Verifying Strong Eventual Consistency in Distributed Systems

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

April 14, 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

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
~~/src/HOL/Library/Monad-Syntax
begin

2.1 Kleisli arrow composition

definition *kleisli* :: ('*b* ⇒ '*b option*) ⇒ ('*b* ⇒ '*b option*) ⇒ ('*b* ⇒ '*b option*) (**infixr** ▷ *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 $x = y \vee y \in set\ xs$
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 $\exists\, m.\ m < length\ xs \wedge xs\ !\ m = x$
using *assms* **by**(*induction xs*, *simp*) (*meson in-set-conv-nth*)

2.3 Lemmas about list

lemma *list-nil-or-snoc*:
shows $xs = [] \vee (\exists\, y\ ys.\ xs = ys@[y])$
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 ≠ []
    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(rule-tac x=[] in exI, rule-tac x=hd xs in exI)
  apply(simp-all add: list-head-length-one)
  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 $\exists\, xs\ ys\ zs.\ cs = xs@(cs!n)\#ys@(cs!m)\#zs$
using *assms*
  apply(*induction n arbitrary*: *cs m*)
  apply(*rule-tac x*=[] **in** *exI*, *clarsimp*)
  apply(*case-tac cs*; *clarsimp*)
  apply(*rule list-nth-split-technical*, *simp*, *force*)
  apply(*case-tac cs*; *clarsimp*)
  apply(*erule-tac x*=*list* **in** *meta-allE*, *erule-tac x*=*m*−*1* **in** *meta-allE*)
  apply(*subgoal-tac m*−*1* < *length list*, *subgoal-tac n*<*m*−*1*, *clarsimp*)
  apply(*rule-tac x*=*a*#*xs* **in** *exI*, *rule-tac x*=*ys* **in** *exI*, *rule-tac x*=*zs* **in** *exI*)
  apply *force*+
done

lemma *list-split-two-elems*:
  assumes *distinct cs*
    and $x \in set\ cs$
    and $y \in set\ cs$
    and $x \neq y$
   shows $\exists\, pre\ mid\ suf.\ cs = pre\ @\ x\ \#\ mid\ @\ y\ \#\ suf \vee cs = pre\ @\ y\ \#\ mid\ @\ x\ \#\ suf$
  using *assms*
  apply(*subgoal-tac* $\exists\, xi.\ xi < length\ cs \wedge x = cs\ !\ xi$)
  apply(*subgoal-tac* $\exists\, yi.\ yi < length\ cs \wedge y = cs\ !\ yi$)
  apply *clarsimp*
  apply(*subgoal-tac xi* ≠ *yi*)
  apply(*case-tac xi* < *yi*)
  apply(*metis list-nth-split One-nat-def less-Suc-eq linorder-neqE-nat not-less-zero*)
  apply(*subgoal-tac yi* < *xi*)
  apply(*metis list-nth-split One-nat-def less-Suc-eq linorder-neqE-nat not-less-zero*)
  using *set-elem-nth linorder-neqE-nat* **apply** *fastforce*+
done

lemma *split-list-unique-prefix*:
  assumes $x \in set\ xs$
  shows $\exists\, pre\ suf.\ xs = pre\ @\ x\ \#\ suf \wedge (\forall\, y \in set\ pre.\ x \neq y)$
using *assms*
  apply(*induction xs*; *clarsimp*)
  apply(*case-tac a* = *x*)
  apply(*rule-tac x*=[] **in** *exI*, *force*)
  apply(*subgoal-tac x* ∈ *set xs*, *clarsimp*)
  apply(*rule-tac x*=*a* # *pre* **in** *exI*)
  apply *force*+
done

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 $x \parallel y$, 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** $\parallel$ *50*) **where**
$s1 \parallel s2 \equiv \neg\,(s1 \prec s2) \wedge \neg\,(s2 \prec s1)$

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

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

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

lemma [*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 [*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 ∃ *prefix suffix*. *ys* @ [*y*] = *prefix* @ *x* # *suffix* ∧ *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 ∈ *set xs*
y ∈ *set ys*
shows ¬ *y* ≺ *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* ⇒ *'b* ⇀ *'b* **where**
apply-operations es ≡ *foldl* (*op* ▷) *Some* (*map interp es*)

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

lemma *apply-operations-Snoc* [*simp*]:
apply-operations (*xs*@[*x*]) = (*apply-operations xs*) ▷ ⟨*x*⟩
by(*auto simp add*: *apply-operations-def kleisli-def*)

3.4 Concurrent operations commute

We say that two operations *x* and *y commute* whenever ⟨*x*⟩ ▷ ⟨*y*⟩ = ⟨*y*⟩ ▷ ⟨*x*⟩, 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* ⇒ *bool* **where**
concurrent-ops-commute xs ≡
∀ *x y*. {*x*, *y*} ⊆ *set xs* ⟶ *concurrent x y* ⟶ ⟨*x*⟩▷⟨*y*⟩ = ⟨*y*⟩▷⟨*x*⟩

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 ≠ None
using assms
  apply(induction xs rule: rev-induct, simp)
  apply(subgoal-tac apply-operations xs initial-state ≠ None)
```

apply(*subgoal-tac apply-operations* (*xs* @ [*x*]) = *apply-operations xs* ▷ ⟨*x*⟩)
apply(*simp add*: *kleisli-def bind-def*)
apply(*erule exE*, *case-tac* ⟨*x*⟩ *y*)
using *no-failure* **apply** *blast*
apply *simp+*
using *trunc-history* **apply** *blast*
done

end
end

4 Axiomatic network models

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

theory
Network
imports
Convergence
begin

4.1 Node histories

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

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

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

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

lemma (**in** *node-histories*) *node-total-order-trans*:
assumes *e1* ⊏^{i} *e2*
and *e2* ⊏^{i} *e3*
shows *e1* ⊏^{i} *e3*
using *assms* **unfolding** *history-order-def*
apply *clarsimp*
apply(*rule-tac x*=*xs* **in** *exI*, *rule-tac x*=*ys* @ *e2* # *ysa* **in** *exI*, *rule-tac x*=*zsa* **in** *exI*)
apply(*subgoal-tac xs* @ *e1* # *ys* = *xsa* ∧ *zs* = *ysa* @ *e3* # *zsa*)
apply *clarsimp*
apply(*rule-tac xs*=*history i* **and** *ys*=[*e2*] **in** *pre-suf-eq-distinct-list*)
apply *auto*

done

lemma (**in** *node-histories*) *local-order-carrier-closed*:
assumes *e1* $\sqsubset^i$ *e2*
shows {*e1*,*e2*} $\subseteq$ *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 $\neg$ (*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* $\in$ *set* (*history i*)
and *e2* $\in$ *set* (*history i*)
and *e1* $\neq$ *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* $\Rightarrow$ *nat* $\Rightarrow$ *bool* (**infix** *prefix of 50*) **where**
xs prefix of i $\equiv$ $\exists$ *ys*. *xs*@*ys* = *history i*

lemma (**in** *node-histories*) *carriers-head-lt*:
assumes *y*#*ys* = *history i*
shows $\neg$(*x* $\sqsubset^i$ *y*)
using *assms*
apply(*clarsimp simp add*: *history-order-def*)
apply (*subgoal-tac xs* @ *x* # *ysa* = [] $\wedge$ *zs* = *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*
using *assms*
apply −
apply (*frule prefix-distinct*)
apply (*insert histories-distinct*[**where** *i*=*i*])
apply (*clarsimp simp*: *history-order-def prefix-of-node-history-def*)
apply (*frule split-list*)
apply *clarsimp*
apply (*subgoal-tac ysb* = *xsa* @ *x* # *ysa* ∧ *zsa* @ *ys* = *zs*)
apply *clarsimp*
apply (*rule-tac xs*=*history i* **and** *ys*=[*y*] **in** *pre-suf-eq-distinct-list*)
apply *auto*
done

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*
using *assms*
apply −
apply(*frule local-order-prefix-closed*, *assumption*, *force*)
apply(*auto simp add*: *node-total-order-irrefl prefix-to-carriers*)
done

lemma (**in** *node-histories*) *events-before-exist*:
assumes $x \in$ *set* (*history i*)
shows ∃ *pre*. *pre* @ [*x*] *prefix of i*
using *assms* **unfolding** *prefix-of-node-history-def* **apply** −
apply(*subgoal-tac* ∃ *idx*. *idx* < *length* (*history i*) ∧ (*history i*) ! *idx* = *x*)
apply(*metis append-take-drop-id take-Suc-conv-app-nth*)
apply(*simp add*: *set-elem-nth*)
done

lemma (**in** *node-histories*) *events-in-local-order*:
assumes *pre* @ [*e2*] *prefix of i*
and *e1* ∈ *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 ∈ set (history i) ⟧ ⟹
                                  ∃ j. Broadcast m ∈ set (history j)
    and deliver-locally: ⟦ Broadcast m ∈ set (history i) ⟧ ⟹
                                  Broadcast m ⊏^i Deliver m
    and msg-id-unique: ⟦ Broadcast m1 ∈ set (history i);
                         Broadcast m2 ∈ set (history j);
                         msg-id m1 = msg-id m2 ⟧ ⟹ i = j ∧ m1 = m2
```

