A formal proof of the Chandy–Lamport distributed snapshot algorithm

Ben Fiedler[1] and Dmitriy Traytel[1]

[1]ETH Zürich

December 14, 2021

Abstract

We provide a suitable distributed system model and implementation the Chandy–Lamport distributed snapshot algorithm [1]. Our main result is a formal termination and correctness proof of the Chandy–Lamport algorithm and its use in stable property detection.

Contents

1 Modelling distributed systems **2**
1.1 The distributed system locale 4
1.1.1 State transitions 4

2 Traces **14**
2.1 Properties of traces 15
2.2 Describing intermediate configurations 16
2.3 Trace-related lemmas 17

3 Utilties **19**

4 Swap lemmas **21**

5 The Chandy–Lamport algorithm **31**
5.1 The computation locale 31
5.2 Termination 33
5.3 Correctness 34
5.3.1 Pre- and postrecording events 35
5.3.2 Event swapping 36
5.3.3 Relating configurations and the computed snapshot . 37
5.3.4 Relating process states 41
5.3.5 Relating channel states 41

5.4 Obtaining the desired traces 45
5.5 Stable property detection 49

6 Extension to infinite traces **50**

7 Example **54**

1 Modelling distributed systems

We assume familiarity with Chandy and Lamport's paper *Distributed Snapshots: Determining Global States of Distributed Systems* [1].

theory *Distributed-System*

imports *Main*

begin

type-synonym *'a fifo* = *'a list*
type-synonym *channel-id* = *nat*

datatype *'m message* =
  Marker
| *Msg 'm*

datatype *recording-state* =
  NotStarted
| *Recording*
| *Done*

We characterize distributed systems by three underlying type variables: Type variable 'p captures the processes of the underlying system. Type variable 's describes the possible states of the processes. Finally, type variable 'm describes all possible messages in said system.

Each process is in exactly one state at any point in time of the system. Processes are interconnected by directed channels, which hold messages in-flight between connected processes. There can be an arbitrary number of channels between different processes. The entire state of the system including the (potentially unfinished) snapshot state is called *configuration*.

record *('p, 's, 'm) configuration* =
  states :: *'p* ⇒ *'s*
  msgs :: *channel-id* ⇒ *'m message fifo*

  process-snapshot :: *'p* ⇒ *'s option*
  channel-snapshot :: *channel-id* ⇒ *'m fifo* ∗ *recording-state*

An event in Chandy and Lamport's formalization describes a process' state transition, optionally producing or consuming (but not both) a message on

a channel. Additionally, a process may either initiate a snapshot spontaneously, or is forced to do so by receiving a snapshot *marker* on one of it's incoming channels.

datatype $('p, 's, 'm)$ *event* =
  isTrans: *Trans* (*occurs-on*: $'p$) $'s$ $'s$
| *isSend*: *Send* (*getId*: *channel-id*)
  (*occurs-on*: $'p$)
  (*partner*: $'p$)
  $'s$ $'s$ (*getMsg*: $'m$)
| *isRecv*: *Recv* (*getId*: *channel-id*)
  (*occurs-on*: $'p$)
  (*partner*: $'p$)
  $'s$ $'s$ (*getMsg*: $'m$)

| *isSnapshot*: *Snapshot* (*occurs-on*: $'p$)
| *isRecvMarker*: *RecvMarker* (*getId*: *channel-id*)
  (*occurs-on*: $'p$)
  (*partner*: $'p$)

We introduce abbreviations and type synoyms for commonly used terms.

type-synonym $('p, 's, 'm)$ *trace* = $('p, 's, 'm)$ *event list*

abbreviation *ps* **where** *ps* ≡ *process-snapshot*
abbreviation *cs* **where** *cs* ≡ *channel-snapshot*

abbreviation *no-snapshot-change* **where**
no-snapshot-change c c' ≡ $((\forall p'.\ ps\ c\ p' = ps\ c'\ p') \wedge (\forall i'.\ cs\ c\ i' = cs\ c'\ i'))$

abbreviation *has-snapshotted* **where**
has-snapshotted c p ≡ *process-snapshot c p* ≠ *None*

A regular event is an event as described in Chandy and Lamport's original paper: A state transition accompanied by the emission or receiving of a message. Nonregular events are related to snapshotting and receiving markers along communication channels.

definition *regular-event*[*simp*]:
regular-event ev ≡ (*isTrans ev* ∨ *isSend ev* ∨ *isRecv ev*)

lemma *nonregular-event*:
~ *regular-event ev* = (*isSnapshot ev* ∨ *isRecvMarker ev*)
⟨*proof*⟩

lemma *event-occurs-on-unique*:
assumes
p ≠ *q*
occurs-on ev = *p*
shows
occurs-on ev ≠ *q*

⟨*proof*⟩

1.1 The distributed system locale

In order to capture Chandy and Lamport's computation system we introduce two locales. The distributed system locale describes global truths, such as the mapping from channel IDs to sender and receiver processes, the transition relations for the underlying computation system and the core assumption that no process has a channel to itself. While not explicitly mentioned in Chandy's and Lamport's work, it makes sense to assume that a channel need not communicate to itself via messages, since it shares memory with itself.

locale *distributed-system* =
  fixes
    channel :: *channel-id* ⇒ (*'p* ∗ *'p*) *option* **and**
    trans :: *'p* ⇒ *'s* ⇒ *'s* ⇒ *bool* **and**
    send :: *channel-id* ⇒ *'p* ⇒ *'p* ⇒ *'s* ⇒ *'s* ⇒ *'m* ⇒ *bool* **and**
    recv :: *channel-id* ⇒ *'p* ⇒ *'p* ⇒ *'s* ⇒ *'s* ⇒ *'m* ⇒ *bool*
  assumes
    no-self-channel:
      ∀ *i*. ∄ *p*. *channel i* = *Some* (*p*, *p*)
begin

1.1.1 State transitions

definition *can-occur* :: (*'p*, *'s*, *'m*) *event* ⇒ (*'p*, *'s*, *'m*) *configuration* ⇒ *bool* **where**
can-occur ev c ≡ (*case ev of*
    Trans p s s' ⇒ *states c p* = *s*
                  ∧ *trans p s s'*
  | *Send i p q s s' msg* ⇒ *states c p* = *s*
                  ∧ *channel i* = *Some* (*p*, *q*)
                  ∧ *send i p q s s' msg*
  | *Recv i p q s s' msg* ⇒ *states c p* = *s*
                  ∧ *channel i* = *Some* (*q*, *p*)
                  ∧ *length* (*msgs c i*) > *0*
                  ∧ *hd* (*msgs c i*) = *Msg msg*
                  ∧ *recv i p q s s' msg*
  | *Snapshot p* ⇒ ¬ *has-snapshotted c p*
  | *RecvMarker i p q* ⇒ *channel i* = *Some* (*q*, *p*)
                  ∧ *length* (*msgs c i*) > *0*
                  ∧ *hd* (*msgs c i*) = *Marker*)

definition *src* **where**
  src i p ≡ (∃ *q*. *channel i* = *Some* (*p*, *q*))

definition *dest* **where**
  dest i q ≡ (∃ *p*. *channel i* = *Some* (*p*, *q*))

lemma *can-occur-Recv*:
assumes
can-occur (*Recv i p q s s′ m*) *c*
shows
states c p = *s* ∧ *channel i* = *Some* (*q*, *p*) ∧ (∃ *xs*. *msgs c i* = *Msg m* # *xs*) ∧
recv i p q s s′ m
⟨*proof*⟩

abbreviation *check-snapshot-occur* **where**
check-snapshot-occur c c′ p ≡
(*can-occur* (*Snapshot p*) *c* ∧
(*ps c′ p* = *Some* (*states c p*))
∧ (∀ *p′*. *states c p′* = *states c′ p′*)
∧ (∀ *p′*. (*p′* ≠ *p*) ⟶ *ps c′ p′* = *ps c p′*)
∧ (∀ *i*. (∃ *q*. *channel i* = *Some* (*p*, *q*)) ⟶ *msgs c′ i* = *msgs c i* @ [*Marker*])
∧ (∀ *i*. (∃ *q*. *channel i* = *Some* (*q*, *p*)) ⟶ *channel-snapshot c′ i* = (*fst* (*channel-snapshot c i*), *Recording*))
∧ (∀ *i*. (∄ *q*. *channel i* = *Some* (*p*, *q*)) ⟶ *msgs c′ i* = *msgs c i*)
∧ (∀ *i*. (∄ *q*. *channel i* = *Some* (*q*, *p*)) ⟶ *channel-snapshot c′ i* = *channel-snapshot c i*))

abbreviation *check-recv-marker-occur* **where**
check-recv-marker-occur c c′ i p q ≡
(*can-occur* (*RecvMarker i p q*) *c*
∧ (∀ *r*. *states c r* = *states c′ r*)
∧ (∀ *r*. (*r* ≠ *p*) ⟶ *process-snapshot c r* = *process-snapshot c′ r*)
∧ (*Marker* # *msgs c′ i* = *msgs c i*)
∧ (*channel-snapshot c′ i* = (*fst* (*channel-snapshot c i*), *Done*))
∧ (*if has-snapshotted c p*
then (*process-snapshot c p* = *process-snapshot c′ p*)
∧ (∀ *i′*. (*i′* ≠ *i*) ⟶ *msgs c′ i′* = *msgs c i′*)
∧ (∀ *i′*. (*i′* ≠ *i*) ⟶ *channel-snapshot c i′* = *channel-snapshot c′ i′*)
else (*process-snapshot c′ p* = *Some* (*states c p*))
∧ (∀ *i′*. *i′* ≠ *i* ∧ (∃ *r*. *channel i′* = *Some* (*p*, *r*))
⟶ *msgs c′ i′* = *msgs c i′* @ [*Marker*])
∧ (∀ *i′*. *i′* ≠ *i* ∧ (∃ *r*. *channel i′* = *Some* (*r*, *p*))
⟶ *channel-snapshot c′ i′* = (*fst* (*channel-snapshot c i′*), *Recording*))
∧ (∀ *i′*. *i′* ≠ *i* ∧ (∄ *r*. *channel i′* = *Some* (*p*, *r*))
⟶ *msgs c′ i′* = *msgs c i′*)
∧ (∀ *i′*. *i′* ≠ *i* ∧ (∄ *r*. *channel i′* = *Some* (*r*, *p*))
⟶ *channel-snapshot c′ i′* = *channel-snapshot c i′*)))

abbreviation *check-trans-occur* **where**
check-trans-occur c c′ p s s′≡
(*can-occur* (*Trans p s s′*) *c*
∧ (*states c′ p* = *s′*)
∧ (∀ *r*. (*r* ≠ *p*) ⟶ *states c′ r* = *states c r*)
∧ (∀ *i*. *msgs c′ i* = *msgs c i*)
∧ (*no-snapshot-change c c′*))

abbreviation *check-send-occur* **where**
check-send-occur c c′ i p q s s′ msg ≡
(*can-occur* (*Send i p q s s′ msg*) *c*
∧ (*states c′ p* = *s′*)
∧ (∀ *r*. (*r* ≠ *p*) ⟶ *states c′ r* = *states c r*)
∧ (*msgs c′ i* = *msgs c i* @ [*Msg msg*])
∧ (∀ *i′*. *i* ≠ *i′* ⟶ *msgs c′ i′* = *msgs c i′*)
∧ (*no-snapshot-change c c′*))

abbreviation *check-recv-occur* **where**
check-recv-occur c c′ i p q s s′ msg ≡
(*can-occur* (*Recv i p q s s′ msg*) *c*
∧ (*states c p* = *s* ∧ *states c′ p* = *s′*)
∧ (∀ *r*. (*r* ≠ *p*) ⟶ *states c′ r* = *states c r*)
∧ (*msgs c i* = *Msg msg* # *msgs c′ i*)
∧ (∀ *i′*. *i* ≠ *i′* ⟶ *msgs c′ i′* = *msgs c i′*)
∧ (∀ *r*. *process-snapshot c r* = *process-snapshot c′ r*)
∧ (∀ *i′*. *i′* ≠ *i* ⟶ *channel-snapshot c i′* = *channel-snapshot c′ i′*)
∧ (*if snd* (*channel-snapshot c i*) = *Recording*
then channel-snapshot c′ i = (*fst* (*channel-snapshot c i*) @ [*msg*], *Recording*)
else channel-snapshot c i = *channel-snapshot c′ i*))

The *next* predicate lets us express configuration transitions using events. The predicate $next(s_1, e, s_2)$ denotes the transition of the configuration s_1 to s_2 via the event e. It ensures that e can occur in state s_1 and the state s_2 is correctly constructed from s_1.

