

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

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

December 14, 2021

Abstract

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

Contents

1	Introduction	2
2	Technical Lemmas	2
2.1	Kleisli arrow composition	3
2.2	Lemmas about sets	3
2.3	Lemmas about list	3
3	Strong Eventual Consistency	5
3.1	Concurrent operations	5
3.2	Happens-before consistency	6
3.3	Apply operations	7
3.4	Concurrent operations commute	8
3.5	Abstract convergence theorem	8
3.6	Convergence and progress	9
4	Axiomatic network models	9
4.1	Node histories	10
4.2	Asynchronous broadcast networks	11
4.3	Causal networks	13
4.4	Dummy network models	16
5	Replicated Growable Array	16
5.1	Insert and delete operations	16
5.2	Well-definedness of insert and delete	17
5.3	Preservation of element indices	17

5.4	Commutativity of concurrent operations	18
5.5	Alternative definition of insert	19
5.6	Network	20
5.7	Strong eventual consistency	24
6	Increment-Decrement Counter	24
7	Observed-Remove Set	25

1 Introduction

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

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

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

2 Technical Lemmas

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

theory

Util

imports

Main

HOL-Library.Monad-Syntax

begin

2.1 Kleisli arrow composition

definition *kleisli* :: ('b ⇒ 'b option) ⇒ ('b ⇒ 'b option) ⇒ ('b ⇒ 'b option) (**infixr** ▷ 65) **where**
 $f \triangleright g \equiv \lambda x. (f\ x \gg (\lambda y. g\ y))$

lemma *kleisli-comm-cong*:

assumes $x \triangleright y = y \triangleright x$
shows $z \triangleright x \triangleright y = z \triangleright y \triangleright x$
⟨*proof*⟩

lemma *kleisli-assoc*:

shows $(z \triangleright x) \triangleright y = z \triangleright (x \triangleright y)$
⟨*proof*⟩

2.2 Lemmas about sets

lemma *distinct-set-notin* [*dest*]:

assumes *distinct* ($x \# xs$)
shows $x \notin \text{set } xs$
⟨*proof*⟩

lemma *set-membership-equality-technicalD* [*dest*]:

assumes $\{x\} \cup (\text{set } xs) = \{y\} \cup (\text{set } ys)$
shows $x = y \vee y \in \text{set } xs$
⟨*proof*⟩

lemma *set-equality-technical*:

assumes $\{x\} \cup (\text{set } xs) = \{y\} \cup (\text{set } ys)$
and $x \notin \text{set } xs$
and $y \notin \text{set } ys$
and $y \in \text{set } xs$
shows $\{x\} \cup (\text{set } xs - \{y\}) = \text{set } ys$
⟨*proof*⟩

lemma *set-elem-nth*:

assumes $x \in \text{set } xs$
shows $\exists m. m < \text{length } xs \wedge xs ! m = x$
⟨*proof*⟩

2.3 Lemmas about list

lemma *list-nil-or-snoc*:

shows $xs = [] \vee (\exists y\ ys. xs = ys@[y])$
⟨*proof*⟩

lemma *suffix-eq-distinct-list*:

assumes *distinct* xs
and $ys@suf1 = xs$
and $ys@suf2 = xs$
shows $suf1 = suf2$
⟨*proof*⟩

lemma *pre-suf-eq-distinct-list*:

assumes *distinct* xs
and $ys \neq []$
and $pre1@ys@suf1 = xs$

and $pre2 @ ys @ suf2 = xs$
shows $pre1 = pre2 \wedge suf1 = suf2$
 <proof>

lemma *list-head-unaaffected*:
assumes $hd (x @ [y, z]) = v$
shows $hd (x @ [y \]) = v$
 <proof>

lemma *list-head-butlast*:
assumes $hd xs = v$
and $length xs > 1$
shows $hd (butlast xs) = v$
 <proof>

lemma *list-head-length-one*:
assumes $hd xs = x$
and $length xs = 1$
shows $xs = [x]$
 <proof>

lemma *list-two-at-end*:
assumes $length xs > 1$
shows $\exists xs' x y. xs = xs' @ [x, y]$
 <proof>

lemma *list-nth-split-technical*:
assumes $m < length cs$
and $cs \neq []$
shows $\exists xs ys. cs = xs @ (cs!m) \# ys$
 <proof>

lemma *list-nth-split*:
assumes $m < length cs$
and $n < m$
and $1 < length cs$
shows $\exists xs ys zs. cs = xs @ (cs!n) \# ys @ (cs!m) \# zs$
 <proof>

lemma *list-split-two-elems*:
assumes *distinct cs*
and $x \in set cs$
and $y \in set cs$
and $x \neq y$
shows $\exists pre mid suf. cs = pre @ x \# mid @ y \# suf \vee cs = pre @ y \# mid @ x \# suf$
 <proof>

lemma *split-list-unique-prefix*:
assumes $x \in set xs$
shows $\exists pre suf. xs = pre @ x \# suf \wedge (\forall y \in set pre. x \neq y)$
 <proof>

lemma *map-filter-append*:
shows $List.map-filter P (xs @ ys) = List.map-filter P xs @ List.map-filter P ys$
 <proof>

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 ⇒ 'a ⇒ bool (**infix** ≤ 50)
and *hb* :: 'a ⇒ 'a ⇒ bool (**infix** < 50) +
fixes *interp* :: 'a ⇒ 'b → 'b (⟨-⟩ [0] 1000)
begin

3.1 Concurrent operations

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

definition *concurrent* :: 'a ⇒ 'a ⇒ bool (**infix** ∥ 50) **where**
 $s1 \parallel s2 \equiv \neg(s1 < s2) \wedge \neg(s2 < s1)$

lemma *concurrentI* [*intro!*]: $\neg(s1 < s2) \implies \neg(s2 < s1) \implies s1 \parallel s2$
 ⟨*proof*⟩

lemma *concurrentD1* [*dest*]: $s1 \parallel s2 \implies \neg(s1 < s2)$
 ⟨*proof*⟩

lemma *concurrentD2* [*dest*]: $s1 \parallel s2 \implies \neg(s2 < s1)$
 ⟨*proof*⟩

lemma *concurrent-refl* [*intro!*, *simp*]: $s \parallel s$
 ⟨*proof*⟩

lemma *concurrent-comm*: $s1 \parallel s2 \longleftrightarrow s2 \parallel s1$
 ⟨*proof*⟩