The axioms can be understood as follows:

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

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

msg-id-unique: We do not assume that the message type *'msg* has any particular structure; we only assume the existence of a function *msg-id*::*'msg*⇒*'msgid* that maps every message to some globally unique identifier of type *'msgid*. We assert this uniqueness by stating that if *m1* and *m2* are any two messages broadcast by any two nodes, and their *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 ∈ set (history i)
  shows ∃ j. Broadcast m ⊏^j Deliver m
  using assms deliver-locally delivery-has-a-cause by blast

lemma (in network) broadcasts-unique:
  assumes i ≠ j
    and Broadcast m ∈ set (history i)
  shows Broadcast m ∉ set (history j)
  using assms msg-id-unique by blast
```

Based on the well-known definition by [8], we say that *m1* ≺ *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* ≺ *m3* and *m3* ≺ *m2*.

```
inductive (in network) hb :: 'msg ⇒ 'msg ⇒ bool where
  ⟦ Broadcast m1 ⊏^i Broadcast m2 ⟧ ⟹ hb m1 m2 |
  ⟦ Deliver m1 ⊏^i Broadcast m2 ⟧ ⟹ hb m1 m2 |
```

⟦ *hb m1 m2*; *hb m2 m3* ⟧ ⟹ *hb m1 m3*

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

definition (**in** *network*) *weak-hb* :: *'msg* ⇒ *'msg* ⇒ *bool* **where**
weak-hb m1 m2 ≡ *hb m1 m2* ∨ *m1* = *m2*

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

lemma (**in** *causal-network*) *causal-broadcast*:
assumes *Deliver m2* ∈ *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 ∃ *i*. *Broadcast m1* ∈ *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 ∃ *i*. *Broadcast m2* ∈ *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* ∈ *set* (*history i*)
shows *Deliver m1* ∈ *set* (*history i*) ∨ *Broadcast m1* ∈ *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*)
apply(*case-tac Deliver m2* ∈ *set* (*history i*))
apply(*subgoal-tac Deliver m1* ∈ *set* (*history i*))
apply *blast*
using *causal-delivery local-order-carrier-closed* **apply** *blast*
apply(*subgoal-tac Broadcast m2* ∈ *set* (*history i*))
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)
  apply(case-tac Deliver m2 ∈ set (history j))
  apply(subgoal-tac Deliver m1 ∈ set (history j))
  apply blast
  using broadcasts-unique hb.intros(3) hb-has-a-reason apply blast
  apply(subgoal-tac Broadcast m2 ∈ set (history j))
  apply blast
  using hb-has-a-reason apply blast
  done

lemma (in causal-network) hb-irrefl:
  assumes hb m1 m2
  shows m1 ≠ m2
  using assms
  apply(induction rule: hb.induct)
  using node-total-order-antisym apply auto[1]
  apply(meson causal-broadcast insert-subset local-order-carrier-closed
        node-total-order-irrefl)
  apply(subgoal-tac ∃ i. Broadcast m3 ∈ set (history i))
  apply(subgoal-tac ∃ j. Broadcast m2 ∈ set (history j))
  apply clarsimp
  apply(subgoal-tac Deliver m2 ∈ set (history j) ∧ Deliver m3 ∈ set (history i))
  apply(meson causal-delivery hb.intros(3) insert-subset local-order-carrier-closed
        network.broadcast-before-delivery network-axioms node-total-order-irrefl)
    apply(meson deliver-locally insert-subset local-order-carrier-closed)
  apply(simp add: hb-broadcast-exists2)+
done

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
  apply(induction rule: hb.induct)
  apply(metis insertI1 local-order-carrier-closed network.broadcasts-unique
        network-axioms subsetCE)
  apply(metis broadcasts-unique insert-subset local-order-carrier-closed
        network.broadcast-before-delivery network-axioms node-total-order-trans)
  apply(case-tac Broadcast m2 ∈ set (history i))
  using node-total-order-trans apply blast
  apply(subgoal-tac Deliver m2 ∈ set (history i))
  apply(subgoal-tac m1 ≠ m2 ∧ m2 ≠ m3)
  apply(metis event.inject(1) hb.intros(1) hb-irrefl network.hb.intros(3) network-axioms
        node-order-is-total hb-irrefl)
  using hb-has-a-reason apply blast+
done

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* (λ*e*. *case e of Deliver m* ⇒ *Some m* | *-* ⇒ *None*) *cs*

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

lemma (**in** *network*) *node-deliver-messages-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*
apply(*clarsimp simp*: *node-deliver-messages-def map-filter-def split*: *event.split-asm*)
using *prefix-to-carriers* **apply** *auto*
done

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*
apply(*induction xs rule*: *rev-induct*)
apply *simp*
apply(*clarsimp simp add*: *node-deliver-messages-append*)
apply *safe*
apply *force*
apply(*clarsimp simp*: *node-deliver-messages-def map-filter-def*)
apply *clarsimp*
apply(*frule prefix-distinct*)
apply *clarsimp*
apply(*subst* (*asm*) *node-deliver-messages-def*) **back back back**
apply(*clarsimp simp add*: *map-filter-def*)
apply(*case-tac x*; *clarsimp*)
apply(*subst* (*asm*) *node-deliver-messages-def*) **back**
apply(*clarsimp simp add*: *map-filter-def*)
apply(*case-tac x*; *clarsimp*)
done

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*
using *assms* **apply** −
apply(*subgoal-tac* $\exists$ *pre suf*. *evts* = *pre* @ (*Deliver last-msg*) # *suf* $\wedge$ *node-deliver-messages suf* = [])
apply(*erule exE*)+
apply(*simp*)
apply(*rule-tac x*=*pre* **in** *exI*)
apply(*rule conjI*)
using *prefix-of-appendD* **apply** *blast*
apply(*subgoal-tac node-deliver-messages* ([*Deliver last-msg*] @ *suf*) = [*last-msg*])
apply(*simp add*: *node-deliver-messages-append*)
apply(*metis append-Nil2 node-deliver-messages-append node-deliver-messages-Deliver*)
apply(*subgoal-tac Deliver last-msg* $\in$ *set evts*)
defer
apply(*simp add*: *prefix-contains-msg*)
apply(*subgoal-tac* $\exists$ *idx*. *idx* < *length evts* $\wedge$ *evts* ! *idx* = *Deliver last-msg*)
apply(*erule exE*)
apply(*subgoal-tac* $\exists$ *pre suf*. *evts* = *pre* @ (*evts* ! *idx*) # *suf*)
defer

using *list-nth-split-technical id-take-nth-drop* **apply** *blast*
apply(*simp add*: *set-elem-nth*)
apply(*erule exE*)+
apply(*rule-tac x*=*pre* **in** *exI*, *rule-tac x*=*suf* **in** *exI*)
apply(*rule conjI*, *simp*, *simp*)
apply(*subgoal-tac node-deliver-messages* (*pre* @ *Deliver last-msg* # *suf*) =
(*node-deliver-messages pre*) @ (*node-deliver-messages* (*Deliver last-msg* # *suf*)))
apply(*subgoal-tac node-deliver-messages* ([*Deliver last-msg*] @ *suf*) = [*last-msg*] @ [])
apply(*metis node-deliver-messages-Deliver node-deliver-messages-append self-append-conv*)
apply(*auto simp add*: *node-deliver-messages-append*)
apply(*subgoal-tac node-deliver-messages* ([*Deliver last-msg*] @ *suf*) = [*last-msg*] @ [])
apply(*simp add*: *node-deliver-messages-append*)
apply(*metis append-Cons node-deliver-messages-Deliver node-deliver-messages-append*
node-deliver-messages-distinct not-Cons-self2 pre-suf-eq-distinct-list self-append-conv2)
done

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 ⋀*m n*. *m* < *length cs* ⟹ *n* < *m* ⟹ *cs* ! *n* ⊏^{i} *cs* ! *m*
shows *hb.hb-consistent* (*node-deliver-messages cs*)
using *assms*
apply −
apply(*induction cs rule*: *rev-induct*)
apply(*unfold node-deliver-messages-def*)
apply(*simp add*: *hb.hb-consistent.intros*(*1*) *map-filter-simps*(*2*))
apply(*case-tac x*; *clarify*)
apply(*simp add*: *List.map-filter-def*)
apply(*subgoal-tac* (⋀*m n*. *m* < *length xs* ⟹ *n* < *m* ⟹ *xs* ! *n* ⊏^{i} *xs* ! *m*))
apply *clarsimp*
apply(*erule-tac x*=*m* **in** *meta-allE*, *erule-tac x*=*n* **in** *meta-allE*, *clarsimp simp add*: *nth-append*)
apply(*subst map-filter-append*)
apply(*clarsimp simp add*: *map-filter-def*)
apply(*rule hb.hb-consistent.intros*)
apply(*subgoal-tac* (⋀*m n*. *m* < *length xs* ⟹ *n* < *m* ⟹ *xs* ! *n* ⊏^{i} *xs* ! *m*))
apply *clarsimp*
apply(*erule-tac x*=*m* **in** *meta-allE*, *erule-tac x*=*n* **in** *meta-allE*, *clarsimp simp add*: *nth-append*)
apply *clarsimp*
apply(*case-tac x*; *clarsimp*)
apply(*drule set-elem-nth*, *erule exE*, *erule conjE*)
apply(*erule-tac x*=*length xs* **in** *meta-allE*, *erule-tac x*=*m* **in** *meta-allE*)
apply *clarsimp*
apply(*subst* (*asm*) *nth-append*, *simp*)
apply(*meson causal-network.causal-delivery causal-network-axioms insert-subset node-histories.local-order-carrier-close
node-histories-axioms node-total-order-irrefl node-total-order-trans*)
done

corollary (**in** *network-with-ops*)
shows *hb.hb-consistent* (*node-deliver-messages* (*history i*))
apply(*subgoal-tac history i* = [] ∨ (∃ *c*. *history i* = [*c*]) ∨ (*length* (*history i*) ≥ *2*))
apply(*erule disjE*, *clarsimp simp add*: *node-deliver-messages-def map-filter-def*)
apply(*erule disjE*, *clarsimp simp add*: *node-deliver-messages-def map-filter-def*)
apply *blast*
apply(*cases history i*; *clarsimp*; *case-tac list*; *clarsimp*)
apply(*rule hb-consistent-technical*[**where** *i*=*i*])
apply(*subst history-order-def*, *clarsimp*)
apply(*metis list-nth-split One-nat-def Suc-le-mono cancel-comm-monoid-add-class.diff-cancel
le-imp-less-Suc length-Cons less-Suc-eq-le less-imp-diff-less neq0-conv nth-Cons-pos*)
apply(*cases history i*; *clarsimp*; *case-tac list*; *clarsimp*)
done

lemma (**in** *network-with-ops*) *hb-consistent-prefix*:
assumes *xs prefix of i*
shows *hb.hb-consistent* (*node-deliver-messages xs*)
using *assms*
apply(*clarsimp simp*: *prefix-of-node-history-def*)
apply(*rule-tac i*=*i* **in** *hb-consistent-technical*)
apply(*subst history-order-def*)
apply(*subgoal-tac xs* = [] ∨ (∃ *c*. *xs* = [*c*]) ∨ (*length* (*xs*) > *1*))
apply(*erule disjE*)
apply *clarsimp*
apply(*erule disjE*)

apply *clarsimp*
apply(*drule list-nth-split*)
apply *assumption*
apply *clarsimp*
apply *clarsimp*
apply(*rule-tac x=xsa* **in** *exI*)
apply(*rule-tac x=ysa* **in** *exI*)
apply(*rule-tac x=zs@ys* **in** *exI*)
apply(*metis Cons-eq-appendI append-assoc*)
apply *force*
done

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-operations pre* = *Some state* $\wedge$ *valid-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*
apply(*subgoal-tac* $\exists$ *pre*. *pre* @ [*Broadcast m*] *prefix of i*)
using *broadcast-only-valid-msgs* **apply** *blast*
using *events-before-exist* **apply** *blast*
done

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** −
apply(*drule delivery-has-a-cause*)
apply(*erule exE*)
apply(*subgoal-tac* $\exists$ *pre*. *pre* @ [*Broadcast m*] *prefix of j*)
using *broadcast-only-valid-msgs* **apply** *blast*
using *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*
using *assms* **apply** −
apply(*subgoal-tac Deliver m* $\in$ *set* (*history i*))
apply(*drule delivery-has-a-cause*)
apply(*erule exE*)
apply(*rule broadcast-is-valid*, *assumption*)
apply(*simp add*: *prefix-elem-to-carriers*)
done

4.4 Dummy network models

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

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

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

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

interpretation *trivial-network-with-constrained-ops*: *network-with-constrained-ops* $\lambda m.\ []$ $(\lambda x\ y.\ Some\ y)$ *0* $\lambda x\ y.\ True$
by *standard* (*simp add*: *trivial-node-histories.prefix-of-node-history-def*)

end

5 Replicated Growable Array

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

theory
Ordered-List
imports
Util
begin

type-synonym $('id, 'v)$ *elt* $= 'id \times 'v \times bool$

5.1 Insert and delete operations

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

hide-const *insert*