primrec *next* ::
('*p*, '*s*, '*m*) *configuration*
⇒ ('*p*, '*s*, '*m*) *event*
⇒ ('*p*, '*s*, '*m*) *configuration*
⇒ *bool*
(- ⊢ - ↦ - [*70*, *70*, *70*]) **where**
next-snapshot: *c* ⊢ *Snapshot p* ↦ *c′* =
check-snapshot-occur c c′ p
| *next-recv-marker*: *c* ⊢ *RecvMarker i p q* ↦ *c′* =
check-recv-marker-occur c c′ i p q
| *next-trans*: *c* ⊢ *Trans p s s′* ↦ *c′* =
check-trans-occur c c′ p s s′
| *next-send*: *c* ⊢ *Send i p q s s′ msg* ↦ *c′* =
check-send-occur c c′ i p q s s′ msg
| *next-recv*: *c* ⊢ *Recv i p q s s′ msg* ↦ *c′* =
check-recv-occur c c′ i p q s s′ msg

Useful lemmas about state transitions

lemma *state-and-event-determine-next*:
assumes
c ⊢ *ev* ↦ *c′* **and**
c ⊢ *ev* ↦ *c″*

shows
$c' = c''$
⟨*proof*⟩

lemma *exists-next-if-can-occur*:
assumes
can-occur ev c
shows
$\exists\, c'.\ c \vdash ev \mapsto c'$
⟨*proof*⟩

lemma *exists-exactly-one-following-state*:
can-occur ev c $\Longrightarrow \exists!c'.\ c \vdash ev \mapsto c'$
⟨*proof*⟩

lemma *no-state-change-if-no-event*:
assumes
$c \vdash ev \mapsto c'$ **and**
occurs-on ev $\neq p$
shows
states c p = states c' p $\wedge$ *process-snapshot c p = process-snapshot c' p*
⟨*proof*⟩

lemma *no-msgs-change-if-no-channel*:
assumes
$c \vdash ev \mapsto c'$ **and**
channel i = None
shows
msgs c i = msgs c' i
⟨*proof*⟩

lemma *no-cs-change-if-no-channel*:
assumes
$c \vdash ev \mapsto c'$ **and**
channel i = None
shows
cs c i = cs c' i
⟨*proof*⟩

lemma *no-msg-change-if-no-event*:
assumes
$c \vdash ev \mapsto c'$ **and**
isSend ev $\longrightarrow$ *getId ev* $\neq i$ **and**
isRecv ev $\longrightarrow$ *getId ev* $\neq i$ **and**
regular-event ev
shows
msgs c i = msgs c' i
⟨*proof*⟩

lemma *no-cs-change-if-no-event*:
assumes
$c \vdash ev \mapsto c'$ **and**
isRecv ev $\longrightarrow$ *getId ev* $\neq i$ **and**
regular-event ev
shows
cs c i = *cs c' i*
⟨*proof*⟩

lemma *happen-implies-can-occur*:
assumes
$c \vdash ev \mapsto c'$
shows
can-occur ev c
⟨*proof*⟩

lemma *snapshot-increases-message-length*:
assumes
ev = *Snapshot p* **and**
$c \vdash ev \mapsto c'$ **and**
channel i = *Some* (*q*, *r*)
shows
length (*msgs c i*) $\leq$ *length* (*msgs c' i*)
⟨*proof*⟩

lemma *recv-marker-changes-head-only-at-i*:
assumes
ev = *RecvMarker i p q* **and**
$c \vdash ev \mapsto c'$ **and**
$i' \neq i$
shows
msgs c i' = [] $\vee$ *hd* (*msgs c i'*) = *hd* (*msgs c' i'*)
⟨*proof*⟩

lemma *recv-marker-other-channels-not-shrinking*:
assumes
ev = *RecvMarker i p q* **and**
$c \vdash ev \mapsto c'$
shows
length (*msgs c i'*) $\leq$ *length* (*msgs c' i'*) $\longleftrightarrow$ $i \neq i'$
⟨*proof*⟩

lemma *regular-event-cannot-induce-snapshot*:
assumes
$\sim$ *has-snapshotted c p* **and**
$c \vdash ev \mapsto c'$
shows
regular-event ev $\longrightarrow$ $\sim$ *has-snapshotted c' p*
⟨*proof*⟩

lemma *regular-event-preserves-process-snapshots*:
assumes
$c \vdash ev \mapsto c'$
shows
regular-event $ev \Longrightarrow ps\ c\ r = ps\ c'\ r$
⟨*proof*⟩

lemma *no-state-change-if-nonregular-event*:
assumes
$\sim$ *regular-event* ev **and**
$c \vdash ev \mapsto c'$
shows
$states\ c\ p = states\ c'\ p$
⟨*proof*⟩

lemma *nonregular-event-induces-snapshot*:
assumes
$\sim$ *has-snapshotted* $c\ p$ **and**
$c \vdash ev \mapsto c'$ **and**
occurs-on $ev = p$ **and**
$\sim$ *regular-event* ev
shows
$\sim$ *regular-event* $ev \longrightarrow$ *has-snapshotted* $c'\ p$
⟨*proof*⟩

lemma *snapshot-state-unchanged*:
assumes
step: $c \vdash ev \mapsto c'$ **and**
has-snapshotted $c\ p$
shows
$ps\ c\ p = ps\ c'\ p$
⟨*proof*⟩

lemma *message-must-be-delivered*:
assumes
valid: $c \vdash ev \mapsto c'$ **and**
delivered: $(msgs\ c\ i \neq [] \wedge hd\ (msgs\ c\ i) = m) \wedge (msgs\ c'\ i = [] \vee hd\ (msgs\ c'\ i) \neq m)$
shows
$(\exists p\ q.\quad ev = RecvMarker\ i\ p\ q \quad \wedge m = Marker)$
$\vee\ (\exists p\ q\ s\ s'\ m'.\ ev = Recv\ i\ p\ q\ s\ s'\ m' \wedge m = Msg\ m')$
⟨*proof*⟩

lemma *message-must-be-delivered-2*:
assumes
$c \vdash ev \mapsto c'$
$m : set\ (msgs\ c\ i)$
$m \notin set\ (msgs\ c'\ i)$

shows
$(\exists\, p\ q.\ ev = RecvMarker\ i\ p\ q \wedge m = Marker) \vee (\exists\, p\ q\ s\ s'\ m'.\ ev = Recv\ i\ p\ q\ s\ s'\ m' \wedge m = Msg\ m')$
⟨*proof*⟩

lemma *recv-marker-means-snapshotted-1*:
assumes
$ev = RecvMarker\ i\ p\ q$ **and**
$c \vdash ev \mapsto c'$
shows
has-snapshotted $c'\ p$
⟨*proof*⟩

lemma *recv-marker-means-snapshotted-2*:
fixes
$c\ c' :: ('p, 's, 'm)\ configuration$ **and**
$ev :: ('p, 's, 'm)\ event$ **and**
$i :: channel\text{-}id$
assumes
$c \vdash ev \mapsto c'$ **and**
$Marker : set\ (msgs\ c\ i)$ **and**
$Marker \notin set\ (msgs\ c'\ i)$ **and**
$channel\ i = Some\ (q, p)$
shows
has-snapshotted $c'\ p$
⟨*proof*⟩

lemma *event-stays-valid-if-no-occurrence*:
assumes
$c \vdash ev \mapsto c'$ **and**
$occurs\text{-}on\ ev \neq occurs\text{-}on\ ev'$ **and**
$can\text{-}occur\ ev'\ c$
shows
$can\text{-}occur\ ev'\ c'$
⟨*proof*⟩

lemma *msgs-unchanged-for-other-is*:
assumes
$c \vdash ev \mapsto c'$ **and**
regular-event ev **and**
$getId\ ev = i$ **and**
$i' \neq i$
shows
$msgs\ c\ i' = msgs\ c'\ i'$
⟨*proof*⟩

lemma *msgs-unchanged-if-snapshotted-RecvMarker-for-other-is*:
assumes
$c \vdash ev \mapsto c'$ **and**

ev = *RecvMarker i p q* **and**
has-snapshotted c p **and**
$i' \neq i$
shows
msgs c i' = *msgs c' i'*
⟨*proof*⟩

lemma *event-can-go-back-if-no-sender*:
assumes
$c \vdash ev \mapsto c'$ **and**
occurs-on ev ≠ *occurs-on ev'* **and**
can-occur ev' c' **and**
~ *isRecvMarker ev'* **and**
~ *isSend ev*
shows
can-occur ev' c
⟨*proof*⟩

lemma *nonregular-event-can-go-back-if-in-distinct-processes*:
assumes
$c \vdash ev \mapsto c'$ **and**
regular-event ev **and**
~ *regular-event ev'* **and**
can-occur ev' c' **and**
occurs-on ev ≠ *occurs-on ev'*
shows
can-occur ev' c
⟨*proof*⟩

lemma *same-state-implies-same-result-state*:
assumes
states c p = *states d p*
$c \vdash ev \mapsto c'$ **and**
$d \vdash ev \mapsto d'$
shows
states d' p = *states c' p*
⟨*proof*⟩

lemma *same-snapshot-state-implies-same-result-snapshot-state*:
assumes
ps c p = *ps d p* **and**
states c p = *states d p* **and**
$c \vdash ev \mapsto c'$ **and**
$d \vdash ev \mapsto d'$
shows
ps d' p = *ps c' p*
⟨*proof*⟩

lemma *same-messages-imply-same-resulting-messages*:

assumes
msgs c i = msgs d i
c ⊢ *ev* ↦ *c′* **and**
d ⊢ *ev* ↦ *d′* **and**
regular-event ev
shows
msgs c′ i = msgs d′ i
⟨*proof*⟩

lemma *Trans-msg*:
assumes
c ⊢ *ev* ↦ *c′* **and**
isTrans ev
shows
msgs c i = msgs c′ i
⟨*proof*⟩

lemma *new-msg-in-set-implies-occurrence*:
assumes
c ⊢ *ev* ↦ *c′* **and**
m ∉ *set* (*msgs c i*) **and**
m ∈ *set* (*msgs c′ i*) **and**
channel i = Some (*p*, *q*)
shows
occurs-on ev = p (**is** *?P*)
⟨*proof*⟩

lemma *new-Marker-in-set-implies-nonregular-occurence*:
assumes
c ⊢ *ev* ↦ *c′* **and**
Marker ∉ *set* (*msgs c i*) **and**
Marker ∈ *set* (*msgs c′ i*) **and**
channel i = Some (*p*, *q*)
shows
~ *regular-event ev* (**is** *?P*)
⟨*proof*⟩

lemma *RecvMarker-implies-Marker-in-set*:
assumes
c ⊢ *ev* ↦ *c′* **and**
ev = RecvMarker cid p q
shows
Marker ∈ *set* (*msgs c cid*)
⟨*proof*⟩

lemma *RecvMarker-given-channel*:
assumes
isRecvMarker ev **and**
getId ev = cid **and**

channel cid = *Some* (*p*, *q*) **and**
can-occur ev c
shows
ev = *RecvMarker cid q p*
⟨*proof*⟩

lemma *Recv-given-channel*:
assumes
isRecv ev **and**
getId ev = *cid* **and**
channel cid = *Some* (*p*, *q*) **and**
can-occur ev c
shows
∃ *s s′ m*. *ev* = *Recv cid q p s s′ m*
⟨*proof*⟩

lemma *same-cs-if-not-recv*:
assumes
c ⊢ *ev* ↦ *c′* **and**
~ *isRecv ev*
shows
fst (*cs c cid*) = *fst* (*cs c′ cid*)
⟨*proof*⟩

lemma *done-only-from-recv-marker*:
assumes
c ⊢ *ev* ↦ *c′* **and**
channel cid = *Some* (*p*, *q*) **and**
snd (*cs c cid*) ≠ *Done* **and**
snd (*cs c′ cid*) = *Done*
shows
ev = *RecvMarker cid q p*
⟨*proof*⟩

lemma *cs-not-not-started-stable*:
assumes
c ⊢ *ev* ↦ *c′* **and**
snd (*cs c cid*) ≠ *NotStarted* **and**
channel cid = *Some* (*p*, *q*)
shows
snd (*cs c′ cid*) ≠ *NotStarted*
⟨*proof*⟩

lemma *fst-cs-changed-by-recv-recording*:
assumes
step: *c* ⊢ *ev* ↦ *c′* **and**
fst (*cs c cid*) ≠ *fst* (*cs c′ cid*) **and**
channel cid = *Some* (*p*, *q*)
shows

snd (*cs c cid*) = *Recording* ∧ (∃ *p q u u′ m*. *ev* = *Recv cid q p u u′ m*)
⟨*proof*⟩

lemma *no-marker-and-snapshotted-implies-no-more-markers*:
assumes
c ⊢ *ev* ↦ *c′* **and**
has-snapshotted c p **and**
Marker ∉ *set* (*msgs c cid*) **and**
channel cid = *Some* (*p*, *q*)
shows
Marker ∉ *set* (*msgs c′ cid*)
⟨*proof*⟩

lemma *same-messages-if-no-occurrence*:
assumes
c ⊢ *ev* ↦ *c′* **and**
~ *occurs-on ev* = *p* **and**
~ *occurs-on ev* = *q* **and**
channel cid = *Some* (*p*, *q*)
shows
msgs c cid = *msgs c′ cid* ∧ *cs c cid* = *cs c′ cid*
⟨*proof*⟩

end

end

2 Traces

Traces extend transitions to finitely many intermediate events.