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

lemma *concurrent-set-empty* [*simp*, *intro!*]:
 $\text{concurrent-set } x \ []$
 ⟨*proof*⟩

lemma *concurrent-set-ConsE* [*elim!*]:

assumes *concurrent-set a (x#xs)*
and *concurrent-set a xs \implies concurrent x a \implies G*
shows *G*
 \langle *proof* \rangle

lemma *concurrent-set-ConsI [intro!]:*
concurrent-set a xs \implies concurrent a x \implies concurrent-set a (x#xs)
 \langle *proof* \rangle

lemma *concurrent-set-appendI [intro!]:*
concurrent-set a xs \implies concurrent-set a ys \implies concurrent-set a (xs@ys)
 \langle *proof* \rangle

lemma *concurrent-set-Cons-Snoc [simp]:*
concurrent-set a (xs@[x]) = concurrent-set a (x#xs)
 \langle *proof* \rangle

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 \implies bool where*
 $[intro!]:$ *hb-consistent [] |*
 $[intro!]:$ \llbracket *hb-consistent xs; $\forall x \in \text{set } xs. \neg y \prec x$ $\rrbracket \implies$ *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 $(x \prec y \vee \text{concurrent } x \ y) = (\neg y \prec x)$
 \langle *proof* \rangle

lemma *consistentI [intro!]:*
assumes *hb-consistent (xs @ ys)*
and $\forall x \in \text{set } (xs @ ys). \neg z \prec x$
shows *hb-consistent (xs @ ys @ [z])*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

lemma *hb-consistent-append-D2 [dest]:*
assumes *hb-consistent (xs @ ys)*
shows *hb-consistent ys*

$\langle proof \rangle$

lemma *hb-consistent-append-elim-ConsD* [elim]:

assumes *hb-consistent* ($y \# ys$)

shows *hb-consistent* *ys*

$\langle proof \rangle$

lemma *hb-consistent-remove1* [intro]:

assumes *hb-consistent* *xs*

shows *hb-consistent* (*remove1* *x* *xs*)

$\langle proof \rangle$

lemma *hb-consistent-singleton* [intro!]:

shows *hb-consistent* [*x*]

$\langle proof \rangle$

lemma *hb-consistent-prefix-suffix-exists*:

assumes *hb-consistent* *ys*

hb-consistent (*xs* @ [*x*])

$\{x\} \cup \text{set } xs = \text{set } ys$

distinct ($x \# xs$)

distinct *ys*

shows $\exists \text{prefix suffix. } ys = \text{prefix} @ x \# \text{suffix} \wedge \text{concurrent-set } x \text{ suffix}$

$\langle proof \rangle$

lemma *hb-consistent-append* [intro!]:

assumes *hb-consistent* *suffix*

hb-consistent *prefix*

$\bigwedge s p. s \in \text{set } \text{suffix} \implies p \in \text{set } \text{prefix} \implies \neg s \prec p$

shows *hb-consistent* (*prefix* @ *suffix*)

$\langle proof \rangle$

lemma *hb-consistent-append-porder*:

assumes *hb-consistent* (*xs* @ *ys*)

$x \in \text{set } xs$

$y \in \text{set } ys$

shows $\neg y \prec x$

$\langle proof \rangle$

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* :: $'a \text{ list} \Rightarrow 'b \rightarrow 'b$ **where**

apply-operations *es* $\equiv \text{foldl } (\triangleright) \text{Some } (\text{map } \text{interp } \text{es})$

lemma *apply-operations-empty* [simp]: *apply-operations* [] *s* = *Some s*

$\langle proof \rangle$

lemma *apply-operations-Snoc* [simp]:

apply-operations (*xs*@[*x*]) = (*apply-operations* *xs*) \triangleright [*x*]

$\langle proof \rangle$

3.4 Concurrent operations commute

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

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

lemma *concurrent-ops-commute-empty* [intro!]: *concurrent-ops-commute* []
 <proof>

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

lemma *concurrent-ops-commute-appendD* [dest]:
assumes *concurrent-ops-commute* (xs@ys)
shows *concurrent-ops-commute* xs
 <proof>

lemma *concurrent-ops-commute-rearrange*:
concurrent-ops-commute (xs@x#ys) = *concurrent-ops-commute* (xs@ys@[x])
 <proof>

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)
 <proof>

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
 <proof>

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
  <proof>
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  $\Rightarrow$  bool
    and initial-state :: 'b
  assumes causality:   op-history xs  $\implies$  hb-consistent xs
  assumes distinctness: op-history xs  $\implies$  distinct xs
  assumes commutativity: op-history xs  $\implies$  concurrent-ops-commute xs
  assumes no-failure:   op-history(xs@[x])  $\implies$  apply-operations xs initial-state = Some state  $\implies$  <x>
state  $\neq$  None
  assumes trunc-history: op-history(xs@[x])  $\implies$  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
  <proof>

```

```

theorem sec-progress:
  assumes op-history xs
  shows apply-operations xs initial-state  $\neq$  None
  <proof>

```

```

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* :: $\text{nat} \Rightarrow \text{'evt list}$
assumes *histories-distinct* [*intro!*, *simp*]: *distinct (history i)*

lemma (**in** *node-histories*) *history-finite*:
shows *finite (set (history i))*
<proof>

definition (**in** *node-histories*) *history-order* :: $\text{'evt} \Rightarrow \text{nat} \Rightarrow \text{'evt} \Rightarrow \text{bool}$ (*-/ □⁻/ - [50,1000,50]50*)
where
 $x \sqsubset^i z \equiv \exists xs\ ys\ zs. xs@x\#ys@z\#zs = \text{history } i$