```
fun insert-body :: ('id::{linorder}, 'v) elt list ⇒ ('id, 'v) elt ⇒ ('id, 'v) elt list where
  insert-body []     e = [e] |
  insert-body (x#xs) e =
    (if fst x < fst e then
       e#x#xs
     else x#insert-body xs e)

fun insert :: ('id::{linorder}, 'v) elt list ⇒ ('id, 'v) elt ⇒ 'id option ⇒ ('id, 'v) elt list option where
  insert xs     e None     = Some (insert-body xs e) |
  insert []     e (Some i) = None |
  insert (x#xs) e (Some i) =
    (if fst x = i then
       Some (x#insert-body xs e)
     else
       insert xs e (Some i) ⤜ (λt. Some (x#t)))

fun delete :: ('id::{linorder}, 'v) elt list ⇒ 'id ⇒ ('id, 'v) elt list option where
  delete []                 i = None |
  delete ((i', v, flag)#xs) i =
```

(*if* $i' = i$ *then*
Some $((i', v, True)\#xs)$
else
delete $xs\ i \gg\!= (\lambda t.\ Some\ ((i',v,flag)\#t))$)

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 $fst\ `\ set\ (insert\text{-}body\ xs\ e) = fst\ `\ set\ xs \cup \{fst\ e\}$
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 ≠ fst e2*
shows *insert-body* (*insert-body xs e1*) *e2* = *insert-body* (*insert-body xs e2*) *e1*
using *assms* **by**(*induction xs*, *auto*)

lemma *insert-insert-body*:
assumes *fst e1 ≠ fst e2*
and *i2 ≠ Some* (*fst e1*)
shows *insert* (*insert-body xs e1*) *e2 i2* = *insert xs e2 i2* ⤜ (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 ≠ fst e2*
and *i ≠ fst e2*
and *i2 ≠ Some* (*fst e1*)
shows *insert* [] *e2 i2* ⤜ (λ*ys*. *insert ys e1* (*Some i*)) = *None*
using *assms* **by** (*cases i2*) *clarsimp*+

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

lemma *insert-commutes*:
assumes *fst e1 ≠ fst e2*
i1 = None ∨ i1 ≠ Some (*fst e2*)
i2 = None ∨ i2 ≠ Some (*fst e1*)
shows *insert xs e1 i1* ⤜ (λ*ys*. *insert ys e2 i2*) =
insert xs e2 i2 ⤜ (λ*ys*. *insert ys e1 i1*)
using *assms* **proof**(*induction rule*: *insert.induct*)
fix *xs* **and** *e* :: (′*a*, ′*b*) *elt*
assume *i2 = None ∨ i2 ≠ Some* (*fst e*) **and** *fst e ≠ fst e2*
thus *insert xs e None* ⤜ (λ*ys*. *insert ys e2 i2*) = *insert xs e2 i2* ⤜ (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 ≠ fst e2* **and** *i2 = None ∨ i2 ≠ Some* (*fst e*) **and** *Some i = None ∨ Some i ≠ Some* (*fst e2*)
thus *insert* [] *e* (*Some i*) ⤜ (λ*ys*. *insert ys e2 i2*) = *insert* [] *e2 i2* ⤜ (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 ≠ i* ⟹
fst e ≠ fst e2 ⟹
Some i = None ∨ Some i ≠ Some (*fst e2*) ⟹
i2 = None ∨ i2 ≠ Some (*fst e*) ⟹
insert xs e (*Some i*) ⤜ (λ*ys*. *insert ys e2 i2*) = *insert xs e2 i2* ⤜ (λ*ys*. *insert ys e* (*Some i*)))
and *fst e ≠ fst e2*