theory *Trace*
imports
HOL−Library.Sublist
Distributed-System

begin

context *distributed-system*

begin

We can think of a trace as the transitive closure of the next relation. A trace consists of initial and final configurations c and c', with an ordered list of events t occurring sequentially on c, yielding c'.

inductive (**in** *distributed-system*) *trace* **where**
tr-init: *trace c* [] *c*
| *tr-step*: ⟦ *c* ⊢ *ev* ↦ *c′*; *trace c′ t c″* ⟧
⟹ *trace c* (*ev* # *t*) *c″*

2.1 Properties of traces

lemma *trace-trans*:
shows
$[\![\mathit{trace}\ c\ t\ c';\ \mathit{trace}\ c'\ t'\ c'']\!] \Longrightarrow \mathit{trace}\ c\ (t\ @\ t')\ c''$
⟨*proof*⟩

lemma *trace-decomp-head*:
assumes
$\mathit{trace}\ c\ (ev\ \#\ t)\ c'$
shows
$\exists\, c''.\ c \vdash ev \mapsto c'' \wedge \mathit{trace}\ c''\ t\ c'$
⟨*proof*⟩

lemma *trace-decomp-tail*:
shows
$\mathit{trace}\ c\ (t\ @\ [ev])\ c' \Longrightarrow \exists\, c''.\ \mathit{trace}\ c\ t\ c'' \wedge c'' \vdash ev \mapsto c'$
⟨*proof*⟩

lemma *trace-snoc*:
assumes
$\mathit{trace}\ c\ t\ c'$ **and**
$c' \vdash ev \mapsto c''$
shows
$\mathit{trace}\ c\ (t\ @\ [ev])\ c''$
⟨*proof*⟩

lemma *trace-rev-induct* [*consumes 1*, *case-names tr-rev-init tr-rev-step*]:
$[\![\mathit{trace}\ c\ t\ c';\ (\bigwedge c.\ P\ c\ []\ c);\ (\bigwedge c\ t\ c'\ ev\ c''.\ \mathit{trace}\ c\ t\ c' \Longrightarrow P\ c\ t\ c' \Longrightarrow c' \vdash ev \mapsto c'' \Longrightarrow P\ c\ (t\ @\ [ev])\ c'')]\!] \Longrightarrow P\ c\ t\ c'$
⟨*proof*⟩

lemma *trace-and-start-determines-end*:
shows
$\mathit{trace}\ c\ t\ c' \Longrightarrow \mathit{trace}\ c\ t\ d' \Longrightarrow c' = d'$
⟨*proof*⟩

lemma *suffix-split-trace*:
shows
$[\![\mathit{trace}\ c\ t\ c';\ \mathit{suffix}\ t'\ t]\!] \Longrightarrow \exists\, c''.\ \mathit{trace}\ c''\ t'\ c'$
⟨*proof*⟩

lemma *prefix-split-trace*:

fixes
c :: ('*p*, '*s*, '*m*) *configuration* **and**
t :: ('*p*, '*s*, '*m*) *trace*
shows
⟦ ∃ *c'*. *trace c t c'*;
prefix t' t
⟧ ⟹ ∃ *c''*. *trace c t' c''*
⟨*proof*⟩

lemma *split-trace*:
shows
⟦ *trace c t c'*;
t = *t'* @ *t''*
⟧ ⟹ ∃ *c''*. *trace c t' c''* ∧ *trace c'' t'' c'*
⟨*proof*⟩

2.2 Describing intermediate configurations

definition *construct-fun-from-rel* :: ('*a* ∗ '*b*) *set* ⇒ '*a* ⇒ '*b* **where**
construct-fun-from-rel R x = (*THE y*. (*x*,*y*) ∈ *R*)

definition *trace-rel* **where**
trace-rel ≡ {((*x*, *t'*), *y*). *trace x t' y*}

lemma *fun-must-admit-trace*:
shows
single-valued R ⟹ *x* ∈ *Domain R*
⟹ (*x*, *construct-fun-from-rel R x*) ∈ *R*
⟨*proof*⟩

lemma *single-valued-trace-rel*:
shows
single-valued trace-rel
⟨*proof*⟩

definition *run-trace* **where**
run-trace ≡ *construct-fun-from-rel trace-rel*

In order to describe intermediate configurations of a trace we introduce the *s* function definition, which, given an initial configuration *c*, a trace *t* and an index $i \in \mathbb{N}$, determines the unique state after the first *i* events of *t*.

definition *s* **where**
s c t i = (*THE c'*. *trace c* (*take i t*) *c'*)

lemma *s-is-partial-execution*:
shows
s c t i = *run-trace* (*c*, *take i t*)
⟨*proof*⟩

lemma *exists-trace-for-any-i*:
assumes
$\exists\, c'.\ trace\ c\ t\ c'$
shows
$trace\ c\ (take\ i\ t)\ (s\ c\ t\ i)$
⟨*proof*⟩

lemma *exists-trace-for-any-i-j*:
assumes
$\exists\, c'.\ trace\ c\ t\ c'$ **and**
$i \leq j$
shows
$trace\ (s\ c\ t\ i)\ (take\ (j - i)\ (drop\ i\ t))\ (s\ c\ t\ j)$
⟨*proof*⟩

lemma *step-Suc*:
assumes
$i < length\ t$ **and**
valid: $trace\ c\ t\ c'$
shows $(s\ c\ t\ i) \vdash (t\ !\ i) \mapsto (s\ c\ t\ (Suc\ i))$
⟨*proof*⟩

2.3 Trace-related lemmas

lemma *snapshot-state-unchanged-trace*:
assumes
$trace\ c\ t\ c'$ **and**
$ps\ c\ p = Some\ u$
shows
$ps\ c'\ p = Some\ u$
⟨*proof*⟩

lemma *no-state-change-if-only-nonregular-events*:
shows
$[\![\ trace\ c\ t\ c';$
$\nexists\, ev.\ ev \in set\ t \wedge regular\text{-}event\ ev \wedge occurs\text{-}on\ ev = p;$
$states\ c\ p = st$
$]\!] \Longrightarrow states\ c'\ p = st$
⟨*proof*⟩

lemma *message-must-be-delivered-2-trace*:
assumes
$trace\ c\ t\ c'$ **and**
$m : set\ (msgs\ c\ i)$ **and**
$m \notin set\ (msgs\ c'\ i)$ **and**
$channel\ i = Some\ (q,\ p)$
shows
$\exists\, ev \in set\ t.\ (\exists\, p\ q.\ ev = RecvMarker\ i\ p\ q \wedge m = Marker) \vee (\exists\, p\ q\ s\ s'\ m'.\ ev = Recv\ i\ q\ p\ s\ s'\ m' \wedge m = Msg\ m')$

⟨*proof*⟩

lemma *marker-must-be-delivered-2-trace*:
assumes
trace c t c′ **and**
Marker : *set* (*msgs c i*) **and**
Marker ∉ *set* (*msgs c′ i*) **and**
channel i = *Some* (*p*, *q*)
shows
∃ *ev* ∈ *set t*. (∃ *p q*. *ev* = *RecvMarker i p q*)
⟨*proof*⟩

lemma *snapshot-stable*:
shows
⟦ *trace c t c′*;
has-snapshotted c p
⟧ ⟹ *has-snapshotted c′ p*
⟨*proof*⟩

lemma *snapshot-stable-2*:
shows
trace c t c′ ⟹ ~ *has-snapshotted c′ p* ⟹ ~ *has-snapshotted c p*
⟨*proof*⟩

lemma *no-markers-if-all-snapshotted*:
shows
⟦ *trace c t c′*;
∀ *p*. *has-snapshotted c p*;
Marker ∉ *set* (*msgs c i*)
⟧ ⟹ *Marker* ∉ *set* (*msgs c′ i*)
⟨*proof*⟩

lemma *event-stays-valid-if-no-occurrence-trace*:
shows
⟦ *trace c t c′*;
list-all (λ*ev*. *occurs-on ev* ≠ *occurs-on ev′*) *t*;
can-occur ev′ c
⟧ ⟹ *can-occur ev′ c′*
⟨*proof*⟩

lemma *event-can-go-back-if-no-sender-trace*:
shows
⟦ *trace c t c′*;
list-all (λ*ev*. *occurs-on ev* ≠ *occurs-on ev′*) *t*;
can-occur ev′ c′;
~ *isRecvMarker ev′*;
list-all (λ*ev*. ~ *isSend ev*) *t*
⟧ ⟹ *can-occur ev′ c*
⟨*proof*⟩

lemma *done-only-from-recv-marker-trace*:
assumes
trace c t c′ **and**
t ≠ [] **and**
snd (*cs c cid*) ≠ *Done* **and**
snd (*cs c′ cid*) = *Done* **and**
channel cid = *Some* (*p*, *q*)
shows
RecvMarker cid q p ∈ *set t*
⟨*proof*⟩

lemma *cs-not-not-started-stable-trace*:
shows
⟦ *trace c t c′*; *snd* (*cs c cid*) ≠ *NotStarted*; *channel cid* = *Some* (*p*, *q*) ⟧ ⟹ *snd* (*cs c′ cid*) ≠ *NotStarted*
⟨*proof*⟩

lemma *no-messages-introduced-if-no-channel*:
assumes
trace: *trace init t final* **and**
no-msgs-if-no-channel: ∀ *i*. *channel i* = *None* ⟶ *msgs init i* = []
shows
channel cid = *None* ⟹ *msgs* (*s init t i*) *cid* = []
⟨*proof*⟩

end

end

3 Utilties

theory *Util*
imports
Main
HOL−Library.Sublist
HOL−Library.Multiset

begin

abbreviation *swap-events* **where**
swap-events i j t ≡ *take i t* @ [*t* ! *j*, *t* ! *i*] @ *take* (*j* − (*i*+1)) (*drop* (*i*+1) *t*) @ *drop* (*j*+1) *t*

lemma *swap-neighbors-2*:
shows
i+1 < *length t* ⟹ *swap-events i* (*i*+1) *t* = (*t*[*i* := *t* ! (*i*+1)])[*i*+1 := *t* ! *i*]
⟨*proof*⟩

lemma *swap-identical-length*:
assumes
$i < j$ **and**
$j < length\ t$
shows
$length\ t = length\ (swap\text{-}events\ i\ j\ t)$
⟨*proof*⟩

lemma *swap-identical-heads*:
assumes
$i < j$ **and**
$j < length\ t$
shows
$take\ i\ t = take\ i\ (swap\text{-}events\ i\ j\ t)$
⟨*proof*⟩

lemma *swap-identical-tails*:
assumes
$i < j$ **and**
$j < length\ t$
shows
$drop\ (j+1)\ t = drop\ (j+1)\ (swap\text{-}events\ i\ j\ t)$
⟨*proof*⟩

lemma *swap-neighbors*:
shows
$i+1 < length\ l \Longrightarrow (l[i := l\ !\ (i+1)])[i+1 := l\ !\ i] = take\ i\ l\ @\ [l\ !\ (i+1),\ l\ !\ i]\ @\ drop\ (i+2)\ l$
⟨*proof*⟩

lemma *swap-events-perm*:
assumes
$i < j$ **and**
$j < length\ t$
shows
$mset\ (swap\text{-}events\ i\ j\ t) = mset\ t$
⟨*proof*⟩

lemma *sum-eq-if-same-subterms*:
fixes
$i :: nat$
shows
$\forall k.\ i \leq k \wedge k < j \longrightarrow f\ k = f'\ k \Longrightarrow sum\ f\ \{i..<j\} = sum\ f'\ \{i..<j\}$
⟨*proof*⟩

lemma *filter-neq-takeWhile*:
assumes
$filter\ ((\neq)\ a)\ l \neq takeWhile\ ((\neq)\ a)\ l$
shows