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

lemma (**in** *node-histories*) *local-order-carrier-closed*:
assumes $e1 \sqsubset^i e2$
shows $\{e1, e2\} \subseteq \text{set (history } i)$
<proof>

lemma (**in** *node-histories*) *node-total-order-irrefl*:
shows $\neg (e \sqsubset^i e)$
<proof>

lemma (**in** *node-histories*) *node-total-order-antisym*:
assumes $e1 \sqsubset^i e2$
and $e2 \sqsubset^i e1$
shows *False*
<proof>

lemma (**in** *node-histories*) *node-order-is-total*:
assumes $e1 \in \text{set (history } i)$
and $e2 \in \text{set (history } i)$
and $e1 \neq e2$
shows $e1 \sqsubset^i e2 \vee e2 \sqsubset^i e1$
<proof>

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

lemma (**in** *node-histories*) *carriers-head-lt*:
assumes $y\#ys = \text{history } i$
shows $\neg (x \sqsubset^i y)$
<proof>

lemma (**in** *node-histories*) *prefix-of-ConsD* [*dest*]:

assumes $x \# xs$ prefix of i
shows $[x]$ prefix of i
 $\langle proof \rangle$

lemma (in *node-histories*) *prefix-of-appendD* [*dest*]:
assumes $xs @ ys$ prefix of i
shows xs prefix of i
 $\langle proof \rangle$

lemma (in *node-histories*) *prefix-distinct*:
assumes xs prefix of i
shows *distinct* xs
 $\langle proof \rangle$

lemma (in *node-histories*) *prefix-to-carriers* [*intro*]:
assumes xs prefix of i
shows $set\ xs \subseteq set\ (history\ i)$
 $\langle proof \rangle$

lemma (in *node-histories*) *prefix-elem-to-carriers*:
assumes xs prefix of i
and $x \in set\ xs$
shows $x \in set\ (history\ i)$
 $\langle proof \rangle$

lemma (in *node-histories*) *local-order-prefix-closed*:
assumes $x \sqsubset^i y$
and xs prefix of i
and $y \in set\ xs$
shows $x \in set\ xs$
 $\langle proof \rangle$

lemma (in *node-histories*) *local-order-prefix-closed-last*:
assumes $x \sqsubset^i y$
and $xs@[y]$ prefix of i
shows $x \in set\ xs$
 $\langle proof \rangle$

lemma (in *node-histories*) *events-before-exist*:
assumes $x \in set\ (history\ i)$
shows $\exists pre.\ pre @ [x]$ prefix of i
 $\langle proof \rangle$

lemma (in *node-histories*) *events-in-local-order*:
assumes $pre @ [e2]$ prefix of i
and $e1 \in set\ pre$
shows $e1 \sqsubset^i e2$
 $\langle proof \rangle$

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* \Rightarrow '*msg event list* +
fixes *msg-id* :: '*msg* \Rightarrow '*msgid*

assumes *delivery-has-a-cause*: $\llbracket \text{Deliver } m \in \text{set } (\text{history } i) \rrbracket \Longrightarrow$
 $\exists j. \text{Broadcast } m \in \text{set } (\text{history } j)$
and *deliver-locally*: $\llbracket \text{Broadcast } m \in \text{set } (\text{history } i) \rrbracket \Longrightarrow$
 $\text{Broadcast } m \sqsubset^i \text{Deliver } m$
and *msg-id-unique*: $\llbracket \text{Broadcast } m1 \in \text{set } (\text{history } i);$
 $\text{Broadcast } m2 \in \text{set } (\text{history } j);$
 $\text{msg-id } m1 = \text{msg-id } m2 \rrbracket \Longrightarrow i = j \wedge m1 = m2$

The axioms can be understood as follows:

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

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

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

lemma (**in** *network*) *broadcast-before-delivery*:

assumes $\text{Deliver } m \in \text{set } (\text{history } i)$
shows $\exists j. \text{Broadcast } m \sqsubset^j \text{Deliver } m$
 $\langle \text{proof} \rangle$

lemma (**in** *network*) *broadcasts-unique*:

assumes $i \neq j$
and $\text{Broadcast } m \in \text{set } (\text{history } i)$
shows $\text{Broadcast } m \notin \text{set } (\text{history } j)$
 $\langle \text{proof} \rangle$

Based on the well-known definition by [8], we say that $m1 \prec m2$ if any of the following is true:

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

inductive (**in** *network*) *hb* :: '*msg* \Rightarrow '*msg* \Rightarrow *bool* **where**

hb-broadcast: $\llbracket \text{Broadcast } m1 \sqsubset^i \text{Broadcast } m2 \rrbracket \Longrightarrow \text{hb } m1 \ m2 \mid$
hb-deliver: $\llbracket \text{Deliver } m1 \sqsubset^i \text{Broadcast } m2 \rrbracket \Longrightarrow \text{hb } m1 \ m2 \mid$
hb-trans: $\llbracket \text{hb } m1 \ m2; \text{hb } m2 \ m3 \rrbracket \Longrightarrow \text{hb } m1 \ m3$

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

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

locale *causal-network* = *network* +

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

lemma (in *causal-network*) *causal-broadcast*:

assumes $Deliver\ m2 \in set\ (history\ j)$

and $Deliver\ m1 \sqsubset^i\ Broadcast\ m2$

shows $Deliver\ m1 \sqsubset^j\ Deliver\ m2$

<proof>

lemma (in *network*) *hb-broadcast-exists1*:

assumes $hb\ m1\ m2$

shows $\exists i. Broadcast\ m1 \in set\ (history\ i)$

<proof>

lemma (in *network*) *hb-broadcast-exists2*:

assumes $hb\ m1\ m2$

shows $\exists i. Broadcast\ m2 \in set\ (history\ i)$

<proof>

4.3 Causal networks

lemma (in *causal-network*) *hb-has-a-reason*:

assumes $hb\ m1\ m2$

and $Broadcast\ m2 \in set\ (history\ i)$

shows $Deliver\ m1 \in set\ (history\ i) \vee Broadcast\ m1 \in set\ (history\ i)$