```
    and Some i = None ∨ Some i ≠ Some (fst e2)
    and i2 = None ∨ i2 ≠ Some (fst e)
  thus insert (x # xs) e (Some i) ≫= (λys. insert ys e2 i2) = insert (x # xs) e2 i2 ≫= (λys. insert
ys e (Some i))
   apply −
   apply(erule disjE)
     apply clarsimp
     apply clarsimp
     apply(case-tac fst x = i)
       apply clarsimp
       apply(case-tac i2)
         apply clarsimp
         apply(force simp add: insert-body-commutes)
       apply clarsimp
     apply(case-tac fst x = a)
       apply clarsimp
       apply(force simp add: insert-body-commutes)
       apply clarsimp
       apply(force simp add: insert-insert-body-commute)
       apply clarsimp
       apply(case-tac i2)
         apply(force cong: Option.bind-cong simp add: insert-insert-body)
         apply clarsimp
         apply(case-tac a = i)
           apply clarsimp
           apply(metis bind-assoc)
           apply clarsimp
           apply(case-tac fst x = a)
             apply clarsimp
             apply(force cong: Option.bind-cong simp add: insert-insert-body)
             apply clarsimp
             apply(metis bind-assoc)
  done
qed

lemma delete-commutes:
  shows delete xs i1 ≫= (λys. delete ys i2) = delete xs i2 ≫= (λys. delete ys i1)
by(induction xs, auto split: bind-splits if-split-asm)

lemma insert-body-delete-commute:
  assumes i2 ≠ fst e
  shows   delete (insert-body xs e) i2 ≫= (λt. Some (x#t)) =
          delete xs i2 ≫= (λy. 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 ≠ fst e
  shows   insert xs e i1 ≫= (λys. delete ys i2) = delete xs i2 ≫= (λys. 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 ⇒ ('id, 'v) elt ⇒ 'id option ⇀ ('id::{linorder}, 'v) elt list
where
  insert' [] e     None     = Some [e] |
  insert' [] e     (Some i) = None |
  insert' (x#xs) e None     =
    (if fst x < fst e then
```

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

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

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

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

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

lemma insert-body-contains-new-elem:
  shows ∃ p s. xs = p @ s ∧ insert-body xs e = p @ e # s
  apply (induction xs)
  apply force
  apply clarsimp
  apply (rule conjI)
  apply clarsimp
  apply (rule-tac x=[] in exI)
  apply (rule-tac x=(a, aa, b) # p @ s in exI)
  apply clarsimp
  apply clarsimp
  apply (rule-tac x=(a, aa, b) # p in exI)
  apply (rule-tac x=s in exI)
  apply clarsimp
done

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
  apply(induction xs arbitrary: pre ref suf)
  apply force
  apply(clarsimp)
```

apply(*case-tac pre*)
apply(*clarsimp*)
apply(*case-tac suf*)
apply *force*
apply *force*
apply *clarsimp*
done

lemma *insert-position-element-technical*:
assumes $\forall x{\in}set\ as.\ a \neq fst\ x$
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*
apply(*induction as arbitrary*: *cs ds*)
apply *simp*
apply *clarsimp*
done

lemma *split-tuple-list-by-id*:
assumes $(a,b,c) \in set\ xs$
and *distinct* $(map\ fst\ xs)$
shows $\exists\, pre\ suf.\ xs = pre\ @\ (a,b,c)\ \#\ suf \wedge (\forall\, y \in set\ pre.\ fst\ y \neq a)$
using *assms*
apply(*induction xs*)
apply *clarsimp*
apply(*case-tac aa* $= (a,b,c)$)
apply(*rule-tac x*=[] **in** *exI*)
apply(*rule-tac x*=*xs* **in** *exI*)
apply *force*
apply(*subgoal-tac* $\exists\, pre\ suf.\ xs = pre\ @\ (a,\ b,\ c)\ \#\ suf \wedge (\forall\, y{\in}set\ pre.\ fst\ y \neq a)$)
apply(*erule exE*)+
apply(*rule-tac x*=*aa*#*pre* **in** *exI*)
apply(*rule-tac x*=*suf* **in** *exI*)
apply(*rule conjI*)
apply *auto*+
done

lemma *insert-preserves-order*:
assumes $i = None \vee (\exists\, i'.\ i = Some\ i' \wedge i' \in fst\ `\ set\ xs)$
and *distinct* $(map\ fst\ xs)$
shows $\exists\, pre\ suf.\ xs = pre@suf \wedge insert\ xs\ e\ i = Some\ (pre\ @\ e\ \#\ suf)$
using *assms*
apply −
apply(*erule disjE*)
apply *clarsimp*
using *insert-body-contains-new-elem* **apply** *metis*
apply(*erule exE*, *clarsimp*)
apply(*subgoal-tac* $\exists\, as\ bs.\ xs = as@(a,aa,b)\#bs \wedge (\forall\, x \in set\ as.\ fst\ x \neq a)$)
apply *clarsimp*
apply(*subgoal-tac* $\exists\, cs\ ds.\ insert\text{-}body\ bs\ e = cs@e\#ds \wedge cs@ds = bs$)
apply *clarsimp*
apply(*rule-tac x*=*as*@(*a*,*aa*,*b*)#*cs* **in** *exI*)
apply(*rule-tac x*=*ds* **in** *exI*)
apply *clarsimp*
apply(*metis insert-position-element-technical*)
apply(*metis insert-body-contains-new-elem*)
using *split-tuple-list-by-id* **apply** *fastforce*
done

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*

export-code *Insert interpret-opers* **in** *OCaml* **file** *rga.ml*

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*

locale *id-consistent-rga-network* = *rga* +
assumes *ids-consistent*: *hb* (*id1*, *op1*) (*id2*, *op2*) ⟹ *id1* < *id2*

definition *indices* :: ('*id* × ('*id*, '*v*) *operation*) *event list* ⇒ '*id list* **where**
indices xs ≡
List.map-filter (λ*x*. *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*
  apply(*induction es arbitrary*: *xs rule*: *rev-induct*; *clarsimp*)
  apply(*case-tac x*; *clarsimp*)
  apply *blast*
  apply(*case-tac b*; *clarsimp*)
  apply(*auto split*: *bind-splits simp add*: *interp-msg-def*)
  apply(*metis* (*no-types*, *hide-lams*) *Un-insert-right image-eqI insert-iff insert-preserve-indices*
      option.sel prefix-of-appendD prod.sel(*1*) *sup-bot.comm-neutral*)
  apply(*metis Un-insert-right fst-conv insert-iff insert-preserve-indices option.sel*)
  apply(*metis* (*no-types*, *hide-lams*) *Un-insert-right insert-iff insert-preserve-indices*′ *option.sel*
      prefix-of-appendD sup-bot.comm-neutral)
  apply(*metis delete-preserve-indices fst-conv image-eqI prefix-of-appendD*)
  using *delete-preserve-indices* **apply** *blast*
done

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*
  apply(*induction es arbitrary*: *xs rule*: *rev-induct*; *clarsimp*)
  apply(*case-tac x*; *clarsimp*)
  apply(*metis image-eqI prefix-of-appendD prod.sel*(*1*))

apply(*case-tac bb*; *clarsimp*)
defer
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*))
apply(*case-tac aba=a*)
apply *blast*
apply(*subgoal-tac* ∃ *xs'*. *apply-operations xsa* = *Some xs'*)
defer
apply(*meson bind-eq-Some-conv*)
apply(*erule exE*)
apply(*metis* (*no-types*, *lifting*) *someone-inserted-id apply-operations-Deliver element-ids-def image-eqI prefix-of-appendD prod.sel*(*1*) *set-map*)
done