$\exists\, i\, j.\ i < j \wedge j < \mathit{length}\ l \wedge l\ !\ i = a \wedge l\ !\ j \neq a$ (**is** *?P*)
⟨*proof*⟩

lemma *util-exactly-one-element*:
assumes
$m \notin \mathit{set}\ l$ **and**
$l' = l\ @\ [m]$
shows
$\exists !j.\ j < \mathit{length}\ l' \wedge l'\ !\ j = m$ (**is** *?P*)
⟨*proof*⟩

lemma *exists-one-iff-filter-one*:
shows
$(\exists !j.\ j < \mathit{length}\ l \wedge l\ !\ j = a) \longleftrightarrow \mathit{length}\ (\mathit{filter}\ ((=)\ a)\ l) = 1$
⟨*proof*⟩

end

4 Swap lemmas

theory *Swap*
imports
Distributed-System

begin

context *distributed-system*

begin

lemma *swap-msgs-Trans-Trans*:
assumes
$c \vdash \mathit{ev} \mapsto d$ **and**
$d \vdash \mathit{ev}' \mapsto e$ **and**
isTrans ev **and**
isTrans ev′ **and**
$c \vdash \mathit{ev}' \mapsto d'$ **and**
$d' \vdash \mathit{ev} \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev′*
shows
msgs e i = *msgs e′ i*
⟨*proof*⟩

lemma *swap-msgs-Send-Send*:
assumes
$c \vdash \mathit{ev} \mapsto d$ **and**
$d \vdash \mathit{ev}' \mapsto e$ **and**
isSend ev **and**

isSend ev′ **and**
c ⊢ *ev′* ↦ *d′* **and**
d′ ⊢ *ev* ↦ *e′* **and**
occurs-on ev ≠ *occurs-on ev′*
shows
msgs e i = *msgs e′ i*
⟨*proof*⟩

lemma *swap-msgs-Recv-Recv*:
assumes
c ⊢ *ev* ↦ *d* **and**
d ⊢ *ev′* ↦ *e* **and**
isRecv ev **and**
isRecv ev′ **and**
c ⊢ *ev′* ↦ *d′* **and**
d′ ⊢ *ev* ↦ *e′* **and**
occurs-on ev ≠ *occurs-on ev′*
shows
msgs e i = *msgs e′ i*
⟨*proof*⟩

lemma *swap-msgs-Send-Trans*:
assumes
c ⊢ *ev* ↦ *d* **and**
d ⊢ *ev′* ↦ *e* **and**
isSend ev **and**
isTrans ev′ **and**
c ⊢ *ev′* ↦ *d′* **and**
d′ ⊢ *ev* ↦ *e′* **and**
occurs-on ev ≠ *occurs-on ev′*
shows
msgs e i = *msgs e′ i*
⟨*proof*⟩

lemma *swap-msgs-Trans-Send*:
assumes
c ⊢ *ev* ↦ *d* **and**
d ⊢ *ev′* ↦ *e* **and**
isTrans ev **and**
isSend ev′ **and**
c ⊢ *ev′* ↦ *d′* **and**
d′ ⊢ *ev* ↦ *e′* **and**
occurs-on ev ≠ *occurs-on ev′*
shows
msgs e i = *msgs e′ i*
⟨*proof*⟩

lemma *swap-msgs-Recv-Trans*:
assumes

$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isRecv ev **and**
isTrans ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
msgs e i = *msgs e' i*
⟨*proof*⟩

lemma *swap-msgs-Trans-Recv*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isTrans ev **and**
isRecv ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
msgs e i = *msgs e' i*
⟨*proof*⟩

lemma *swap-msgs-Send-Recv*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isSend ev **and**
isRecv ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
msgs e i = *msgs e' i*
⟨*proof*⟩

lemma *swap-msgs-Recv-Send*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isRecv ev **and**
isSend ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
msgs e i = *msgs e' i*
⟨*proof*⟩

lemma *same-cs-implies-same-resulting-cs*:
assumes
$cs\ c\ i = cs\ d\ i$
$c \vdash ev \mapsto c'$ **and**
$d \vdash ev \mapsto d'$ **and**
regular-event ev
shows
$cs\ c'\ i = cs\ d'\ i$
⟨*proof*⟩

lemma *regular-event-implies-same-channel-snapshot-Recv-Recv*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isRecv ev **and**
isRecv ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *regular-event-implies-same-channel-snapshot-Recv*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
$\sim$ *isRecv ev* **and**
regular-event ev **and**
isRecv ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *same-messages-2*:
assumes
$\forall p.$ *has-snapshotted c p = has-snapshotted d p* **and**
msgs c i = msgs d i **and**
$c \vdash ev \mapsto c'$ **and**
$d \vdash ev \mapsto d'$ **and**
$\sim$ *regular-event ev*
shows
msgs c' i = msgs d' i
⟨*proof*⟩

lemma *same-cs-2*:
assumes
$\forall\, p.$ *has-snapshotted* $c\ p =$ *has-snapshotted* $d\ p$ **and**
cs $c\ i =$ *cs* $d\ i$ **and**
$c \vdash ev \mapsto c'$ **and**
$d \vdash ev \mapsto d'$
shows
cs $c'\ i =$ *cs* $d'\ i$
⟨*proof*⟩

lemma *swap-Snapshot-Trans*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isSnapshot ev **and**
isTrans ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on $ev \neq$ *occurs-on* ev'
shows
msgs $e\ i =$ *msgs* $e'\ i$
⟨*proof*⟩

lemma *swap-msgs-Trans-RecvMarker*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isRecvMarker ev **and**
isTrans ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on $ev \neq$ *occurs-on* ev'
shows
msgs $e'\ i =$ *msgs* $e\ i$
⟨*proof*⟩

lemma *swap-Trans-Snapshot*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isTrans ev **and**
isSnapshot ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on $ev \neq$ *occurs-on* ev'
shows
msgs $e\ i =$ *msgs* $e'\ i$
⟨*proof*⟩

lemma *swap-Send-Snapshot*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isSend ev **and**
isSnapshot ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
msgs e i = *msgs e' i*
⟨*proof*⟩

lemma *swap-Snapshot-Send*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isSnapshot ev **and**
isSend ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
msgs e i = *msgs e' i*
⟨*proof*⟩

lemma *swap-Recv-Snapshot*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isRecv ev **and**
isSnapshot ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
msgs e i = *msgs e' i*
⟨*proof*⟩

lemma *swap-Snapshot-Recv*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isSnapshot ev **and**
isRecv ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows

msgs e i = msgs e′ i
⟨*proof*⟩

lemma *swap-msgs-Recv-RecvMarker*:
assumes
c ⊢ ev ↦ d **and**
d ⊢ ev′ ↦ e **and**
isRecv ev **and**
isRecvMarker ev′ **and**
c ⊢ ev′ ↦ d′ **and**
d′ ⊢ ev ↦ e′ **and**
occurs-on ev ≠ occurs-on ev′
shows
msgs e i = msgs e′ i
⟨*proof*⟩

lemma *swap-RecvMarker-Recv*:
assumes
c ⊢ ev ↦ d **and**
d ⊢ ev′ ↦ e **and**
isRecvMarker ev **and**
isRecv ev′ **and**
c ⊢ ev′ ↦ d′ **and**
d′ ⊢ ev ↦ e′ **and**
occurs-on ev ≠ occurs-on ev′
shows
msgs e i = msgs e′ i
⟨*proof*⟩

lemma *swap-msgs-Send-RecvMarker*:
assumes
c ⊢ ev ↦ d **and**
d ⊢ ev′ ↦ e **and**
isSend ev **and**
isRecvMarker ev′ **and**
c ⊢ ev′ ↦ d′ **and**
d′ ⊢ ev ↦ e′ **and**
occurs-on ev ≠ occurs-on ev′
shows
msgs e i = msgs e′ i
⟨*proof*⟩

lemma *swap-RecvMarker-Send*:
assumes
c ⊢ ev ↦ d **and**
d ⊢ ev′ ↦ e **and**
isRecvMarker ev **and**
isSend ev′ **and**
c ⊢ ev′ ↦ d′ **and**

$d' \vdash ev \mapsto e'$ **and**
$occurs\text{-}on\ ev \neq occurs\text{-}on\ ev'$
shows
$msgs\ e\ i = msgs\ e'\ i$
⟨*proof*⟩

lemma *swap-cs-Trans-Snapshot*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
$isTrans\ ev$ **and**
$isSnapshot\ ev'$ **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *swap-cs-Snapshot-Trans*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
$isSnapshot\ ev$ **and**
$isTrans\ ev'$ **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *swap-cs-Send-Snapshot*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
$isSend\ ev$ **and**
$isSnapshot\ ev'$ **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *swap-cs-Snapshot-Send*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
$isSnapshot\ ev$ **and**
$isSend\ ev'$ **and**
$c \vdash ev' \mapsto d'$ **and**

$d' \vdash ev \mapsto e'$
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *swap-cs-Recv-Snapshot*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isRecv ev **and**
isSnapshot ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *swap-cs-Snapshot-Recv*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isSnapshot ev **and**
isRecv ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *swap-cs-Trans-RecvMarker*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isTrans ev **and**
isRecvMarker ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *swap-cs-RecvMarker-Trans*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isRecvMarker ev **and**
isTrans ev' **and**

$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *swap-cs-Send-RecvMarker*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isSend ev **and**
isRecvMarker ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *swap-cs-RecvMarker-Send*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isRecvMarker ev **and**
isSend ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

lemma *swap-cs-Recv-RecvMarker*:
assumes
$c \vdash ev \mapsto d$ **and**
$d \vdash ev' \mapsto e$ **and**
isRecv ev **and**
isRecvMarker ev' **and**
$c \vdash ev' \mapsto d'$ **and**
$d' \vdash ev \mapsto e'$ **and**
occurs-on ev $\neq$ *occurs-on ev'*
shows
$cs\ e\ i = cs\ e'\ i$
⟨*proof*⟩

end

end

5 The Chandy–Lamport algorithm

theory *Snapshot*
imports
HOL−Library.Sublist
Distributed-System
Trace
Util
Swap

begin

5.1 The computation locale

We extend the distributed system locale presented earlier: Now we are given a trace t of the distributed system between two configurations, the initial and final configuartions of t. Our objective is to show that the Chandy–Lamport algorithm terminated successfully and exhibits the same properties as claimed in [1]. In the initial state no snapshotting must have taken place yet, however the computation itself may have progressed arbitrarily far already.

We assume that there exists at least one process, that the total number of processes in the system is finite, and that there are only finitely many channels between the processes. The process graph is strongly connected. Finally there are Chandy and Lamport's core assumptions: every process snapshots at some time and no marker may remain in a channel forever.

locale *computation* = *distributed-system* +
fixes
init final :: ('*a*, '*b*, '*c*) *configuration*
assumes
finite-channels:
finite {*i*. ∃ *p q*. *channel i* = *Some* (*p*, *q*)} **and**
strongly-connected-raw:
∀ *p q*. (*p* ≠ *q*) ⟶
(*tranclp* (λ*p q*. (∃ *i*. *channel i* = *Some* (*p*, *q*)))) *p q* **and**

at-least-two-processes:
card (*UNIV* :: '*a set*) > *1* **and**
finite-processes:
finite (*UNIV* :: '*a set*) **and**

no-initial-Marker:
∀ *i*. (∃ *p q*. *channel i* = *Some* (*p*, *q*))
⟶ *Marker* ∉ *set* (*msgs init i*) **and**
no-msgs-if-no-channel:
∀ *i*. *channel i* = *None* ⟶ *msgs init i* = [] **and**
no-initial-process-snapshot:

$\forall\, p.\ ^{\sim}$ *has-snapshotted init p* **and**
no-initial-channel-snapshot:
$\forall\, i.$ *channel-snapshot init i* = ([], *NotStarted*) **and**

valid: $\exists\, t.$ *trace init t final* **and**
l1: $\forall\, t\ i\ cid.$ *trace init t final*
$\wedge$ *Marker* $\in$ *set* (*msgs* (*s init t i*) *cid*)
$\longrightarrow$ ($\exists\, j.\ j \geq i \wedge$ *Marker* $\notin$ *set* (*msgs* (*s init t j*) *cid*)) **and**
l2: $\forall\, t\ p.$ *trace init t final*
$\longrightarrow$ ($\exists\, i.$ *has-snapshotted* (*s init t i*) *p* $\wedge$ *i* $\leq$ *length t*)
begin

definition *has-channel* **where**
has-channel p q $\longleftrightarrow$ ($\exists\, i.$ *channel i* = *Some* (*p*, *q*))

lemmas *strongly-connected* = *strongly-connected-raw*[*folded has-channel-def*]

lemma *exists-some-channel*:
shows $\exists\, i\ p\ q.$ *channel i* = *Some* (*p*, *q*)
⟨*proof*⟩

abbreviation *S* **where**
S $\equiv$ *s init*

lemma *no-messages-if-no-channel*:
assumes *trace init t final*
shows *channel cid* = *None* $\Longrightarrow$ *msgs* (*s init t i*) *cid* = []
⟨*proof*⟩

lemma *S-induct* [*consumes 3*, *case-names S-init S-step*]:
⟦ *trace init t final*; *i* $\leq$ *j*; *j* $\leq$ *length t*;
$\bigwedge i.$ *P i i*;
$\bigwedge i\ j.$ *i* < *j* $\Longrightarrow$ *j* $\leq$ *length t* $\Longrightarrow$ (*S t i*) $\vdash$ (*t* ! *i*) $\mapsto$ (*S t* (*Suc i*)) $\Longrightarrow$ *P* (*Suc i*) *j* $\Longrightarrow$ *P i j*
⟧ $\Longrightarrow$ *P i j*
⟨*proof*⟩

lemma *exists-index*:
assumes
trace init t final **and**
ev $\in$ *set* (*take* (*j* − *i*) (*drop i t*))
shows
$\exists\, k.\ i \leq k \wedge k < j \wedge ev = t\ !\ k$
⟨*proof*⟩

lemma *no-change-if-ge-length-t*:
assumes
trace init t final **and**
i $\geq$ *length t* **and**