<proof>

lemma (in *causal-network*) *hb-cross-node-delivery*:

assumes $hb\ m1\ m2$

and $Broadcast\ m1 \in set\ (history\ i)$

and $Broadcast\ m2 \in set\ (history\ j)$

and $i \neq j$

shows $Deliver\ m1 \in set\ (history\ j)$

<proof>

lemma (in *causal-network*) *hb-irrefl*:

assumes $hb\ m1\ m2$

shows $m1 \neq m2$

<proof>

lemma (in *causal-network*) *hb-broadcast-broadcast-order*:

assumes $hb\ m1\ m2$

and $Broadcast\ m1 \in set\ (history\ i)$

and $Broadcast\ m2 \in set\ (history\ i)$

shows $Broadcast\ m1 \sqsubset^i\ Broadcast\ m2$

<proof>

lemma (in *causal-network*) *hb-antisym*:

assumes $hb\ x\ y$

and $hb\ y\ x$

shows *False*

<proof>

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

lemma (in *network*) *node-deliver-messages-empty* [simp]:
shows *node-deliver-messages* [] = []
 <proof>

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

lemma (in *network*) *node-deliver-messages-append*:
shows *node-deliver-messages* (xs@ys) = (*node-deliver-messages* xs)@(*node-deliver-messages* ys)
 <proof>

lemma (in *network*) *node-deliver-messages-Broadcast* [simp]:
shows *node-deliver-messages* [Broadcast m] = []
 <proof>

lemma (in *network*) *node-deliver-messages-Deliver* [simp]:
shows *node-deliver-messages* [Deliver m] = [m]
 <proof>

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)
 <proof>

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
 <proof>

lemma (in *network*) *node-deliver-messages-distinct*:
assumes xs prefix of i
shows distinct (*node-deliver-messages* xs)
 <proof>

lemma (in *network*) *drop-last-message*:
assumes evts prefix of i
and *node-deliver-messages* evts = msgs @ [last-msg]
shows \exists pre. pre prefix of i \wedge *node-deliver-messages* pre = msgs
 <proof>

locale *network-with-ops* = *causal-network history fst*
for *history* :: nat \Rightarrow ('msgid \times 'op) event list +
fixes *interp* :: 'op \Rightarrow 'state \rightarrow 'state
and *initial-state* :: 'state

context *network-with-ops* **begin**

definition *interp-msg* :: 'msgid \times 'op \Rightarrow 'state \rightarrow 'state **where**
interp-msg msg state \equiv *interp* (snd msg) state

sublocale *hb: happens-before weak-hb hb interp-msg*
 <proof>

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
⟨proof⟩

lemma (in *network-with-ops*) *apply-operations-Broadcast* [simp]:
shows *apply-operations* (xs @ [Broadcast m]) = *apply-operations* xs
⟨proof⟩

lemma (in *network-with-ops*) *apply-operations-Deliver* [simp]:
shows *apply-operations* (xs @ [Deliver m]) = (*apply-operations* xs ≫≡ interp-msg m)
⟨proof⟩

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

corollary (in *network-with-ops*)
shows hb.hb-consistent (node-deliver-messages (history i))
⟨proof⟩

lemma (in *network-with-ops*) *hb-consistent-prefix*:
assumes xs prefix of i
shows hb.hb-consistent (node-deliver-messages xs)
⟨proof⟩

locale *network-with-constrained-ops* = *network-with-ops* +
fixes valid-msg :: 'c ⇒ ('a × 'b) ⇒ bool
assumes broadcast-only-valid-msgs: pre @ [Broadcast m] prefix of i ⇒
∃ state. *apply-operations* pre = Some state ∧ valid-msg state m

lemma (in *network-with-constrained-ops*) *broadcast-is-valid*:
assumes Broadcast m ∈ set (history i)
shows ∃ state. valid-msg state m
⟨proof⟩

lemma (in *network-with-constrained-ops*) *deliver-is-valid*:
assumes Deliver m ∈ set (history i)
shows ∃ j pre state. pre @ [Broadcast m] prefix of j ∧ *apply-operations* pre = Some state ∧ valid-msg
state m
⟨proof⟩

lemma (in *network-with-constrained-ops*) *deliver-in-prefix-is-valid*:
assumes xs prefix of i
and Deliver m ∈ set xs
shows ∃ state. valid-msg state m
⟨proof⟩

4.4 Dummy network models

interpretation *trivial-node-histories: node-histories* $\lambda m. []$
 $\langle proof \rangle$

interpretation *trivial-network: network* $\lambda m. []$ *id*
 $\langle proof \rangle$

interpretation *trivial-causal-network: causal-network* $\lambda m. []$ *id*
 $\langle proof \rangle$

interpretation *trivial-network-with-ops: network-with-ops* $\lambda m. []$ $(\lambda x y. \text{Some } y)$ *0*
 $\langle proof \rangle$

interpretation *trivial-network-with-constrained-ops: network-with-constrained-ops* $\lambda m. []$ $(\lambda x y. \text{Some } y)$ *0* $\lambda x y. \text{True}$
 $\langle proof \rangle$

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 $(\text{'id}, \text{'v}) \text{elt} = \text{'id} \times \text{'v} \times \text{bool}$

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* :: $(\text{'id}::\{\text{linorder}\}, \text{'v}) \text{elt list} \Rightarrow (\text{'id}, \text{'v}) \text{elt} \Rightarrow (\text{'id}, \text{'v}) \text{elt list}$ **where**
insert-body [] $e = [e]$ |
insert-body $(x\#\text{xs}) e =$
 (if $\text{fst } x < \text{fst } e$ then
 $e\#x\#\text{xs}$
 else $x\#\text{insert-body } \text{xs } e$)

fun *insert* :: $(\text{'id}::\{\text{linorder}\}, \text{'v}) \text{elt list} \Rightarrow (\text{'id}, \text{'v}) \text{elt} \Rightarrow \text{'id option} \Rightarrow (\text{'id}, \text{'v}) \text{elt list option}$ **where**
insert $\text{xs } e \text{ None} = \text{Some } (\text{insert-body } \text{xs } e)$ |
insert [] $e (\text{Some } i) = \text{None}$ |
insert $(x\#\text{xs}) e (\text{Some } i) =$
 (if $\text{fst } x = i$ then


```

    Some (x#insert-body xs e)
  else
    insert xs e (Some i) ≫ (λt. Some (x#t))