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

lemma (**in** *rga*) *insert-msg-id*:
assumes *Broadcast* (*i*, *Insert e n*) ∈ *set* (*history j*)
shows *fst e* = *i*
apply(*subgoal-tac* ∃ *state*. *valid-rga-msg state* (*i*, *Insert e n*))
defer
using *assms broadcast-is-valid* **apply** *blast*
apply(*erule exE*)
apply(*unfold valid-rga-msg-def*)
apply(*clarsimp*)
apply(*case-tac n*)
apply(*simp*, *simp*)
done

lemma (**in** *rga*) *allowed-insert*:
assumes *Broadcast* (*i*, *Insert e n*) ∈ *set* (*history j*)
shows *n* = *None* ∨ (∃ *i' e' n'*. *n* = *Some* (*fst e'*) ∧ *Deliver* (*i'*, *Insert e' n'*) ⊏^{j} *Broadcast* (*i*, *Insert e n*))
apply(*subgoal-tac* ∃ *pre*. *pre* @ [*Broadcast* (*i*, *Insert e n*)] *prefix of j*)
defer
apply(*simp add*: *assms events-before-exist*)
apply(*erule exE*)
apply(*subgoal-tac* ∃ *state*. *apply-operations pre* = *Some state* ∧ *valid-rga-msg state* (*i*, *Insert e n*))
defer
apply(*simp add*: *broadcast-only-valid-msgs*)
apply(*erule exE*, *erule conjE*)
apply(*unfold valid-rga-msg-def*)
apply(*case-tac n*)
apply *simp*+
apply(*subgoal-tac a* ∈ *element-ids state*)
defer
using *apply-opers-idx-elems* **apply** *blast*
apply(*subgoal-tac* ∃ *i' v' f' n'*. *Deliver* (*i'*, *Insert* (*a*, *v'*, *f'*) *n'*) ∈ *set pre*)
defer
using *deliver-insert-exists* **apply** *auto*[*1*]
using *events-in-local-order* **apply** *blast*
done

lemma (**in** *rga*) *allowed-delete*:
assumes *Broadcast* (*i*, *Delete x*) ∈ *set* (*history j*)
shows ∃ *i' n' v b*. *Deliver* (*i'*, *Insert* (*x*, *v*, *b*) *n'*) ⊏^{j} *Broadcast* (*i*, *Delete x*)
apply(*subgoal-tac* ∃ *pre*. *pre* @ [*Broadcast* (*i*, *Delete x*)] *prefix of j*)
defer
apply(*simp add*: *assms events-before-exist*)
apply(*erule exE*)
apply(*subgoal-tac* ∃ *state*. *apply-operations pre* = *Some state* ∧ *valid-rga-msg state* (*i*, *Delete x*))
defer
apply(*simp add*: *broadcast-only-valid-msgs*)
apply(*erule exE*, *erule conjE*)
apply(*unfold valid-rga-msg-def*)
apply(*subgoal-tac x* ∈ *element-ids state*)
defer
using *apply-opers-idx-elems* **apply** *simp*
apply(*subgoal-tac* ∃ *i' v' f' n'*. *Deliver* (*i'*, *Insert* (*x*, *v'*, *f'*) *n'*) ∈ *set pre*)
defer
using *deliver-insert-exists* **apply** *auto*[*1*]
using *events-in-local-order* **apply** *blast*
done

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 ∃ *i' n' v b*. *Deliver* (*i'*, *Insert* (*x*, *v*, *b*) *n'*) ⊏^{j} *Deliver* (*i*, *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 ∃ *i' n v b*. *Deliver* (*i'*, *Insert* (*m*, *v*, *b*) *n*) ∈ *set 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* = *None* ∨ (∃ *i' n' n'' v b*. *n* = *Some n'* ∧ *Deliver* (*i'*, *Insert* (*n'*, *v*, *b*) *n''*) ⊏^{j} *Deliver* (*i*, *Insert e n*))
using *assms*
apply −
apply(*frule delivery-has-a-cause*)
apply(*erule exE*)
apply(*cases n*; *clarsimp*)
apply(*frule allowed-insert*)
apply *clarsimp*
apply(*frule local-order-carrier-closed*)
apply *clarsimp*
apply(*frule delivery-has-a-cause*) **back**
apply *clarsimp*
apply(*drule causal-broadcast*[*rotated*, **where** *j*=*j*])
apply *auto*
done

lemma (**in** *rga*) *allowed-insert-deliver-in-set*:
assumes (*es*@[*Deliver* (*i*, *Insert e m*)]) *prefix of j*
shows $m = None \vee (\exists i'\ m'\ n\ v\ b.\ m = Some\ m' \wedge Deliver\ (i', Insert\ (m', v, b)\ n) \in set\ es)$
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 $\exists ys.$ *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 $\exists ys.$ *delete s n* = *Some ys*
using *assms*
apply −
apply(*frule allowed-delete-deliver-in-set*)
apply *clarsimp*
apply(*rule delete-no-failure*)
apply(*drule idx-in-elem-inserted*)
apply(*metis apply-opers-idx-elems element-ids-def prefix-of-appendD prod.sel*(*1*) *set-map*)
done

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 *assms*
apply(*subgoal-tac e1* = *e2*)
apply(*metis insert-id-unique*)
apply(*cases e1*, *cases e2*; *clarsimp*)
using *insert-id-unique* **by** *force*

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*
using *assms*
apply −
apply(*subgoal-tac Deliver* (*i1*, *Insert e1 n1*) ∈ *set* (*history i*))
apply(*subgoal-tac Deliver* (*i2*, *Insert e2 n2*) ∈ *set* (*history i*))
apply(*subgoal-tac* $\exists j.$ *Broadcast* (*i1*, *Insert e1 n1*) ∈ *set* (*history j*))
apply(*subgoal-tac* $\exists j.$ *Broadcast* (*i2*, *Insert e2 n2*) ∈ *set* (*history j*))
apply(*erule exE*)+
apply(*rule Insert-equal*, *force*, *force*, *force*)
apply(*simp add*: *delivery-has-a-cause*)
apply(*simp add*: *delivery-has-a-cause*)
apply(*auto simp add*: *node-deliver-messages-def prefix-msg-in-history*)
done

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

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

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

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

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

lemma (in rga) apply-operations-distinct:
  assumes xs prefix of i
    and apply-operations xs = Some ys
  shows distinct (map fst ys)
oops