$j \geq i$
shows
$S\ t\ i = S\ t\ j$
⟨*proof*⟩

lemma *no-marker-if-no-snapshot*:
shows
⟦ *trace init t final*; *channel cid* = *Some* (*p*, *q*);
~ *has-snapshotted* (*S t i*) *p* ⟧
⟹ *Marker* ∉ *set* (*msgs* (*S t i*) *cid*)
⟨*proof*⟩

5.2 Termination

We prove that the snapshot algorithm terminates, as exhibited by lemma `snapshot_algorithm_must_terminate`. In the final configuration all processes have snapshotted, and no markers remain in the channels.

lemma *must-exist-snapshot*:
assumes
trace init t final
shows
∃ *p i*. *Snapshot p* = *t* ! *i*
⟨*proof*⟩

lemma *recv-marker-means-snapshotted*:
assumes
trace init t final **and**
ev = *RecvMarker cid p q* **and**
(*S t i*) ⊢ *ev* ↦ (*S t* (*Suc i*))
shows
has-snapshotted (*S t i*) *q*
⟨*proof*⟩

lemma *recv-marker-means-cs-Done*:
assumes
trace init t final **and**
t ! *i* = *RecvMarker cid p q* **and**
i < *length t*
shows
snd (*cs* (*S t* (*i*+*1*)) *cid*) = *Done*
⟨*proof*⟩

lemma *snapshot-produces-marker*:
assumes
trace init t final **and**
~ *has-snapshotted* (*S t i*) *p* **and**
has-snapshotted (*S t* (*Suc i*)) *p* **and**
channel cid = *Some* (*p*, *q*)
shows

Marker : *set* (*msgs* (*S t* (*Suc i*)) *cid*) ∨ *has-snapshotted* (*S t i*) *q*
⟨*proof*⟩

lemma *exists-snapshot-for-all-p*:
assumes
trace init t final
shows
∃ *i*. ~ *has-snapshotted* (*S t i*) *p* ∧ *has-snapshotted* (*S t* (*Suc i*)) *p* (**is** *?Q*)
⟨*proof*⟩

lemma *all-processes-snapshotted-in-final-state*:
assumes
trace init t final
shows
has-snapshotted final p
⟨*proof*⟩

definition *next-marker-free-state* **where**
next-marker-free-state t i cid = (*LEAST j*. *j* ≥ *i* ∧ *Marker* ∉ *set* (*msgs* (*S t j*) *cid*))

lemma *exists-next-marker-free-state*:
assumes
channel cid = *Some* (*p*, *q*)
trace init t final
shows
∃!*j*. *next-marker-free-state t i cid* = *j* ∧ *j* ≥ *i* ∧ *Marker* ∉ *set* (*msgs* (*S t j*) *cid*)
⟨*proof*⟩

theorem *snapshot-algorithm-must-terminate*:
assumes
trace init t final
shows
∃ *phi*. ((∀ *p*. *has-snapshotted* (*S t phi*) *p*)
∧ (∀ *cid*. *Marker* ∉ *set* (*msgs* (*S t phi*) *cid*)))
⟨*proof*⟩

5.3 Correctness

The greatest part of this work is spent on the correctness of the Chandy-Lamport algorithm. We prove that the snapshot is consistent, i.e. there exists a permutation t' of the trace t and an intermediate configuration c' of t' such that the configuration recorded in the snapshot corresponds to the snapshot taken during execution of t, which is given as Theorem 1 in [1].

lemma *snapshot-stable-ver-2*:
shows *trace init t final* ⟹ *has-snapshotted* (*S t i*) *p* ⟹ *j* ≥ *i* ⟹ *has-snapshotted* (*S t j*) *p*

⟨*proof*⟩

lemma *snapshot-stable-ver-3*:
shows *trace init t final* $\Longrightarrow$ $\sim$ *has-snapshotted* (*S t i*) *p* $\Longrightarrow$ *i* $\geq$ *j* $\Longrightarrow$ $\sim$ *has-snapshotted* (*S t j*) *p*
⟨*proof*⟩

lemma *marker-must-stay-if-no-snapshot*:
assumes
trace init t final **and**
has-snapshotted (*S t i*) *p* **and**
$\sim$ *has-snapshotted* (*S t i*) *q* **and**
channel cid = *Some* (*p*, *q*)
shows
Marker : *set* (*msgs* (*S t i*) *cid*)
⟨*proof*⟩

5.3.1 Pre- and postrecording events

definition *prerecording-event*:
prerecording-event t i ≡
i < *length t* ∧ *regular-event* (*t* ! *i*)
∧ $\sim$ *has-snapshotted* (*S t i*) (*occurs-on* (*t* ! *i*))

definition *postrecording-event*:
postrecording-event t i ≡
i < *length t* ∧ *regular-event* (*t* ! *i*)
∧ *has-snapshotted* (*S t i*) (*occurs-on* (*t* ! *i*))

abbreviation *neighboring* **where**
neighboring t i j ≡ *i* < *j* ∧ *j* < *length t* ∧ *regular-event* (*t* ! *i*) ∧ *regular-event* (*t* ! *j*)
∧ (∀ *k*. *i* < *k* ∧ *k* < *j* ⟶ $\sim$ *regular-event* (*t* ! *k*))

lemma *pre-if-regular-and-not-post*:
assumes
regular-event (*t* ! *i*) **and**
$\sim$ *postrecording-event t i* **and**
i < *length t*
shows
prerecording-event t i
⟨*proof*⟩

lemma *post-if-regular-and-not-pre*:
assumes
regular-event (*t* ! *i*) **and**
$\sim$ *prerecording-event t i* **and**
i < *length t*
shows

postrecording-event t i
⟨*proof*⟩

lemma *post-before-pre-different-processes*:
assumes
i < *j* **and**
j < *length t* **and**
neighboring: ∀ *k*. (*i* < *k* ∧ *k* < *j*) ⟶ ~ *regular-event* (*t* ! *k*) **and**
post-ei: *postrecording-event t i* **and**
pre-ej: *prerecording-event t j* **and**
valid: *trace init t final*
shows
occurs-on (*t* ! *i*) ≠ *occurs-on* (*t* ! *j*)
⟨*proof*⟩

lemma *post-before-pre-neighbors*:
assumes
i < *j* **and**
j < *length t* **and**
neighboring: ∀ *k*. (*i* < *k* ∧ *k* < *j*) ⟶ ~ *regular-event* (*t* ! *k*) **and**
post-ei: *postrecording-event t i* **and**
pre-ej: *prerecording-event t j* **and**
valid: *trace init t final*
shows
Ball (*set* (*take* (*j* − (*i*+*1*)) (*drop* (*i*+*1*) *t*))) (%*ev*. ~ *regular-event ev* ∧ ~ *occurs-on ev* = *occurs-on* (*t* ! *j*))
⟨*proof*⟩

lemma *can-swap-neighboring-pre-and-postrecording-events*:
assumes
i < *j* **and**
j < *length t* **and**
occurs-on (*t* ! *i*) = *p* **and**
occurs-on (*t* ! *j*) = *q* **and**
neighboring: ∀ *k*. (*i* < *k* ∧ *k* < *j*)
⟶ ~ *regular-event* (*t* ! *k*) **and**
post-ei: *postrecording-event t i* **and**
pre-ej: *prerecording-event t j* **and**
valid: *trace init t final*
shows
can-occur (*t* ! *j*) (*S t i*)
⟨*proof*⟩

5.3.2 Event swapping

lemma *swap-events*:
shows ⟦ *i* < *j*; *j* < *length t*;
∀ *k*. (*i* < *k* ∧ *k* < *j*) ⟶ ~ *regular-event* (*t* ! *k*);
postrecording-event t i; *prerecording-event t j*;

$$\begin{array}{l} \quad \mathit{trace}\ \mathit{init}\ t\ \mathit{final}\ ⟧ \\ \Longrightarrow \mathit{trace}\ \mathit{init}\ (\mathit{swap\text{-}events}\ i\ j\ t)\ \mathit{final} \\ \quad \wedge\ (\forall k.\ k \geq j + 1 \longrightarrow S\ (\mathit{swap\text{-}events}\ i\ j\ t)\ k = S\ t\ k) \\ \quad \wedge\ (\forall k.\ k \leq i \longrightarrow S\ (\mathit{swap\text{-}events}\ i\ j\ t)\ k = S\ t\ k) \\ \quad \wedge\ \mathit{prerecording\text{-}event}\ (\mathit{swap\text{-}events}\ i\ j\ t)\ i \\ \quad \wedge\ \mathit{postrecording\text{-}event}\ (\mathit{swap\text{-}events}\ i\ j\ t)\ (i{+}1) \\ \quad \wedge\ (\forall k.\ k > i{+}1 \wedge k < j{+}1 \\ \qquad \longrightarrow {}^{\sim}\ \mathit{regular\text{-}event}\ ((\mathit{swap\text{-}events}\ i\ j\ t)\ !\ k)) \end{array}$$

⟨*proof*⟩

5.3.3 Relating configurations and the computed snapshot

definition *ps-equal-to-snapshot* **where**
ps-equal-to-snapshot c c′ ≡
∀ *p*. *Some* (*states c p*) = *process-snapshot c′ p*

definition *cs-equal-to-snapshot* **where**
cs-equal-to-snapshot c c′ ≡
∀ *cid*. *channel cid* ≠ *None*
⟶ *filter* ((≠) *Marker*) (*msgs c cid*)
= *map Msg* (*fst* (*channel-snapshot c′ cid*))

definition *state-equal-to-snapshot* **where**
state-equal-to-snapshot c c′ ≡
ps-equal-to-snapshot c c′ ∧ *cs-equal-to-snapshot c c′*

lemma *init-is-s-t-0*:
assumes
trace init t final
shows
init = (*S t 0*)
⟨*proof*⟩

lemma *final-is-s-t-len-t*:
assumes
trace init t final
shows
final = *S t* (*length t*)
⟨*proof*⟩

lemma *snapshot-event*:
assumes
trace init t final **and**
~ *has-snapshotted* (*S t i*) *p* **and**
has-snapshotted (*S t* (*i*+1)) *p*
shows
isSnapshot (*t* ! *i*) ∨ *isRecvMarker* (*t* ! *i*)
⟨*proof*⟩

lemma *snapshot-state*:
assumes
trace init t final **and**
states (*S t i*) *p* = *u* **and**
~ *has-snapshotted* (*S t i*) *p* **and**
has-snapshotted (*S t* (*i*+*1*)) *p*
shows
ps (*S t* (*i*+*1*)) *p* = *Some u*
⟨*proof*⟩

lemma *snapshot-state-unchanged-trace-2*:
shows
⟦ *trace init t final*; *i* ≤ *j*; *j* ≤ *length t*;
ps (*S t i*) *p* = *Some u*
⟧ ⟹ *ps* (*S t j*) *p* = *Some u*
⟨*proof*⟩

lemma *no-recording-cs-if-not-snapshotted*:
shows
⟦ *trace init t final*; ~ *has-snapshotted* (*S t i*) *p*;
channel cid = *Some* (*q*, *p*) ⟧ ⟹ *cs* (*S t i*) *cid* = *cs init cid*
⟨*proof*⟩

lemma *cs-done-implies-has-snapshotted*:
assumes
trace init t final **and**
snd (*cs* (*S t i*) *cid*) = *Done* **and**
channel cid = *Some* (*p*, *q*)
shows
has-snapshotted (*S t i*) *q*
⟨*proof*⟩

lemma *exactly-one-snapshot*:
assumes
trace init t final
shows
∃!*i*. ~ *has-snapshotted* (*S t i*) *p* ∧ *has-snapshotted* (*S t* (*i*+*1*)) *p* (**is** *?P*)
⟨*proof*⟩

lemma *initial-cs-changes-implies-nonregular-event*:
assumes
trace init t final **and**
snd (*cs* (*S t i*) *cid*) = *NotStarted* **and**
snd (*cs* (*S t* (*i*+*1*)) *cid*) ≠ *NotStarted* **and**
channel cid = *Some* (*p*, *q*)
shows
~ *regular-event* (*t* ! *i*)
⟨*proof*⟩

lemma *cs-in-initial-state-implies-not-snapshotted*:
assumes
trace init t final **and**
snd (*cs* (*S t i*) *cid*) = *NotStarted* **and**
channel cid = *Some* (*p*, *q*)
shows
~ *has-snapshotted* (*S t i*) *q*
⟨*proof*⟩