```

```

fun delete :: ('id::{linorder}, 'v) elt list ⇒ 'id ⇒ ('id, 'v) elt list option where
  delete []           i = None |
  delete ((i', v, flag)#xs) i =
    (if i' = i then
      Some ((i', v, True)#xs)
    else
      delete xs i ≫ (λt. Some ((i',v,flag)#t)))

```

5.2 Well-definedness of insert and delete

lemma *insert-no-failure*:

```

assumes i = None ∨ (∃ i'. i = Some i' ∧ i' ∈ fst ' set xs)
shows   ∃ xs'. insert xs e i = Some xs'
⟨proof⟩

```

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

```

assumes insert xs e i = None
shows   i ≠ None
⟨proof⟩

```

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

```

assumes insert xs e (Some i) = None
shows   i ∉ fst ' set xs
⟨proof⟩

```

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

```

assumes i ∉ fst ' set xs
shows   insert xs e (Some i) = None
⟨proof⟩

```

lemma *delete-no-failure*:

```

assumes i ∈ fst ' set xs
shows   ∃ xs'. delete xs i = Some xs'
⟨proof⟩

```

lemma *delete-None-index-not-in* [dest]:

```

assumes delete xs i = None
shows   i ∉ fst ' set xs
⟨proof⟩

```

lemma *index-not-in-delete-None* [simp]:

```

assumes i ∉ fst ' set xs
shows   delete xs i = None
⟨proof⟩

```

5.3 Preservation of element indices

lemma *insert-body-preserve-indices* [simp]:

```

shows   fst ' set (insert-body xs e) = fst ' set xs ∪ {fst e}
⟨proof⟩

```

lemma *insert-preserve-indices*:

```

assumes ∃ ys. insert xs e i = Some ys
shows   fst ' set (the (insert xs e i)) = fst ' set xs ∪ {fst e}

```

<proof>

corollary *insert-preserve-indices'*:

assumes $insert\ xs\ e\ i = Some\ ys$

shows $fst\ 'set\ (the\ (insert\ xs\ e\ i)) = fst\ 'set\ xs\ \cup\ \{fst\ e\}$

<proof>

lemma *delete-preserve-indices*:

assumes $delete\ xs\ i = Some\ ys$

shows $fst\ 'set\ xs = fst\ 'set\ ys$

<proof>

5.4 Commutativity of concurrent operations

lemma *insert-body-commutes*:

assumes $fst\ e1 \neq fst\ e2$

shows $insert\ body\ (insert\ body\ xs\ e1)\ e2 = insert\ body\ (insert\ body\ xs\ e2)\ e1$

<proof>

lemma *insert-insert-body*:

assumes $fst\ e1 \neq fst\ e2$

and $i2 \neq Some\ (fst\ e1)$

shows $insert\ (insert\ body\ xs\ e1)\ e2\ i2 = insert\ xs\ e2\ i2 \ggg (\lambda ys. Some\ (insert\ body\ ys\ e1))$

<proof>

lemma *insert-Nil-None*:

assumes $fst\ e1 \neq fst\ e2$

and $i \neq fst\ e2$

and $i2 \neq Some\ (fst\ e1)$

shows $insert\ []\ e2\ i2 \ggg (\lambda ys. insert\ ys\ e1\ (Some\ i)) = None$

<proof>

lemma *insert-insert-body-commute*:

assumes $i \neq fst\ e1$

and $fst\ e1 \neq fst\ e2$

shows $insert\ (insert\ body\ xs\ e1)\ e2\ (Some\ i) =$

$insert\ xs\ e2\ (Some\ i) \ggg (\lambda y. Some\ (insert\ body\ y\ e1))$

<proof>

lemma *insert-commutes*:

assumes $fst\ e1 \neq fst\ e2$

$i1 = None \vee i1 \neq Some\ (fst\ e2)$

$i2 = None \vee i2 \neq Some\ (fst\ e1)$

shows $insert\ xs\ e1\ i1 \ggg (\lambda ys. insert\ ys\ e2\ i2) =$

$insert\ xs\ e2\ i2 \ggg (\lambda ys. insert\ ys\ e1\ i1)$

<proof>

lemma *delete-commutes*:

shows $delete\ xs\ i1 \ggg (\lambda ys. delete\ ys\ i2) = delete\ xs\ i2 \ggg (\lambda ys. delete\ ys\ i1)$

<proof>

lemma *insert-body-delete-commute*:

assumes $i2 \neq fst\ e$

shows $delete\ (insert\ body\ xs\ e)\ i2 \ggg (\lambda t. Some\ (x\#t)) =$

$delete\ xs\ i2 \ggg (\lambda y. Some\ (x\#\ insert\ body\ y\ e))$

<proof>

lemma *insert-delete-commute*:

assumes $i2 \neq \text{fst } e$
shows $\text{insert } xs \ e \ i1 \gg= (\lambda ys. \text{delete } ys \ i2) = \text{delete } xs \ i2 \gg= (\lambda ys. \text{insert } ys \ e \ i1)$
 $\langle \text{proof} \rangle$

5.5 Alternative definition of insert

fun $\text{insert}' :: ('id::\{\text{linorder}\}, 'v) \text{elt list} \Rightarrow ('id, 'v) \text{elt} \Rightarrow 'id \text{option} \rightarrow ('id::\{\text{linorder}\}, 'v) \text{elt list}$

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 $[\text{elim!}, \text{dest}]$:

assumes $\text{insert}' \ xs \ e \ \text{None} = \text{None}$

shows False

$\langle \text{proof} \rangle$

lemma $\text{insert-body-insert}'$:

shows $\text{insert}' \ xs \ e \ \text{None} = \text{Some} \ (\text{insert-body } xs \ e)$

$\langle \text{proof} \rangle$

lemma $\text{insert-insert}'$:

shows $\text{insert } xs \ e \ i = \text{insert}' \ xs \ e \ i$

$\langle \text{proof} \rangle$

lemma $\text{insert-body-stop-iteration}$:

assumes $\text{fst } e > \text{fst } x$

shows $\text{insert-body } (x\#xs) \ e = e\#x\#xs$

$\langle \text{proof} \rangle$

lemma $\text{insert-body-contains-new-elem}$:

shows $\exists p \ s. \ xs = p @ s \wedge \text{insert-body } xs \ e = p @ e \# s$