lemma (in rga) 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(rule ext)
  apply(simp add: kleisli-def interp-msg-def)
  apply(case-tac b; case-tac ba)
  apply clarsimp
  apply(case-tac ab = ad)
  apply(subgoal-tac (ab, ac, bb) = (ad, ae, bc) ∧ x12a = x12)
  apply force
  defer
```

apply(*subgoal-tac Ordered-List.insert x* (*ab*, *ac*, *bb*) *x12* ≫= (λ*x*. *Ordered-List.insert x* (*ad*, *ae*, *bc*) *x12a*) = *Ordered-List.insert x* (*ad*, *ae*, *bc*) *x12a* ≫= (λ*x*. *Ordered-List.insert x* (*ab*, *ac*, *bb*) *x12*))
apply(*metis* (*no-types*, *lifting*) *Option.bind-cong interpret-opers.simps*(*1*))
apply(*rule insert-commutes*)
apply *simp*
prefer *2*
apply(*subst* (*asm*) *hb.concurrent-comm*)
apply(*rule insert-commute-assms*)
prefer *2*
apply *assumption*
apply *clarsimp*
apply(*rule conjI*)
apply(*rule prefix-msg-in-history*, *assumption*, *force*)
apply(*rule prefix-msg-in-history*, *assumption*, *force*)
apply(*rule insert-commute-assms*)
prefer *2*
apply *assumption*
apply *clarsimp*
apply(*rule conjI*)
apply(*rule prefix-msg-in-history*, *assumption*, *force*)
apply(*rule prefix-msg-in-history*, *assumption*, *force*)
apply(*clarsimp simp del*: *delete.simps*)
apply(*subgoal-tac Ordered-List.insert x* (*ab*, *ac*, *bb*) *x12* ≫= (λ*x*. *Ordered-List.delete x x2*) = *delete x x2* ≫= (λ*x*. *Ordered-List.insert x* (*ab*, *ac*, *bb*) *x12*))
apply(*metis* (*no-types*, *lifting*) *Option.bind-cong interpret-opers.simps*)
apply(*rule insert-delete-commute*)
apply(*rule Insert-Delete-concurrent*)
apply *clarsimp*
using *prefix-msg-in-history* **apply** *blast*
apply(*clarsimp*)
apply(*clarsimp simp del*: *delete.simps*)
apply(*subgoal-tac delete x x2* ≫= (λ*x*. *insert x* (*ab*, *ac*, *bb*) *x12*) = *Ordered-List.insert x* (*ab*, *ac*, *bb*) *x12* ≫= (λ*x*. *delete x x2*))
apply(*metis* (*no-types*, *lifting*) *Option.bind-cong interpret-opers.simps*)
apply(*rule insert-delete-commute*[*symmetric*])
apply(*rule Insert-Delete-concurrent*)
using *prefix-msg-in-history* **apply** *blast*
apply(*subst* (*asm*) *hb.concurrent-comm*)
apply *assumption*
apply(*clarsimp simp del*: *delete.simps*)
apply(*subgoal-tac delete x x2* ≫= (λ*x*. *delete x x2a*) = *delete x x2a* ≫= (λ*x*. *delete x x2*))
apply(*metis* (*mono-tags*, *lifting*) *Option.bind-cong interpret-opers.simps*(*2*))
apply(*rule delete-commutes*)
using *same-insert* **apply** *force*
done

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* ≠ *None*
using *assms*
apply(*induction xs rule*: *rev-induct*)
apply *clarsimp*
apply(*case-tac x*; *clarsimp*)
apply *force*

apply(*case-tac b*; *clarsimp*)
apply(*metis bind.bind-lunit interpret-opers.simps*(*1*) *prefix-of-appendD rga.Insert-no-failure rga-axioms interp-msg-def prod.sel*(*2*))
apply(*metis bind.bind-lunit interpret-opers.simps*(*2*) *local.delete-no-failure prefix-of-appendD interp-msg-def prod.sel*(*2*))
done

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*
λ*ops*.∃ *xs i*. *xs prefix of i* ∧ *node-deliver-messages xs* = *ops* []
apply(*standard*; *clarsimp*)
apply(*auto simp add*: *hb-consistent-prefix node-deliver-messages-distinct concurrent-operations-commute apply-operations-def*)
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

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

interpretation *non-trivial-rga-implementation*: *rga* λ*m*. *if m* = *0 then* [*Broadcast* (*0*, *Insert* (*0*, *0*, *False*) *None*), *Deliver* (*0*, *Insert* (*0*, *0*, *False*) *None*)] *else* []
oops

end

6 Implementation of integer numbers by target-language integers

theory *Code-Target-Int*
imports *../GCD*
begin

code-datatype *int-of-integer*

declare [[*code drop*: *integer-of-int*]]

context
includes *integer.lifting*

begin

lemma [*code*]:
integer-of-int (*int-of-integer k*) = *k*
by *transfer rule*

lemma [*code*]:
Int.Pos = *int-of-integer* ∘ *integer-of-num*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma [*code*]:
Int.Neg = *int-of-integer* ∘ *uminus* ∘ *integer-of-num*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma [*code-abbrev*]:
int-of-integer (*numeral k*) = *Int.Pos k*
by *transfer simp*

lemma [*code-abbrev*]:
int-of-integer (− *numeral k*) = *Int.Neg k*
by *transfer simp*

lemma [*code*, *symmetric*, *code-post*]:
0 = *int-of-integer 0*
by *transfer simp*

lemma [*code*, *symmetric*, *code-post*]:
1 = *int-of-integer 1*
by *transfer simp*

lemma [*code-post*]:
int-of-integer (− *1*) = − *1*
by *simp*

lemma [*code*]:
k + *l* = *int-of-integer* (*of-int k* + *of-int l*)
by *transfer simp*

lemma [*code*]:
− *k* = *int-of-integer* (− *of-int k*)
by *transfer simp*

lemma [*code*]:
k − *l* = *int-of-integer* (*of-int k* − *of-int l*)
by *transfer simp*

lemma [*code*]:
Int.dup k = *int-of-integer* (*Code-Numeral.dup* (*of-int k*))
by *transfer simp*

declare [[*code drop*: *Int.sub*]]

lemma [*code*]:
k ∗ *l* = *int-of-integer* (*of-int k* ∗ *of-int l*)
by *simp*

lemma [*code*]:
k div l = *int-of-integer* (*of-int k div of-int l*)

by *simp*

lemma [*code*]:
k mod l = int-of-integer (of-int k mod of-int l)
by *simp*

lemma [*code*]:
divmod m n = map-prod int-of-integer int-of-integer (divmod m n)
unfolding *prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv*
by *transfer simp*

lemma [*code*]:
HOL.equal k l = HOL.equal (of-int k :: integer) (of-int l)
by *transfer (simp add: equal)*

lemma [*code*]:
k ≤ l ⟷ (of-int k :: integer) ≤ of-int l
by *transfer rule*

lemma [*code*]:
k < l ⟷ (of-int k :: integer) < of-int l
by *transfer rule*

declare [[*code drop*: *gcd* :: *int* ⇒ - *lcm* :: *int* ⇒ -]]

lemma *gcd-int-of-integer* [*code*]:
gcd (int-of-integer x) (int-of-integer y) = int-of-integer (gcd x y)
by *transfer rule*

lemma *lcm-int-of-integer* [*code*]:
lcm (int-of-integer x) (int-of-integer y) = int-of-integer (lcm x y)
by *transfer rule*

end

lemma (**in** *ring-1*) *of-int-code-if*:
of-int k = (if k = 0 then 0
else if k < 0 then − of-int (− k)
else let
l = 2 ∗ of-int (k div 2);
j = k mod 2
in if j = 0 then l else l + 1)
proof −
from *div-mult-mod-eq* **have** ∗: *of-int k = of-int (k div 2 ∗ 2 + k mod 2)* **by** *simp*
show *?thesis*
by (*simp add*: *Let-def of-int-add* [*symmetric*]) (*simp add*: ∗ *mult.commute*)
qed

declare *of-int-code-if* [*code*]

lemma [*code*]:
nat = nat-of-integer ∘ of-int
including *integer.lifting* **by** *transfer (simp add: fun-eq-iff)*

code-identifier
code-module *Code-Target-Int* ⇀
(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

7 Avoidance of pattern matching on natural numbers

theory *Code-Abstract-Nat*
imports *Main*
begin

When natural numbers are implemented in another than the conventional inductive *0*/*Suc* representation, it is necessary to avoid all pattern matching on natural numbers altogether. This is accomplished by this theory (up to a certain extent).

7.1 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [*code*, *code-unfold*]:
  case-nat = (λ*f g n*. *if n* = *0 then f else g* (*n* − *1*))
  by (*auto simp add*: *fun-eq-iff dest*!: *gr0-implies-Suc*)

7.2 Preprocessors

The term *Suc n* is no longer a valid pattern. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a code equation) must be eliminated. This can be accomplished – as far as possible – by applying the following transformation rule:

lemma *Suc-if-eq*:
  assumes ⋀*n*. *f* (*Suc n*) ≡ *h n*
  assumes *f 0* ≡ *g*
  shows *f n* ≡ *if n* = *0 then g else h* (*n* − *1*)
  by (*rule eq-reflection*) (*cases n*, *insert assms*, *simp-all*)

The rule above is built into a preprocessor that is plugged into the code generator.

setup ‹
let

val Suc-if-eq = *Thm.incr-indexes 1* @{*thm Suc-if-eq*};

fun remove-suc ctxt thms =
  let
    val vname = *singleton* (*Name.variant-list* (*map fst*
      (*fold* (*Term.add-var-names o Thm.full-prop-of*) *thms* [])) *n*);
    val cv = *Thm.cterm-of ctxt* (*Var* ((*vname*, *0*), *HOLogic.natT*));
    val lhs-of = *snd o Thm.dest-comb o fst o Thm.dest-comb o Thm.cprop-of*;
    val rhs-of = *snd o Thm.dest-comb o Thm.cprop-of*;
    fun find-vars ct = (*case Thm.term-of ct of*
        (*Const* (@{*const-name Suc*}, -) $ *Var* -) => [(*cv*, *snd* (*Thm.dest-comb ct*))]
      | - $ - =>
        let val (*ct1*, *ct2*) = *Thm.dest-comb ct*
        in
          map (*apfst* (*fn ct => Thm.apply ct ct2*)) (*find-vars ct1*) @
          map (*apfst* (*Thm.apply ct1*)) (*find-vars ct2*)
        end
      | - => []);
    val eqs = *maps*
      (*fn thm => map* (*pair thm*) (*find-vars* (*lhs-of thm*))) *thms*;
    fun mk-thms (*thm*, (*ct*, *cv*′)) =