lemma *nonregular-event-in-initial-state-implies-cs-changed*:
assumes
trace init t final **and**
snd (*cs* (*S t i*) *cid*) = *NotStarted* **and**
~ *regular-event* (*t* ! *i*) **and**
occurs-on (*t* ! *i*) = *q* **and**
channel cid = *Some* (*p*, *q*) **and**
i < *length t*
shows
snd (*cs* (*S t* (*i*+*1*)) *cid*) ≠ *NotStarted*
⟨*proof*⟩

lemma *cs-recording-implies-snapshot*:
assumes
trace init t final **and**
snd (*cs* (*S t i*) *cid*) = *Recording* **and**
channel cid = *Some* (*p*, *q*)
shows
has-snapshotted (*S t i*) *q*
⟨*proof*⟩

lemma *cs-done-implies-both-snapshotted*:
assumes
trace init t final **and**
snd (*cs* (*S t i*) *cid*) = *Done* **and**
i < *length t* **and**
channel cid = *Some* (*p*, *q*)
shows
has-snapshotted (*S t i*) *p*
has-snapshotted (*S t i*) *q*
⟨*proof*⟩

lemma *cs-done-implies-same-snapshots*:
assumes *trace init t final i* ≤ *j j* ≤ *length t*
shows *snd* (*cs* (*S t i*) *cid*) = *Done* ⟹ *channel cid* = *Some* (*p*, *q*) ⟹ *cs* (*S t i*) *cid* = *cs* (*S t j*) *cid*
⟨*proof*⟩

lemma *snapshotted-and-not-done-implies-marker-in-channel*:
assumes

trace init t final **and**
has-snapshotted (*S t i*) *p* **and**
snd (*cs* (*S t i*) *cid*) ≠ *Done* **and**
i ≤ *length t* **and**
channel cid = *Some* (*p*, *q*)
shows
Marker : *set* (*msgs* (*S t i*) *cid*)
⟨*proof*⟩

lemma *no-marker-left-in-final-state*:
assumes
trace init t final
shows
Marker ∉ *set* (*msgs final cid*) (**is** *?P*)
⟨*proof*⟩

lemma *all-channels-done-in-final-state*:
assumes
trace init t final **and**
channel cid = *Some* (*p*, *q*)
shows
snd (*cs final cid*) = *Done*
⟨*proof*⟩

lemma *cs-NotStarted-implies-empty-cs*:
shows
⟦ *trace init t final*; *channel cid* = *Some* (*p*, *q*); *i* < *length t*; ~ *has-snapshotted* (*S t i*) *q* ⟧
⟹ *cs* (*S t i*) *cid* = ([], *NotStarted*)
⟨*proof*⟩

lemma *fst-changed-by-recv-recording-trace*:
assumes
i < *j* **and**
j ≤ *length t* **and**
trace init t final **and**
fst (*cs* (*S t i*) *cid*) ≠ *fst* (*cs* (*S t j*) *cid*) **and**
channel cid = *Some* (*p*, *q*)
shows
∃ *k*. *i* ≤ *k* ∧ *k* < *j* ∧ (∃ *p q u u' m*. *t* ! *k* = *Recv cid q p u u' m*) ∧ (*snd* (*cs* (*S t k*) *cid*) = *Recording*) (**is** *?P*)
⟨*proof*⟩

lemma *cs-not-nil-implies-postrecording-event*:
assumes
trace init t final **and**
fst (*cs* (*S t i*) *cid*) ≠ [] **and**
i ≤ *length t* **and**
channel cid = *Some* (*p*, *q*)

shows
$\exists j.\ j < i \wedge$ *postrecording-event t j*
⟨*proof*⟩

5.3.4 Relating process states

lemma *snapshot-state-must-have-been-reached*:
assumes
trace init t final **and**
ps final p = *Some u* **and**
$\sim$ *has-snapshotted* (*S t i*) *p* **and**
has-snapshotted (*S t* (*i*+*1*)) *p* **and**
i < *length t*
shows
states (*S t i*) *p* = *u*
⟨*proof*⟩

lemma *ps-after-all-prerecording-events*:
assumes
trace init t final **and**
$\forall i'.\ i' \geq i \longrightarrow \sim$ *prerecording-event t i'* **and**
$\forall j'.\ j' < i \longrightarrow \sim$ *postrecording-event t j'*
shows
ps-equal-to-snapshot (*S t i*) *final*
⟨*proof*⟩

5.3.5 Relating channel states

lemma *cs-when-recording*:
shows
⟦ *i* < *j*; *j* ≤ *length t*; *trace init t final*;
has-snapshotted (*S t i*) *p*;
snd (*cs* (*S t i*) *cid*) = *Recording*;
snd (*cs* (*S t j*) *cid*) = *Done*;
channel cid = *Some* (*p*, *q*) ⟧
⟹ *map Msg* (*fst* (*cs* (*S t j*) *cid*))
= *map Msg* (*fst* (*cs* (*S t i*) *cid*)) @ *takeWhile* ((≠) *Marker*) (*msgs* (*S t i*) *cid*)
⟨*proof*⟩

lemma *cs-when-recording-2*:
shows
⟦ *i* ≤ *j*; *trace init t final*;
$\sim$ *has-snapshotted* (*S t i*) *p*;
$\forall k.\ i \leq k \wedge k < j \longrightarrow \sim$ *occurs-on* (*t* ! *k*) = *p*;
snd (*cs* (*S t i*) *cid*) = *Recording*;
channel cid = *Some* (*p*, *q*) ⟧
⟹ *map Msg* (*fst* (*cs* (*S t j*) *cid*)) @ *takeWhile* ((≠) *Marker*) (*msgs* (*S t j*) *cid*)

$= map\ Msg\ (fst\ (cs\ (S\ t\ i)\ cid))\ @\ takeWhile\ ((\neq)\ Marker)\ (msgs\ (S\ t\ i)\ cid)$
$\wedge\ snd\ (cs\ (S\ t\ j)\ cid) = Recording$
⟨*proof*⟩

lemma *cs-when-recording-3*:
shows
$[\![\ i \leq j;\ trace\ init\ t\ final;$
$\sim\ has\text{-}snapshotted\ (S\ t\ i)\ q;$
$\forall k.\ i \leq k \wedge k < j \longrightarrow \sim\ occurs\text{-}on\ (t\ !\ k) = q;$
$snd\ (cs\ (S\ t\ i)\ cid) = NotStarted;$
$has\text{-}snapshotted\ (S\ t\ i)\ p;$
$Marker : set\ (msgs\ (S\ t\ i)\ cid);$
$channel\ cid = Some\ (p,\ q)\ ]\!]$
$\Longrightarrow map\ Msg\ (fst\ (cs\ (S\ t\ j)\ cid))\ @\ takeWhile\ ((\neq)\ Marker)\ (msgs\ (S\ t\ j)\ cid)$
$= map\ Msg\ (fst\ (cs\ (S\ t\ i)\ cid))\ @\ takeWhile\ ((\neq)\ Marker)\ (msgs\ (S\ t\ i)\ cid)$
$\wedge\ snd\ (cs\ (S\ t\ j)\ cid) = NotStarted$
⟨*proof*⟩

lemma *at-most-one-marker*:
shows
$[\![\ trace\ init\ t\ final;\ channel\ cid = Some\ (p,\ q)\ ]\!]$
$\Longrightarrow Marker \notin set\ (msgs\ (S\ t\ i)\ cid)$
$\vee\ (\exists!j.\ j < length\ (msgs\ (S\ t\ i)\ cid) \wedge msgs\ (S\ t\ i)\ cid\ !\ j = Marker)$
⟨*proof*⟩

lemma *last-changes-implies-send-when-msgs-nonempty*:
assumes
$trace\ init\ t\ final$ **and**
$msgs\ (S\ t\ i)\ cid \neq []$ **and**
$msgs\ (S\ t\ (i{+}1))\ cid \neq []$ **and**
$last\ (msgs\ (S\ t\ i)\ cid) = Marker$ **and**
$last\ (msgs\ (S\ t\ (i{+}1))\ cid) \neq Marker$ **and**
$channel\ cid = Some\ (p,\ q)$
shows
$(\exists\ u\ u'\ m.\ t\ !\ i = Send\ cid\ p\ q\ u\ u'\ m)$
⟨*proof*⟩

lemma *no-marker-after-RecvMarker*:
assumes
$trace\ init\ t\ final$ **and**
$(S\ t\ i) \vdash RecvMarker\ cid\ p\ q \mapsto (S\ t\ (i{+}1))$ **and**
$channel\ cid = Some\ (q,\ p)$
shows
$Marker \notin set\ (msgs\ (S\ t\ (i{+}1))\ cid)$
⟨*proof*⟩

lemma *no-marker-and-snapshotted-implies-no-more-markers-trace*:
shows
$[\![$ *trace init t final*; $i \leq j$; $j \leq$ *length t*;
has-snapshotted $(S\ t\ i)\ p$;
Marker $\notin$ *set* $(msgs\ (S\ t\ i)\ cid)$;
channel cid = *Some* $(p,\ q)$ $]\!]$
$\Longrightarrow$ *Marker* $\notin$ *set* $(msgs\ (S\ t\ j)\ cid)$
⟨*proof*⟩

lemma *marker-not-vanishing-means-always-present*:
shows
$[\![$ *trace init t final*; $i \leq j$; $j \leq$ *length t*;
Marker : *set* $(msgs\ (S\ t\ i)\ cid)$;
Marker : *set* $(msgs\ (S\ t\ j)\ cid)$;
channel cid = *Some* $(p,\ q)$
$]\!] \Longrightarrow \forall k.\ i \leq k \wedge k \leq j \longrightarrow$ *Marker* : *set* $(msgs\ (S\ t\ k)\ cid)$
⟨*proof*⟩

lemma *last-stays-if-no-recv-marker-and-no-send*:
shows $[\![$ *trace init t final*; $i < j$; $j \leq$ *length t*;
last $(msgs\ (S\ t\ i)\ cid)$ = *Marker*;
Marker : *set* $(msgs\ (S\ t\ i)\ cid)$;
Marker : *set* $(msgs\ (S\ t\ j)\ cid)$;
$\forall k.\ i \leq k \wedge k < j \longrightarrow {}^{\sim}\ (\exists u\ u'\ m.\ t\ !\ k = Send\ cid\ p\ q\ u\ u'\ m)$;
channel cid = *Some* $(p,\ q)$ $]\!]$
$\Longrightarrow$ *last* $(msgs\ (S\ t\ j)\ cid)$ = *Marker*
⟨*proof*⟩

lemma *last-changes-implies-send-when-msgs-nonempty-trace*:
assumes
trace init t final
$i < j$
$j \leq$ *length t*
Marker : *set* $(msgs\ (S\ t\ i)\ cid)$
Marker : *set* $(msgs\ (S\ t\ j)\ cid)$
last $(msgs\ (S\ t\ i)\ cid)$ = *Marker*
last $(msgs\ (S\ t\ j)\ cid) \neq$ *Marker*
channel cid = *Some* $(p,\ q)$
shows
$\exists k\ u\ u'\ m.\ i \leq k \wedge k < j \wedge t\ !\ k = Send\ cid\ p\ q\ u\ u'\ m$
⟨*proof*⟩

lemma *msg-after-marker-and-nonempty-implies-postrecording-event*:
assumes
trace init t final **and**
Marker : *set* $(msgs\ (S\ t\ i)\ cid)$ **and**
Marker $\neq$ *last* $(msgs\ (S\ t\ i)\ cid)$ **and**
$i \leq$ *length t* **and**
channel cid = *Some* $(p,\ q)$

shows
$\exists j.\ j < i \wedge$ *postrecording-event* $t\ j$ (**is** *?P*)
⟨*proof*⟩

lemma *same-messages-if-no-occurrence-trace*:
shows
⟦ *trace init t final*; $i \leq j$; $j \leq$ *length* t;
$(\forall k.\ i \leq k \wedge k < j \longrightarrow$ *occurs-on* $(t\ !\ k) \neq p \wedge$ *occurs-on* $(t\ !\ k) \neq q)$;
channel cid = *Some* $(p,\ q)$ ⟧
$\Longrightarrow$ *msgs* $(S\ t\ i)$ *cid* = *msgs* $(S\ t\ j)$ *cid* $\wedge$ *cs* $(S\ t\ i)$ *cid* = *cs* $(S\ t\ j)$ *cid*
⟨*proof*⟩

lemma *snapshot-step-cs-preservation-p*:
assumes
$c \vdash ev \mapsto c'$ **and**
$\sim$ *regular-event ev* **and**
occurs-on ev = p **and**
channel cid = *Some* $(p,\ q)$
shows
map Msg (*fst* (*cs c cid*)) @ *takeWhile* $((\neq)$ *Marker*$)$ (*msgs c cid*)
= *map Msg* (*fst* (*cs* c' *cid*)) @ *takeWhile* $((\neq)$ *Marker*$)$ (*msgs* c' *cid*)
$\wedge$ *snd* (*cs c cid*) = *snd* (*cs* c' *cid*)
⟨*proof*⟩