$\langle \text{proof} \rangle$

lemma $\text{insert-between-elements}$:

assumes $xs = \text{pre}@ref\#suf$

and $\text{distinct } (\text{map } \text{fst } xs)$

and $\bigwedge i'. i' \in \text{fst } \text{'set } xs \implies i' < \text{fst } e$

shows $\text{insert } xs \ e \ (\text{Some } (\text{fst } ref)) = \text{Some } (\text{pre} @ ref \# e \# suf)$

$\langle \text{proof} \rangle$

lemma $\text{insert-position-element-technical}$:

assumes $\forall x \in \text{set } as. a \neq \text{fst } x$

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)*
 ⟨*proof*⟩

lemma *split-tuple-list-by-id*:

assumes (*a, b, c*) ∈ *set xs*

and *distinct* (*map fst xs*)

shows ∃ *pre suf*. *xs* = *pre @ (a, b, c) # suf* ∧ (∀ *y* ∈ *set pre*. *fst y* ≠ *a*)

⟨*proof*⟩

lemma *insert-preserves-order*:

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

and *distinct* (*map fst xs*)

shows ∃ *pre suf*. *xs* = *pre @ suf* ∧ *insert xs e i* = *Some (pre @ e # suf)*

⟨*proof*⟩

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-ops* :: (*'id::linorder, 'v*) *operation* ⇒ (*'id, 'v*) *elt list* → (*'id, 'v*) *elt list* (⟨-⟩ [0] 1000)

where

interpret-ops (*Insert e n*) *xs* = *insert xs e n* |

interpret-ops (*Delete n*) *xs* = *delete xs n*

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

element-ids list ≡ *set (map fst list)*

definition *valid-rga-msg* :: (*'id, 'v*) *elt list* ⇒ *'id* × (*'id::linorder, 'v*) *operation* ⇒ *bool* **where**

valid-rga-msg list msg ≡ *case msg of*

(*i, Insert e None*) ⇒ *fst e* = *i* |

(*i, Insert e (Some pos)*) ⇒ *fst e* = *i* ∧ *pos* ∈ *element-ids list* |

(*i, Delete pos*) ⇒ *pos* ∈ *element-ids list*

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

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* [] = []

⟨*proof*⟩

lemma *indices-append* [*simp*]:

shows *indices (xs@ys)* = *indices xs @ indices ys*

⟨*proof*⟩

lemma *indices-Broadcast-singleton* [simp]:

shows *indices* [Broadcast *b*] = []

⟨*proof*⟩

lemma *indices-Deliver-Insert* [simp]:

shows *indices* [Deliver (*i*, Insert *e n*)] = [fst *e*]

⟨*proof*⟩

lemma *indices-Deliver-Delete* [simp]:

shows *indices* [Deliver (*i*, Delete *n*)] = []

⟨*proof*⟩

lemma (in *rga*) *idx-in-elem-inserted* [intro]:

assumes Deliver (*i*, Insert *e n*) ∈ *set xs*

shows fst *e* ∈ *set (indices xs)*

⟨*proof*⟩

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)*

⟨*proof*⟩

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*⟩

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*

⟨*proof*⟩

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*

⟨*proof*⟩

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*

⟨*proof*⟩

lemma (in *rga*) *insert-msg-id*:

assumes Broadcast (*i*, Insert *e n*) ∈ *set (history j)*

shows fst *e* = *i*

⟨*proof*⟩

lemma (in *rga*) *allowed-insert*:

assumes $Broadcast\ (i,\ Insert\ e\ n) \in set\ (history\ j)$

shows $n = None \vee (\exists i'\ e'\ n'.\ n = Some\ (fst\ e') \wedge Deliver\ (i',\ Insert\ e'\ n') \sqsubset^j Broadcast\ (i,\ Insert\ e\ n))$

<proof>

lemma (in *rga*) *allowed-delete*:

assumes $Broadcast\ (i,\ Delete\ x) \in set\ (history\ j)$

shows $\exists i'\ n'\ v\ b.\ Deliver\ (i',\ Insert\ (x,\ v,\ b)\ n') \sqsubset^j Broadcast\ (i,\ Delete\ x)$

<proof>

lemma (in *rga*) *insert-id-unique*:

assumes $fst\ e1 = fst\ e2$

and $Broadcast\ (i1,\ Insert\ e1\ n1) \in set\ (history\ i)$

and $Broadcast\ (i2,\ Insert\ e2\ n2) \in set\ (history\ j)$

shows $Insert\ e1\ n1 = Insert\ e2\ n2$

<proof>

lemma (in *rga*) *allowed-delete-deliver*:

assumes $Deliver\ (i,\ Delete\ x) \in set\ (history\ j)$

shows $\exists i'\ n'\ v\ b.\ Deliver\ (i',\ Insert\ (x,\ v,\ b)\ n') \sqsubset^j Deliver\ (i,\ Delete\ x)$

<proof>

lemma (in *rga*) *allowed-delete-deliver-in-set*:

assumes $(es@[Deliver\ (i,\ Delete\ m)])\ prefix\ of\ j$

shows $\exists i'\ n'\ v\ b.\ Deliver\ (i',\ Insert\ (m,\ v,\ b)\ n) \in set\ es$

<proof>

lemma (in *rga*) *allowed-insert-deliver*:

assumes $Deliver\ (i,\ Insert\ e\ n) \in set\ (history\ j)$

shows $n = None \vee (\exists i'\ n'\ n''\ v\ b.\ n = Some\ n' \wedge Deliver\ (i',\ Insert\ (n',\ v,\ b)\ n'') \sqsubset^j Deliver\ (i,\ Insert\ e\ n))$

<proof>

lemma (in *rga*) *allowed-insert-deliver-in-set*:

assumes $(es@[Deliver\ (i,\ Insert\ e\ m)])\ prefix\ of\ j$

shows $m = None \vee (\exists i'\ m'\ n'\ v\ b.\ m = Some\ m' \wedge Deliver\ (i',\ Insert\ (m',\ v,\ b)\ n) \in set\ es)$