```
    let
      val thm' =
        Thm.implies-elim
         (Conv.fconv-rule (Thm.beta-conversion true)
           (Thm.instantiate'
             [SOME (Thm.ctyp-of-cterm ct)] [SOME (Thm.lambda cv ct),
               SOME (Thm.lambda cv' (rhs-of thm)), NONE, SOME cv']
             Suc-if-eq)) (Thm.forall-intr cv' thm)
    in
      case map-filter (fn thm'' =>
          SOME (thm'', singleton
            (Variable.trade (K (fn [thm'''] => [thm''' RS thm'])))
              (Variable.declare-thm thm'' ctxt)) thm'')
        handle THM - => NONE) thms of
          [] => NONE
        | thmps =>
            let val (thms1, thms2) = split-list thmps
            in SOME (subtract Thm.eq-thm (thm :: thms1) thms @ thms2) end
    end
  in get-first mk-thms eqs end;

fun eqn-suc-base-preproc ctxt thms =
  let
    val dest = fst o Logic.dest-equals o Thm.prop-of;
    val contains-suc = exists-Const (fn (c, -) => c = @{const-name Suc});
  in
    if forall (can dest) thms andalso exists (contains-suc o dest) thms
      then thms |> perhaps-loop (remove-suc ctxt) |> (Option.map o map) Drule.zero-var-indexes
       else NONE
  end;

val eqn-suc-preproc = Code-Preproc.simple-functrans eqn-suc-base-preproc;

in

  Code-Preproc.add-functrans (eqn-Suc, eqn-suc-preproc)

end;
›
```

end

8 Implementation of natural numbers by target-language integers

theory *Code-Target-Nat*
imports *Code-Abstract-Nat*
begin

8.1 Implementation for *nat*

context
includes *natural.lifting integer.lifting*
begin

lift-definition *Nat* :: *integer* ⇒ *nat*
  is *nat*

.

lemma [*code-post*]:
Nat 0 = 0
Nat 1 = 1
Nat (numeral k) = numeral k
by (*transfer*, *simp*)+

lemma [*code-abbrev*]:
integer-of-nat = of-nat
by *transfer rule*

lemma [*code-unfold*]:
Int.nat (int-of-integer k) = nat-of-integer k
by *transfer rule*

lemma [*code abstype*]:
Code-Target-Nat.Nat (integer-of-nat n) = n
by *transfer simp*

lemma [*code abstract*]:
integer-of-nat (nat-of-integer k) = max 0 k
by *transfer auto*

lemma [*code-abbrev*]:
nat-of-integer (numeral k) = nat-of-num k
by *transfer* (*simp add*: *nat-of-num-numeral*)

lemma [*code abstract*]:
integer-of-nat (nat-of-num n) = integer-of-num n
by *transfer* (*simp add*: *nat-of-num-numeral*)

lemma [*code abstract*]:
integer-of-nat 0 = 0
by *transfer simp*

lemma [*code abstract*]:
integer-of-nat 1 = 1
by *transfer simp*

lemma [*code*]:
Suc n = n + 1
by *simp*

lemma [*code abstract*]:
integer-of-nat (m + n) = of-nat m + of-nat n
by *transfer simp*

lemma [*code abstract*]:
integer-of-nat (m − n) = max 0 (of-nat m − of-nat n)
by *transfer simp*

lemma [*code abstract*]:
integer-of-nat (m ∗ n) = of-nat m ∗ of-nat n
by *transfer* (*simp add*: *of-nat-mult*)

lemma [*code abstract*]:
integer-of-nat (m div n) = of-nat m div of-nat n

by *transfer* (*simp add*: *zdiv-int*)

lemma [*code abstract*]:
integer-of-nat (*m mod n*) = *of-nat m mod of-nat n*
by *transfer* (*simp add*: *zmod-int*)

lemma [*code*]:
Divides.divmod-nat m n = (*m div n*, *m mod n*)
by (*fact divmod-nat-div-mod*)

lemma [*code*]:
divmod m n = *map-prod nat-of-integer nat-of-integer* (*divmod m n*)
by (*simp only*: *prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv*)
(*transfer*, *simp-all only*: *nat-div-distrib nat-mod-distrib*
zero-le-numeral nat-numeral)

lemma [*code*]:
HOL.equal m n = *HOL.equal* (*of-nat m* :: *integer*) (*of-nat n*)
by *transfer* (*simp add*: *equal*)

lemma [*code*]:
m ≤ *n* ⟷ (*of-nat m* :: *integer*) ≤ *of-nat n*
by *simp*

lemma [*code*]:
m < *n* ⟷ (*of-nat m* :: *integer*) < *of-nat n*
by *simp*

lemma *num-of-nat-code* [*code*]:
num-of-nat = *num-of-integer* ∘ *of-nat*
by *transfer* (*simp add*: *fun-eq-iff*)

end

lemma (**in** *semiring-1*) *of-nat-code-if*:
of-nat n = (*if n* = *0 then 0*
else let
(*m*, *q*) = *Divides.divmod-nat n 2*;
m' = *2* ∗ *of-nat m*
in if q = *0 then m' else m'* + *1*)
proof −
from *div-mult-mod-eq* **have** ∗: *of-nat n* = *of-nat* (*n div 2* ∗ *2* + *n mod 2*) **by** *simp*
show *?thesis*
by (*simp add*: *Let-def divmod-nat-div-mod of-nat-add* [*symmetric*])
(*simp add*: ∗ *mult.commute of-nat-mult add.commute*)
qed

declare *of-nat-code-if* [*code*]

definition *int-of-nat* :: *nat* ⇒ *int* **where**
[*code-abbrev*]: *int-of-nat* = *of-nat*

lemma [*code*]:
int-of-nat n = *int-of-integer* (*of-nat n*)
by (*simp add*: *int-of-nat-def*)

lemma [*code abstract*]:
integer-of-nat (*nat k*) = *max 0* (*integer-of-int k*)

including *integer.lifting* **by** *transfer auto*

lemma *term-of-nat-code* [*code*]:
— Use *nat-of-integer* in term reconstruction instead of *Code-Target-Nat.Nat* such that reconstructed terms can be fed back to the code generator
term-of-class.term-of n =
Code-Evaluation.App
(*Code-Evaluation.Const* (*STR ''Code-Numeral.nat-of-integer''*)
(*typerep.Typerep* (*STR ''fun''*)
[*typerep.Typerep* (*STR ''Code-Numeral.integer''*) [],
typerep.Typerep (*STR ''Nat.nat''*) []]))
(*term-of-class.term-of* (*integer-of-nat n*))
by (*simp add*: *term-of-anything*)

lemma *nat-of-integer-code-post* [*code-post*]:
nat-of-integer 0 = *0*
nat-of-integer 1 = *1*
nat-of-integer (*numeral k*) = *numeral k*
including *integer.lifting* **by** (*transfer*, *simp*)+

code-identifier
code-module *Code-Target-Nat* ⇀
(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

9 Implementation of natural and integer numbers by target-language integers

theory *Code-Target-Numeral*
imports *Code-Target-Int Code-Target-Nat*
begin

end

10 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
~~/*src*/*HOL*/*Library*/*Code-Target-Numeral*
begin

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

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

export-code *Increment counter-op* **in** *OCaml* **file** *counter.ml*

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

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

lemma (**in** *counter*) *concurrent-operations-commute*:
assumes *xs prefix of i*
shows *hb.concurrent-ops-commute* (*node-deliver-messages xs*)
using *assms*
apply(*clarsimp simp*: *hb.concurrent-ops-commute-def*)
apply(*unfold interp-msg-def*, *simp*)
apply(*case-tac b*; *case-tac ba*)
apply(*auto simp add*: *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*. ∃ *xs i*. *xs prefix of i* ∧ *node-deliver-messages xs* = *ops 0*
apply(*standard*; *clarsimp*)
apply(*auto simp add*: *hb-consistent-prefix drop-last-message node-deliver-messages-distinct concurrent-operations-com*
apply(*metis* (*full-types*) *interp-msg-def counter-op.elims*)
using *drop-last-message* **apply** *blast*
done

end
end

11 Observed-Remove Set

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

theory
ORSet
imports
Network
begin

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

type-synonym ('*id*, '*a*) *state* = '*a* ⇒ '*id set*

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

definition *interpret-op* :: ('*id*, '*a*) *operation* ⇒ ('*id*, '*a*) *state* ⇀ ('*id*, '*a*) *state* (⟨-⟩ [*0*] *1000*) **where**
interpret-op oper state ≡
let before = *state* (*op-elem oper*);
after = *case oper of Add i e* ⇒ *before* ∪ {*i*} | *Rem is e* ⇒ *before* − *is*
in Some (*state* ((*op-elem oper*) := *after*))

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

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

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

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

lemma (**in** *orset*) *apply-operations-never-fails*:
assumes *xs prefix of i*
shows *apply-operations xs* ≠ *None*
using *assms*
apply(*induction xs rule*: *rev-induct*)
apply *clarsimp*
apply(*case-tac x*; *clarsimp*)
apply *force*
apply(*metis interpret-op-def interp-msg-def bind.bind-lunit prefix-of-appendD*)
done

lemma (**in** *orset*) *add-id-valid*:
assumes *xs prefix of j*
and *Deliver* (*i1*, *Add i2 e*) ∈ *set xs*
shows *i1* = *i2*
apply(*subgoal-tac* ∃ *state*. *valid-behaviours state* (*i1*, *Add i2 e*))
apply(*simp add*: *valid-behaviours-def*)
using *assms deliver-in-prefix-is-valid* **apply** *blast*
done

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 $e \neq e' \Longrightarrow$ *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* $\notin$ *set* (*added-ids* [*Deliver* (i, *Add i2 e*)] *e*)
shows *i1* $\neq$ *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* $\subseteq$ *set* (*added-ids es x*)
using *assms*
apply (*induct es arbitrary*: *f rule*: *rev-induct*)
apply *force*
apply (*case-tac xa*)
apply *clarsimp*
apply *force*
apply *clarsimp*
apply (*case-tac b*)
apply(*subgoal-tac xs prefix of j*, *clarsimp split*: *bind-splits*)
apply(*clarsimp simp add*: *interp-msg-def*)
apply(*erule-tac x=xb* **in** *meta-allE*, *clarsimp simp add*: *interpret-op-def*)
apply(*clarsimp split*: *if-split-asm simp add*: *op-elem-def added-id-not-in-set*)
apply *force*
apply *force*
apply *force*
apply(*subgoal-tac xs prefix of j*, *clarsimp split*: *bind-splits*)
apply(*erule-tac x=xb* **in** *meta-allE*, *clarsimp simp add*: *interpret-op-def interp-msg-def*)
apply(*clarsimp split*: *if-split-asm simp add*: *op-elem-def*)
apply *force*
apply *force*
apply *force*
done

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*
apply (*induct xs rule*: *rev-induct*)
apply *clarsimp*
apply (*case-tac x*)
apply(*simp add*: *prefix-of-appendD*)
apply *clarsimp*
apply (*case-tac b*)
apply *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 (*metis added-ids-Deliver-Rem-collapse empty-iff list.set*(*1*) *prefix-of-appendD*)
done

lemma (**in** *orset*) *Broadcast-Deliver-prefix-closed*:
assumes *xs* @ [*Broadcast* (r, *Rem ix e*)] *prefix of j*