lemma *snapshot-step-cs-preservation-q*:
assumes
$c \vdash ev \mapsto c'$ **and**
$\sim$ *regular-event ev* **and**
occurs-on ev = q **and**
channel cid = *Some* $(p,\ q)$ **and**
Marker $\notin$ *set* (*msgs c cid*) **and**
$\sim$ *has-snapshotted c q*
shows
map Msg (*fst* (*cs c cid*)) @ *takeWhile* $((\neq)$ *Marker*$)$ (*msgs c cid*)
= *map Msg* (*fst* (*cs* c' *cid*)) @ *takeWhile* $((\neq)$ *Marker*$)$ (*msgs* c' *cid*)
$\wedge$ *snd* (*cs* c' *cid*) = *Recording*
⟨*proof*⟩

lemma *Marker-in-channel-implies-not-done*:
assumes
trace init t final **and**
Marker : *set* (*msgs* $(S\ t\ i)$ *cid*) **and**
channel cid = *Some* $(p,\ q)$ **and**
$i \leq$ *length* t
shows
snd (*cs* $(S\ t\ i)$ *cid*) $\neq$ *Done*
⟨*proof*⟩

lemma *keep-empty-if-no-events*:

shows
$[\![$ *trace init t final*; $i \leq j$; $j \leq$ *length t*;
msgs $(S\ t\ i)$ *cid* $= []$;
has-snapshotted $(S\ t\ i)$ *p*;
channel cid $=$ *Some* (p, q);
$\forall k.\ i \leq k \wedge k < j \wedge$ *regular-event* $(t\ !\ k) \longrightarrow {}^{\sim}$ *occurs-on* $(t\ !\ k) = p\ ]\!]$
$\Longrightarrow$ *msgs* $(S\ t\ j)$ *cid* $= []$
⟨*proof*⟩

lemma *last-unchanged-or-empty-if-no-events*:
shows
$[\![$ *trace init t final*; $i \leq j$; $j \leq$ *length t*;
msgs $(S\ t\ i)$ *cid* $\neq []$;
last (*msgs* $(S\ t\ i)$ *cid*) $=$ *Marker*;
has-snapshotted $(S\ t\ i)$ *p*;
channel cid $=$ *Some* (p, q);
$\forall k.\ i \leq k \wedge k < j \wedge$ *regular-event* $(t\ !\ k) \longrightarrow {}^{\sim}$ *occurs-on* $(t\ !\ k) = p\ ]\!]$
$\Longrightarrow$ *msgs* $(S\ t\ j)$ *cid* $= [] \vee$ (*msgs* $(S\ t\ j)$ *cid* $\neq [] \wedge$ *last* (*msgs* $(S\ t\ j)$ *cid*) $=$ *Marker*)
⟨*proof*⟩

lemma *cs-after-all-prerecording-events*:
assumes
trace init t final **and**
$\forall i'.\ i' \geq i \longrightarrow {}^{\sim}$ *prerecording-event t* i' **and**
$\forall j'.\ j' < i \longrightarrow {}^{\sim}$ *postrecording-event t* j' **and**
$i \leq$ *length t*
shows
cs-equal-to-snapshot $(S\ t\ i)$ *final*
⟨*proof*⟩

lemma *snapshot-after-all-prerecording-events*:
assumes
trace init t final **and**
$\forall i'.\ i' \geq i \longrightarrow {}^{\sim}$ *prerecording-event t* i' **and**
$\forall j'.\ j' < i \longrightarrow {}^{\sim}$ *postrecording-event t* j' **and**
$i \leq$ *length t*
shows
state-equal-to-snapshot $(S\ t\ i)$ *final*
⟨*proof*⟩

5.4 Obtaining the desired traces

abbreviation *all-prerecording-before-postrecording* **where**
all-prerecording-before-postrecording t $\equiv \exists i.\ (\forall j.\ j < i \longrightarrow {}^{\sim}$ *postrecording-event t j*)
$\wedge\ (\forall j.\ j \geq i \longrightarrow {}^{\sim}$ *prerecording-event t j*)
$\wedge\ i \leq$ *length t*
$\wedge$ *trace init t final*

definition *count-violations* :: ('*a*, '*b*, '*c*) *trace* ⇒ *nat* **where**
count-violations t = *sum* (λ*i*. *if postrecording-event t i*
then card {*j* ∈ {*i*+1..<*length t*}. *prerecording-event t j*}
else 0)
{*0*..<*length t*}

lemma *violations-ge-0*:
shows
(*if postrecording-event t i*
then card {*j* ∈ {*i*+1..<*length t*}. *prerecording-event t j*}
else 0) ≥ *0*
⟨*proof*⟩

lemma *count-violations-ge-0*:
shows
count-violations t ≥ *0*
⟨*proof*⟩

lemma *violations-0-implies-all-subterms-0*:
assumes
count-violations t = *0*
shows
∀ *i* ∈ {*0*..<*length t*}. (*if postrecording-event t i*
then card {*j* ∈ {*i*+1..<*length t*}. *prerecording-event t j*}
else 0) = *0*
⟨*proof*⟩

lemma *exists-postrecording-violation-if-count-greater-0*:
assumes
count-violations t > *0*
shows
∃ *i*. *postrecording-event t i* ∧ *card* {*j* ∈ {*i*+1..<*length t*}. *prerecording-event t j*} > *0* (**is** *?P*)
⟨*proof*⟩

lemma *exists-prerecording-violation-when-card-greater-0*:
assumes
card {*j* ∈ {*i*+1..<*length t*}. *prerecording-event t j*} > *0*
shows
∃ *j* ∈ {*i*+1..<*length t*}. *prerecording-event t j*
⟨*proof*⟩

lemma *card-greater-0-if-post-after-pre*:
assumes
i < *j* **and**
postrecording-event t i **and**
prerecording-event t j
shows

card $\{j \in \{i+1..<\textit{length } t\}.\ \textit{prerecording-event } t\ j\} > 0$
⟨*proof*⟩

lemma *exists-neighboring-violation-pair*:
assumes
trace init t final **and**
count-violations $t > 0$
shows
$\exists\, i\ j.\ i < j \wedge \textit{postrecording-event } t\ i \wedge \textit{prerecording-event } t\ j$
$\wedge\ (\forall\, k.\ (i < k \wedge k < j) \longrightarrow {}^{\sim}\ \textit{regular-event}\ (t\ !\ k)) \wedge j < \textit{length } t$
⟨*proof*⟩

lemma *same-cardinality-post-swap-1*:
assumes
prerecording-event t j **and**
postrecording-event t i **and**
$i < j$ **and**
$j < \textit{length } t$ **and**
count-violations $t = \textit{Suc } n$ **and**
$\forall\, k.\ (i < k \wedge k < j) \longrightarrow {}^{\sim}\ \textit{regular-event}\ (t\ !\ k)$ **and**
trace init t final
shows
$\{k \in \{0..<i\}.\ \textit{prerecording-event } t\ k\}$
$= \{k \in \{0..<i\}.\ \textit{prerecording-event}\ (\textit{swap-events } i\ j\ t)\ k\}$
⟨*proof*⟩

lemma *same-cardinality-post-swap-2*:
assumes
prerecording-event t j **and**
postrecording-event t i **and**
$i < j$ **and**
$j < \textit{length } t$ **and**
count-violations $t = \textit{Suc } n$ **and**
$\forall\, k.\ (i < k \wedge k < j) \longrightarrow {}^{\sim}\ \textit{regular-event}\ (t\ !\ k)$ **and**
trace init t final
shows
card $\{k \in \{i..<j+1\}.\ \textit{prerecording-event } t\ k\}$
$= \textit{card}\ \{k \in \{i..<j+1\}.\ \textit{prerecording-event}\ (\textit{swap-events } i\ j\ t)\ k\}$
⟨*proof*⟩

lemma *same-cardinality-post-swap-3*:
assumes
prerecording-event t j **and**
postrecording-event t i **and**
$i < j$ **and**
$j < \textit{length } t$ **and**
count-violations $t = \textit{Suc } n$ **and**
$\forall\, k.\ (i < k \wedge k < j) \longrightarrow {}^{\sim}\ \textit{regular-event}\ (t\ !\ k)$ **and**
trace init t final

shows
$\{k \in \{j+1..<length\ t\}.\ prerecording\text{-}event\ t\ k\}$
$= \{k \in \{j+1..<length\ (swap\text{-}events\ i\ j\ t)\}.\ prerecording\text{-}event\ (swap\text{-}events\ i\ j\ t)\ k\}$
⟨*proof*⟩

lemma *card-ip1-to-j-is-1-in-normal-events*:
assumes
prerecording-event t j **and**
postrecording-event t i **and**
$i < j$ **and**
$j < length\ t$ **and**
count-violations t $= Suc\ n$ **and**
$\forall k.\ (i < k \wedge k < j) \longrightarrow \sim regular\text{-}event\ (t\ !\ k)$ **and**
count-violations t $= Suc\ n$ **and**
trace init t final
shows
card $\{k \in \{i+1..<j+1\}.\ prerecording\text{-}event\ t\ k\} = 1$
⟨*proof*⟩

lemma *card-ip1-to-j-is-0-in-swapped-events*:
assumes
prerecording-event t j **and**
postrecording-event t i **and**
$i < j$ **and**
$j < length\ t$ **and**
count-violations t $= Suc\ n$ **and**
$\forall k.\ (i < k \wedge k < j) \longrightarrow \sim regular\text{-}event\ (t\ !\ k)$ **and**
count-violations t $= Suc\ n$ **and**
trace init t final
shows
card $\{k \in \{i+1..<j+1\}.\ prerecording\text{-}event\ (swap\text{-}events\ i\ j\ t)\ k\} = 0$
⟨*proof*⟩

lemma *count-violations-swap*:
assumes
prerecording-event t j **and**
postrecording-event t i **and**
$i < j$ **and**
$j < length\ t$ **and**
count-violations t $= Suc\ n$ **and**
$\forall k.\ (i < k \wedge k < j) \longrightarrow \sim regular\text{-}event\ (t\ !\ k)$ **and**
count-violations t $= Suc\ n$ **and**
trace init t final
shows
count-violations $(swap\text{-}events\ i\ j\ t) = n$
⟨*proof*⟩

lemma *desired-trace-always-exists*:

assumes
trace init t final
shows
$\exists\, t'.\ mset\ t' = mset\ t$
$\quad \wedge\ all\text{-}prerecording\text{-}before\text{-}postrecording\ t'$
⟨*proof*⟩

theorem *snapshot-algorithm-is-correct*:
assumes
trace init t final
shows
$\exists\, t'\ i.\ trace\ init\ t'\ final \wedge mset\ t' = mset\ t$
$\quad \wedge\ state\text{-}equal\text{-}to\text{-}snapshot\ (S\ t'\ i)\ final \wedge i \leq length\ t'$
⟨*proof*⟩

5.5 Stable property detection

Finally, we show that the computed snapshot is indeed suitable for stable property detection, as claimed in [1].

definition *stable* **where**
$stable\ p \equiv (\forall\, c.\ p\ c \longrightarrow (\forall\, t\ c'.\ trace\ c\ t\ c' \longrightarrow p\ c'))$

lemma *has-snapshot-stable*:
assumes
trace init t final
shows
stable (λ*c*. *has-snapshotted c p*)
⟨*proof*⟩

definition *some-snapshot-state* **where**
some-snapshot-state t ≡
SOME $(t',\ i)$. *trace init t final*
$\wedge$ *trace init t' final* $\wedge$ *mset t' = mset t*
$\wedge$ *state-equal-to-snapshot* $(S\ t'\ i)$ *final*

lemma *split-S*:
assumes
trace init t final
shows
trace $(S\ t\ i)$ $(drop\ i\ t)$ *final*
⟨*proof*⟩

theorem *Stable-Property-Detection*:
assumes
stable p **and**
trace init t final **and**
$(t',\ i) = some\text{-}snapshot\text{-}state\ t$ **and**
$p\ (S\ t'\ i)$
shows

p final
⟨*proof*⟩

end

end
theory *Co-Snapshot*
imports
Snapshot
Ordered-Resolution-Prover.Lazy-List-Chain
begin

6 Extension to infinite traces

The computation locale assumes that there already exists a known final configuration c' to the given initial c and trace t. However, we can show that the snapshot algorithm must terminate correctly even if the underlying computation itself does not terminate. We relax the trace relation to allow for a potentially infinite number of "intermediate" events, and show that the algorithm's correctness still holds when imposing the same constraints as in the computation locale.

