Add i e1*$\rangle$
  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* $\neq$ *None*
using *assms* **proof**(*induction xs rule*: *rev-induct*, *clarsimp*)
  case (*snoc x xs*) **thus** *?case*
  proof (*cases x*)
    case (*Broadcast e*) **thus** *?thesis*
      using *snoc* **by** *force*
  next
    case (*Deliver e*) **thus** *?thesis*
      using *snoc* **by** (*clarsimp*, *metis interpret-op-def interp-msg-def bind.bind-lunit prefix-of-appendD*)
  qed

qed

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

definition (**in** *orset*) *added-ids* :: (′*id* × (′*id*, ′*b*) *operation*) *event list* ⇒ ′*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*)

qed
qed

lemma (**in** *orset*) *Deliver-added-ids*:
assumes *xs prefix of j*
and $i \in$ *set* (*added-ids xs e*)
shows *Deliver* (*i*, *Add i e*) $\in$ *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
qed

lemma (**in** *orset*) *Broadcast-Deliver-prefix-closed*:
assumes *xs* @ [*Broadcast* (*r*, *Rem ix e*)] *prefix of j*
and $i \in ix$
shows *Deliver* (*i*, *Add i e*) $\in$ *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*) $\in$ *set xs*
and $i \in ix$
shows *Deliver* (*i*, *Add i e*) $\in$ *set xs*
using *assms Broadcast-Deliver-prefix-closed* **by** (*induction xs rule*: *rev-induct*; *force*)

lemma (**in** *orset*) *concurrent-add-remove-independent-technical*:
assumes $i \in is$
and *xs prefix of j*
and (*i*, *Add i e*) $\in$ *set* (*node-deliver-messages xs*) **and** (*ir*, *Rem is e*) $\in$ *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*) $\in$ *set pre*
using *Broadcast-Deliver-prefix-closed assms*(*1*) **by** *auto*
ultimately show *?thesis*
using *hb.intros*(*2*) *events-in-local-order* **by** *blast*
qed

lemma (**in** *orset*) *Deliver-Add-same-id-same-message*:

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

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

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

lemma (**in** *orset*) *rem-rem-commute*:
shows ⟨*Rem i1 e1*⟩ ▷ ⟨*Rem i2 e2*⟩ = ⟨*Rem i2 e2*⟩ ▷ ⟨*Rem i1 e1*⟩
by(*unfold interpret-op-def op-elem-def kleisli-def*, *fastforce*)

lemma (**in** *orset*) *concurrent-operations-commute*:
assumes *xs prefix of i*
shows *hb.concurrent-ops-commute* (*node-deliver-messages xs*)
proof −
{ **fix** *a b x y*
assume (*a*, *b*) ∈ *set* (*node-deliver-messages xs*)
(*x*, *y*) ∈ *set* (*node-deliver-messages xs*)
hb.concurrent (*a*, *b*) (*x*, *y*)
hence *interp-msg* (*a*, *b*) ▷ *interp-msg* (*x*, *y*) = *interp-msg* (*x*, *y*) ▷ *interp-msg* (*a*, *b*)
apply(*unfold interp-msg-def*, *case-tac b*; *case-tac y*; *simp add*: *add-add-commute rem-rem-commute hb.concurrent-def*)
apply (*metis add-id-valid add-rem-commute assms concurrent-add-remove-independent hb.concurrentD1 hb.concurrentD2 prefix-contains-msg*)+
done
} **thus** *?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* $i \wedge$ *node-deliver-messages* $xs = ops\ \lambda x.\{\}$
apply(*standard*; *clarsimp simp add*: *hb-consistent-prefix node-deliver-messages-distinct concurrent-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*
done

end
end

References

[1] P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based CRDTs by delta-mutation. In *International Conference on Networked Systems (NETYS)*, May 2015.

[2] C. Baquero, P. S. Almeida, and A. Shoker. Making operation-based CRDTs operation-based. In *14th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 126–140, June 2014.

[3] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak DT map: a composable, convergent replicated dictionary. In *1st Workshop on Principles and Practice of Eventual Consistency (PaPEC)*, Apr. 2014.

[4] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer, second edition, Feb. 2011.

[5] J. Day-Richter. What's different about the new Google Docs: Making collaboration fast, Sept. 2010.

[6] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *8th European Conference on Computer-Supported Cooperative Work (ECSCW)*, pages 277–293, Sept. 2003.

[7] A. Imine, M. Rusinowitch, G. Oster, and P. Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science*, 351(2):167–183, Feb. 2006.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[9] G. Oster, P. Urso, P. Molli, and A. Imine. Proving correctness of transformation functions in collaborative editing systems. Technical Report RR-5795, Dec. 2005.

[10] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[11] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report 7506, INRIA, 2011.

[12] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 386–400, Oct. 2011.

[13] M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle framework. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, pages 33–38, 2008.