```
    and i ∈ ix
  shows Deliver (i, Add i e) ∈ set xs
  using assms
  apply(subgoal-tac ∃ y. apply-operations xs = Some y)
   apply clarsimp
   apply(subgoal-tac ix = y e)
    apply clarsimp
    apply(frule-tac x=e in apply-operations-added-ids)
     apply force
    apply(clarsimp)
  using Deliver-added-ids apply blast
   apply (metis (mono-tags, lifting) broadcast-only-valid-msgs operation.case(2) option.simps(1)
     valid-behaviours-def case-prodD)
  using broadcast-only-valid-msgs apply blast
  done

lemma (in orset) Broadcast-Deliver-prefix-closed2:
  assumes xs prefix of j
    and Broadcast (r, Rem ix e) ∈ set xs
    and i ∈ ix
  shows Deliver (i, Add i e) ∈ set xs
  using assms
  apply(induction xs rule: rev-induct)
   apply clarsimp
  apply(erule meta-impE, force)
  apply clarsimp
  apply(erule disjE)
    defer
   apply force
  apply clarsimp
    using Broadcast-Deliver-prefix-closed apply metis
done

lemma (in orset) concurrent-add-remove-independent-technical:
  assumes i ∈ is
    and xs prefix of j
    and (i, Add i e) ∈ set (node-deliver-messages xs) and (ir, Rem is e) ∈ set (node-deliver-messages
xs)
  shows hb (i, Add i e) (ir, Rem is e)
    using assms
  apply(subgoal-tac ∃ pre k. pre@[Broadcast (ir, Rem is e)] prefix of k)
   apply clarsimp
     apply(frule broadcast-only-valid-msgs, clarsimp simp add: valid-behaviours-def)
     apply(subgoal-tac Deliver (i, Add i e) ∈ set pre)
      apply(rule-tac i=k in hb.intros(2))
    using events-in-local-order apply blast
     apply(insert Broadcast-Deliver-prefix-closed2)
     apply(erule-tac x=pre @ [Broadcast (ir, Rem (state e) e)] in meta-allE)
     apply(erule-tac x=k in meta-allE, erule-tac x=ir in meta-allE, erule-tac x=is in meta-allE)
     apply(erule-tac x=e in meta-allE, erule-tac x=i in meta-allE)
      apply clarsimp
    using delivery-has-a-cause events-before-exist prefix-msg-in-history apply blast
    done

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
  apply(subgoal-tac ∃ pre k. pre@[Broadcast (i, Add i e1)] prefix of k)
```

apply(*subgoal-tac* ∃ *pre k*. *pre*@[*Broadcast* (*i*, *Add i e2*)] *prefix of k*)
apply *clarsimp*
apply(*subgoal-tac Broadcast* (*i*, *Add i e1*) ∈ *set* (*history k*))
apply(*subgoal-tac Broadcast* (*i*, *Add i e2*) ∈ *set* (*history ka*))
apply(*drule msg-id-unique*, *assumption*)
apply(*drule broadcast-only-valid-msgs*)+
apply(*clarsimp simp add*: *valid-behaviours-def*)
apply *force*
using *prefix-of-node-history-def* **apply**(*metis Un-insert-right insert-subset list.simps*(15) *prefix-to-carriers set-append*)
using *prefix-of-node-history-def* **apply**(*metis Un-insert-right insert-subset list.simps*(15) *prefix-to-carriers set-append*)
using *assms*(2) *delivery-has-a-cause events-before-exist* **apply** *blast*
using *assms*(1) *delivery-has-a-cause events-before-exist* **apply** *blast*
done

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*
using *assms*
apply(*subgoal-tac* ∃ *pre k*. *pre*@[*Broadcast* (*ir*, *Rem is e2*)] *prefix of k*)
apply *clarsimp*
apply(*frule broadcast-only-valid-msgs*, *clarsimp simp add*: *valid-behaviours-def*)
apply(*subgoal-tac Deliver* (*i*, *Add i e2*) ∈ *set pre*)
apply(*rule-tac j*=*j* **and** *i*=*i* **in** *Deliver-Add-same-id-same-message*)
using *prefix-msg-in-history* **apply** *blast*
using *causal-broadcast events-in-local-order local-order-prefix-closed prefix-contains-msg prefix-to-carriers* **apply** *blast*
apply(*rule Broadcast-Deliver-prefix-closed*, *assumption*, *assumption*)
using *delivery-has-a-cause events-before-exist prefix-msg-in-history* **apply** *blast*
done

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*)
using *assms*
apply(*clarsimp simp*: *hb.concurrent-ops-commute-def*)
apply(*unfold interp-msg-def*, *simp*)
apply(*case-tac b*; *case-tac ba*)
apply(*simp add*: *add-add-commute hb.concurrent-def*)
apply(*metis add-rem-commute concurrent-add-remove-independent hb.concurrent-def add-id-valid prefix-contains-msg*)
apply(*metis add-rem-commute concurrent-add-remove-independent hb.concurrent-def add-id-valid prefix-contains-msg*)
apply(*simp add*: *rem-rem-commute hb.concurrent-def*)

```
  done

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
  λops.∃ xs i. xs prefix of i ∧ node-deliver-messages xs = ops λx.{}
  apply(standard; clarsimp)
     apply(auto 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.