<proof>

lemma (in *rga*) *Insert-no-failure*:

assumes $es\ @\ [Deliver\ (i,\ Insert\ e\ n)]\ prefix\ of\ j$

and *apply-operations* $es = Some\ s$

shows $\exists ys.\ insert\ s\ e\ n = Some\ ys$

<proof>

lemma (in *rga*) *delete-no-failure*:

assumes $es\ @\ [Deliver\ (i,\ Delete\ n)]\ prefix\ of\ j$

and *apply-operations* $es = Some\ s$

shows $\exists ys.\ delete\ s\ n = Some\ ys$

<proof>

lemma (in *rga*) *Insert-equal*:

assumes $fst\ e1 = fst\ e2$

and $Broadcast\ (i1,\ Insert\ e1\ n1) \in set\ (history\ i)$

and $Broadcast\ (i2,\ Insert\ e2\ n2) \in set\ (history\ j)$

shows $Insert\ e1\ n1 = Insert\ e2\ n2$

<proof>

lemma (in *rga*) *same-insert*:

assumes $\text{fst } e1 = \text{fst } e2$
and xs prefix of i
and $(i1, \text{Insert } e1 \ n1) \in \text{set } (\text{node-deliver-messages } xs)$
and $(i2, \text{Insert } e2 \ n2) \in \text{set } (\text{node-deliver-messages } xs)$
shows $\text{Insert } e1 \ n1 = \text{Insert } e2 \ n2$

<proof>

lemma (in *rga*) *insert-commute-assms*:

assumes $\{\text{Deliver } (i, \text{Insert } e \ n), \text{Deliver } (i', \text{Insert } e' \ n')\} \subseteq \text{set } (\text{history } j)$
and $\text{hb.concurrent } (i, \text{Insert } e \ n) \ (i', \text{Insert } e' \ n')$
shows $n = \text{None} \vee n \neq \text{Some } (\text{fst } e')$

<proof>

lemma *subset-reorder*:

assumes $\{a, b\} \subseteq c$
shows $\{b, a\} \subseteq c$

<proof>

lemma (in *rga*) *Insert-Insert-concurrent*:

assumes $\{\text{Deliver } (i, \text{Insert } e \ k), \text{Deliver } (i', \text{Insert } e' \ (\text{Some } m))\} \subseteq \text{set } (\text{history } j)$
and $\text{hb.concurrent } (i, \text{Insert } e \ k) \ (i', \text{Insert } e' \ (\text{Some } m))$
shows $\text{fst } e \neq m$

<proof>

lemma (in *rga*) *insert-valid-assms*:

assumes $\text{Deliver } (i, \text{Insert } e \ n) \in \text{set } (\text{history } j)$
shows $n = \text{None} \vee n \neq \text{Some } (\text{fst } e)$

<proof>

lemma (in *rga*) *Insert-Delete-concurrent*:

assumes $\{\text{Deliver } (i, \text{Insert } e \ n), \text{Deliver } (i', \text{Delete } n')\} \subseteq \text{set } (\text{history } j)$
and $\text{hb.concurrent } (i, \text{Insert } e \ n) \ (i', \text{Delete } n')$
shows $n' \neq \text{fst } e$

<proof>

lemma (in *rga*) *concurrent-operations-commute*:

assumes xs prefix of i
shows $\text{hb.concurrent-ops-commute } (\text{node-deliver-messages } xs)$

<proof>

corollary (in *rga*) *concurrent-operations-commute'*:

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

<proof>

lemma (in *rga*) *apply-operations-never-fails*:

assumes xs prefix of i
shows $\text{apply-operations } xs \neq \text{None}$

<proof>

lemma (in *rga*) *apply-operations-never-fails'*:

shows $\text{apply-operations } (\text{history } i) \neq \text{None}$

<proof>

corollary (in *rga*) *rga-convergence*:

assumes $\text{set } (\text{node-deliver-messages } xs) = \text{set } (\text{node-deliver-messages } ys)$
and xs prefix of i

and ys prefix of j
shows $apply\text{-}operations\ xs = apply\text{-}operations\ ys$
 ⟨*proof*⟩

5.7 Strong eventual consistency

context *rga* **begin**

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

end

interpretation *trivial-rga-implementation*: *rga* $\lambda x.$ []
 ⟨*proof*⟩

end

6 Increment-Decrement Counter

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

theory

Counter

imports

Network

begin

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

fun *counter-op* :: *operation* \Rightarrow *int* \rightarrow *int* **where**
 $counter\text{-}op\ Increment\ x = Some\ (x + 1)$ |
 $counter\text{-}op\ Decrement\ x = Some\ (x - 1)$

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

lemma (**in** *counter*) $counter\text{-}op\ x \triangleright counter\text{-}op\ y = counter\text{-}op\ y \triangleright counter\text{-}op\ x$
 ⟨*proof*⟩

lemma (**in** *counter*) *concurrent-operations-commute*:

assumes xs prefix of i

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

⟨*proof*⟩

corollary (**in** *counter*) *counter-convergence*:

assumes $set\ (\text{node-deliver-messages } xs) = set\ (\text{node-deliver-messages } ys)$

and xs prefix of i

and ys prefix of j

shows $apply\text{-}operations\ xs = apply\text{-}operations\ ys$

⟨*proof*⟩

context *counter* **begin**

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
 $\lambda ops. \exists xs\ i. xs\ \text{prefix of } i \wedge \text{node-deliver-messages } xs = ops\ 0$
 ⟨*proof*⟩

end
end

7 Observed-Remove Set

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

theory

ORSet

imports

Network

begin

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

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

definition op-elem :: ('id, 'a) operation \Rightarrow 'a **where**

op-elem oper \equiv case oper of Add i e \Rightarrow e | Rem is e \Rightarrow e

definition interpret-op :: ('id, 'a) operation \Rightarrow ('id, 'a) state \rightarrow ('id, 'a) state ($\langle - \rangle$ [0] 1000) **where**

interpret-op oper state \equiv

let before = state (op-elem oper);

after = case oper of Add i e \Rightarrow before \cup {i} | Rem is e \Rightarrow before - is

in Some (state ((op-elem oper) := after))

definition valid-behaviours :: ('id, 'a) state \Rightarrow 'id \times ('id, 'a) operation \Rightarrow bool **where**

valid-behaviours state msg \equiv

case msg of

(i, Add j e) \Rightarrow i = j |

(i, Rem is e) \Rightarrow is = state e

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

lemma (in orset) add-add-commute:

shows \langle Add i1 e1 $\rangle \triangleright \langle$ Add i2 e2 $\rangle = \langle$ Add i2 e2 $\rangle \triangleright \langle$ Add i1 e1 \rangle

\langle proof \rangle

lemma (in orset) add-rem-commute:

assumes $i \notin is$

shows \langle Add i e1 $\rangle \triangleright \langle$ Rem is e2 $\rangle = \langle$ Rem is e2 $\rangle \triangleright \langle$ Add i e1 \rangle

\langle proof \rangle

lemma (in orset) apply-operations-never-fails:

assumes xs prefix of i

shows apply-operations xs \neq None

\langle proof \rangle

lemma (in orset) add-id-valid:

assumes xs prefix of j

and Deliver (i1, Add i2 e) \in set xs

shows i1 = i2

\langle proof \rangle

definition (in orset) added-ids :: ('id \times ('id, 'b) operation) event list \Rightarrow 'b \Rightarrow 'id list **where**

added-ids es p \equiv List.map-filter (λx . case x of Deliver (i, Add j e) \Rightarrow if e = p then Some j else None

| - \Rightarrow None) es

lemma (in orset) [simp]:
shows added-ids [] e = []
<proof>

lemma (in orset) [simp]:
shows added-ids (xs @ ys) e = added-ids xs e @ added-ids ys e
<proof>

lemma (in orset) added-ids-Broadcast-collapse [simp]:
shows added-ids ([Broadcast e]) e' = []
<proof>

lemma (in orset) added-ids-Deliver-Rem-collapse [simp]:
shows added-ids ([Deliver (i, Rem is e)]) e' = []
<proof>

lemma (in orset) added-ids-Deliver-Add-diff-collapse [simp]:
shows $e \neq e' \implies$ added-ids ([Deliver (i, Add j e)]) e' = []
<proof>

lemma (in orset) added-ids-Deliver-Add-same-collapse [simp]:
shows added-ids ([Deliver (i, Add j e)]) e = [j]
<proof>

lemma (in orset) added-id-not-in-set:
assumes $i1 \notin \text{set (added-ids [Deliver (i, Add i2 e)])}$ e)
shows $i1 \neq i2$
<proof>

lemma (in orset) apply-operations-added-ids:
assumes es prefix of j
and apply-operations es = Some f
shows $f x \subseteq \text{set (added-ids es x)}$
<proof>

lemma (in orset) Deliver-added-ids:
assumes xs prefix of j
and $i \in \text{set (added-ids xs e)}$
shows $\text{Deliver (i, Add i e)} \in \text{set xs}$
<proof>

lemma (in orset) Broadcast-Deliver-prefix-closed:
assumes $xs @ [\text{Broadcast (r, Rem ix e)}]$ prefix of j
and $i \in ix$
shows $\text{Deliver (i, Add i e)} \in \text{set xs}$
<proof>

lemma (in orset) Broadcast-Deliver-prefix-closed2:
assumes xs prefix of j
and $\text{Broadcast (r, Rem ix e)} \in \text{set xs}$
and $i \in ix$
shows $\text{Deliver (i, Add i e)} \in \text{set xs}$
<proof>

lemma (in orset) concurrent-add-remove-independent-technical:
assumes $i \in is$

and xs prefix of j
and $(i, \text{Add } i \ e) \in \text{set } (\text{node-deliver-messages } xs)$ **and** $(ir, \text{Rem } is \ e) \in \text{set } (\text{node-deliver-messages } xs)$
shows $hb \ (i, \text{Add } i \ e) \ (ir, \text{Rem } is \ e)$
 $\langle \text{proof} \rangle$

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

lemma (**in orset**) *ids-imply-messages-same*:
assumes $i \in is$
and xs prefix of j
and $(i, \text{Add } i \ e1) \in \text{set } (\text{node-deliver-messages } xs)$ **and** $(ir, \text{Rem } is \ e2) \in \text{set } (\text{node-deliver-messages } xs)$
shows $e1 = e2$
 $\langle \text{proof} \rangle$

corollary (**in orset**) *concurrent-add-remove-independent*:
assumes $\neg hb \ (i, \text{Add } i \ e1) \ (ir, \text{Rem } is \ e2)$ **and** $\neg hb \ (ir, \text{Rem } is \ e2) \ (i, \text{Add } i \ e1)$
and xs prefix of j
and $(i, \text{Add } i \ e1) \in \text{set } (\text{node-deliver-messages } xs)$ **and** $(ir, \text{Rem } is \ e2) \in \text{set } (\text{node-deliver-messages } xs)$
shows $i \notin is$
 $\langle \text{proof} \rangle$

lemma (**in orset**) *rem-rem-commute*:
shows $\langle \text{Rem } i1 \ e1 \rangle \triangleright \langle \text{Rem } i2 \ e2 \rangle = \langle \text{Rem } i2 \ e2 \rangle \triangleright \langle \text{Rem } i1 \ e1 \rangle$
 $\langle \text{proof} \rangle$

lemma (**in orset**) *concurrent-operations-commute*:
assumes xs prefix of i
shows $hb.\text{concurrent-ops-commute } (\text{node-deliver-messages } xs)$
 $\langle \text{proof} \rangle$

theorem (**in orset**) *convergence*:
assumes $\text{set } (\text{node-deliver-messages } xs) = \text{set } (\text{node-deliver-messages } ys)$
and xs prefix of i **and** ys prefix of j
shows $\text{apply-operations } xs = \text{apply-operations } ys$
 $\langle \text{proof} \rangle$

context orset begin

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

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.