We use a preexisting theory of lazy list chains by Schlichtkrull, Blanchette, Traytel and Waldmann [2] to construct infinite traces.

primrec *ltake* **where**
ltake 0 t = []
| *ltake* (*Suc i*) *t* = (*case t of LNil* ⇒ [] | *LCons x t'* ⇒ *x* # *ltake i t'*)

primrec *ldrop* **where**
ldrop 0 t = *t*
| *ldrop* (*Suc i*) *t* = (*case t of LNil* ⇒ *LNil* | *LCons x t'* ⇒ *ldrop i t'*)

lemma *ltake-LNil*[*simp*]: *ltake i LNil* = []
⟨*proof*⟩

lemma *ltake-LCons*: *0* < *i* ⟹ *ltake i* (*LCons x t*) = *x* # *ltake* (*i* − *1*) *t*
⟨*proof*⟩

lemma *take-ltake*: *i* ≤ *j* ⟹ *take i* (*ltake j xs*) = *ltake i xs*
⟨*proof*⟩

lemma *nth-ltake* [*simp*]: *i* < *min n* (*llength xs*) ⟹ (*ltake n xs*) ! *i* = *lnth xs i*
⟨*proof*⟩

lemma *length-ltake*[*simp*]: *length* (*ltake i xs*) = (*case llength xs of* ∞ ⇒ *i* | *enat m* ⇒ *min i m*)
⟨*proof*⟩

lemma *ltake-prepend*:
ltake i (*prepend xs t*) = (*if i* ≤ *length xs then take i xs else xs* @ *ltake* (*i* − *length xs*) *t*)
⟨*proof*⟩

lemma *prepend-ltake-ldrop-id*: *prepend* (*ltake i t*) (*ldrop i t*) = *t*
⟨*proof*⟩

context *distributed-system*
begin

coinductive *cotrace* **where**
cotr-init: *cotrace c LNil*
| *cotr-step*: ⟦ *c* ⊢ *ev* ↦ *c*′; *cotrace c*′ *t* ⟧ ⟹ *cotrace c* (*LCons ev t*)

lemma *cotrace-trace*: *cotrace c t* ⟹ ∃!*c*′. *trace c* (*ltake i t*) *c*′
⟨*proof*⟩

lemma *cotrace-trace*′: *cotrace c t* ⟹ ∃ *c*′. *trace c* (*ltake i t*) *c*′
⟨*proof*⟩

definition *cos* **where** *cos c t i* = *s c* (*ltake i t*) *i*

lemma *cotrace-trace-cos*: *cotrace c t* ⟹ *trace c* (*ltake i t*) (*cos c t i*)
⟨*proof*⟩

lemma *s-0*[*simp*]: *s c t 0* = *c*
⟨*proof*⟩

lemma *s-chop*: *i* ≤ *length t* ⟹ *s c t i* = *s c* (*take i t*) *i*
⟨*proof*⟩

lemma *cotrace-prepend*: *trace c t c*′ ⟹ *cotrace c*′ *u* ⟹ *cotrace c* (*prepend t u*)
⟨*proof*⟩

lemma *s-Cons*: ∃ *c*″. *trace c*′ *xs c*″ ⟹ *c* ⊢ *ev* ↦ *c*′ ⟹ *s c* (*ev* # *xs*) (*Suc i*) = *s c*′ *xs i*
⟨*proof*⟩

lemma *cotrace-ldrop*: *cotrace c t* ⟹ *i* ≤ *llength t* ⟹ *cotrace* (*cos c t i*) (*ldrop i t*)
⟨*proof*⟩

end

locale *cocomputation* = *distributed-system* +
fixes
init :: ('*a*, '*b*, '*c*) *configuration*
assumes

finite-channels:
finite $\{i.\ \exists\, p\ q.\ channel\ i = Some\ (p,\ q)\}$ **and**
strongly-connected-raw:
$\forall\, p\ q.\ (p \neq q) \longrightarrow$
$(tranclp\ (\lambda p\ q.\ (\exists\, i.\ channel\ i = Some\ (p,\ q))))\ p\ q$ **and**

at-least-two-processes:
$card\ (UNIV :: 'a\ set) > 1$ **and**
finite-processes:
$finite\ (UNIV :: 'a\ set)$ **and**

no-initial-Marker:
$\forall\, i.\ (\exists\, p\ q.\ channel\ i = Some\ (p,\ q))$
$\longrightarrow Marker \notin set\ (msgs\ init\ i)$ **and**
no-msgs-if-no-channel:
$\forall\, i.\ channel\ i = None \longrightarrow msgs\ init\ i = []$ **and**
no-initial-process-snapshot:
$\forall\, p.\ \neg\ has\text{-}snapshotted\ init\ p$ **and**
no-initial-channel-snapshot:
$\forall\, i.\ channel\text{-}snapshot\ init\ i = ([],\ NotStarted)$ **and**

valid: $\exists\, t.\ cotrace\ init\ t$ **and**
l1: $\forall\, t\ i\ cid.\ cotrace\ init\ t$
$\wedge\ Marker \in set\ (msgs\ (cos\ init\ t\ i)\ cid)$
$\longrightarrow (\exists j \leq llength\ t.\ j \geq i \wedge Marker \notin set\ (msgs\ (cos\ init\ t\ j)\ cid))$ **and**
l2: $\forall\, t\ p.\ cotrace\ init\ t$
$\longrightarrow (\exists\, i \leq llength\ t.\ has\text{-}snapshotted\ (cos\ init\ t\ i)\ p)$
begin

abbreviation *coS* **where** $coS \equiv cos\ init$

definition $some\text{-}snapshot\ t\ p = (SOME\ i.\ has\text{-}snapshotted\ (coS\ t\ i)\ p \wedge i \leq llength\ t)$

lemma *has-snapshotted*:
$cotrace\ init\ t \Longrightarrow has\text{-}snapshotted\ (coS\ t\ (some\text{-}snapshot\ t\ p))\ p \wedge some\text{-}snapshot\ t\ p \leq llength\ t$
⟨*proof*⟩

lemma *cotrace-cos*: $cotrace\ init\ t \Longrightarrow j < llength\ t \Longrightarrow$
$(coS\ t\ j) \vdash lnth\ t\ j \mapsto (coS\ t\ (Suc\ j))$
⟨*proof*⟩

lemma *snapshot-stable*:
$cotrace\ init\ t \Longrightarrow i \leq j \Longrightarrow has\text{-}snapshotted\ (coS\ t\ i)\ p \Longrightarrow has\text{-}snapshotted\ (coS\ t\ j)\ p$
⟨*proof*⟩

lemma *no-markers-if-all-snapshotted*:

cotrace init t $\Longrightarrow$ $i \leq j \Longrightarrow \forall\, p.$ *has-snapshotted* (*coS t i*) *p* $\Longrightarrow$
Marker $\notin$ *set* (*msgs* (*coS t i*) *c*) $\Longrightarrow$ *Marker* $\notin$ *set* (*msgs* (*coS t j*) *c*)
⟨*proof*⟩

lemma *cotrace-all-have-snapshotted*:
assumes *cotrace init t*
shows $\exists\, i \leq$ *llength t*. $\forall\, p.$ *has-snapshotted* (*coS t i*) *p*
⟨*proof*⟩

lemma *no-messages-if-no-channel*:
assumes *cotrace init t*
shows *channel cid* = *None* $\Longrightarrow$ *msgs* (*coS t i*) *cid* = []
⟨*proof*⟩

lemma *cotrace-all-have-snapshotted-and-no-markers*:
assumes *cotrace init t*
shows $\exists\, i \leq$ *llength t*. ($\forall\, p.$ *has-snapshotted* (*coS t i*) *p*) $\wedge$
($\forall\, c.$ *Marker* $\notin$ *set* (*msgs* (*coS t i*) *c*))
⟨*proof*⟩

context
fixes *t*
assumes *cotrace*: *cotrace init t*
begin

definition *final-i* $\equiv$
(*SOME i*. $i \leq$ *llength t* $\wedge$ ($\forall\, p.$ *has-snapshotted* (*coS t i*) *p*) $\wedge$ ($\forall\, c.$ *Marker* $\notin$ *set* (*msgs* (*coS t i*) *c*)))

definition *final* **where**
final = *coS t final-i*

lemma *final-i*: *final-i* $\leq$ *llength t* ($\forall\, p.$ *has-snapshotted* (*coS t final-i*) *p*) ($\forall\, c.$ *Marker* $\notin$ *set* (*msgs* (*coS t final-i*) *c*))
⟨*proof*⟩

lemma *final*: $\exists\, t.$ *trace init t final* ($\forall\, p.$ *has-snapshotted final p*) ($\forall\, c.$ *Marker* $\notin$ *set* (*msgs final c*))
⟨*proof*⟩

interpretation *computation channel trans send recv init final*
⟨*proof*⟩

definition *coperm* **where**
coperm l r = ($\exists\, xs\ ys\ z.$ *mset xs* = *mset ys* $\wedge$ *l* = *prepend xs z* $\wedge$ *r* = *prepend ys z*)

lemma *copermIL*: *mset ys* = *mset xs* $\Longrightarrow$ *t* = *prepend xs z* $\Longrightarrow$ *coperm* (*prepend ys z*) *t*

⟨*proof*⟩

lemma *snapshot-algorithm-is-cocorrect*:
∃ *t′ i*. *cotrace init t′* ∧ *coperm t′ t* ∧ *state-equal-to-snapshot* (*coS t′ i*) *final* ∧ *i* ≤ *final-i*
⟨*proof*⟩

end

print-statement *snapshot-algorithm-is-cocorrect*

end

end

7 Example

We provide an example in order to prove that our locale is non-vacuous. This example corresponds to the computation and associated snapshot described in Section 4 of [1].

theory *Example*
imports
Snapshot

begin

datatype *PType* = *P* | *Q*
datatype *MType* = *M* | *M′*
datatype *SType* = *S-Wait* | *S-Send* | *T-Wait* | *T-Send*

fun *trans* :: *PType* ⇒ *SType* ⇒ *SType* ⇒ *bool* **where**
trans p s s′ = *False*

fun *send* :: *channel-id* ⇒ *PType* ⇒ *PType* ⇒ *SType*
⇒ *SType* ⇒ *MType* ⇒ *bool* **where**
send c p q s s′ m = ((*c* = *0* ∧ *p* = *P* ∧ *q* = *Q*
∧ *s* = *S-Send* ∧ *s′* = *S-Wait* ∧ *m* = *M*)
∨ (*c* = *1* ∧ *p* = *Q* ∧ *q* = *P*
∧ *s* = *T-Send* ∧ *s′* = *T-Wait* ∧ *m* = *M′*))

fun *recv* :: *channel-id* ⇒ *PType* ⇒ *PType* ⇒ *SType*
⇒ *SType* ⇒ *MType* ⇒ *bool* **where**
recv c p q s s′ m = ((*c* = *1* ∧ *p* = *P* ∧ *q* = *Q*
∧ *s* = *S-Wait* ∧ *s′* = *S-Send* ∧ *m* = *M′*)
∨ (*c* = *0* ∧ *p* = *Q* ∧ *q* = *P*
∧ *s* = *T-Wait* ∧ *s′* = *T-Send* ∧ *m* = *M*))

fun *chan* :: *nat* ⇒ (*PType* ∗ *PType*) *option* **where**

```
 chan n = (if n = 0 then Some (P, Q)
          else if n = 1 then Some (Q, P)
          else None)

abbreviation init :: (PType, SType, MType) configuration where
  init ≡ (|
      states = (%p. if p = P then S-Send else T-Send),
      msgs = (%d. []),
      process-snapshot = (%p. None),
      channel-snapshot = (%d. ([], NotStarted))
    |)

abbreviation t0 where t0 ≡ Snapshot P

abbreviation s1 :: (PType, SType, MType) configuration where
  s1 ≡ (|
      states = (%p. if p = P then S-Send else T-Send),
      msgs = (%d. if d = 0 then [Marker] else []),
      process-snapshot = (%p. if p = P then Some S-Send else None),
      channel-snapshot = (%d. if d = 1 then ([], Recording) else ([], NotStarted))
    |)

abbreviation t1 where t1 ≡ Send 0 P Q S-Send S-Wait M

abbreviation s2 :: (PType, SType, MType) configuration where
  s2 ≡ (|
      states = (%p. if p = P then S-Wait else T-Send),
      msgs = (%d. if d = 0 then [Marker, Msg M] else []),
      process-snapshot = (%p. if p = P then Some S-Send else None),
      channel-snapshot = (%d. if d = 1 then ([], Recording) else ([], NotStarted))
    |)

abbreviation t2 where t2 ≡ Send 1 Q P T-Send T-Wait M'

abbreviation s3 :: (PType, SType, MType) configuration where
  s3 ≡ (|
      states = (%p. if p = P then S-Wait else T-Wait),
     msgs = (%d. if d = 0 then [Marker, Msg M] else if d = 1 then [Msg M'] else
[]),
      process-snapshot = (%p. if p = P then Some S-Send else None),
      channel-snapshot = (%d. if d = 1 then ([], Recording) else ([], NotStarted))
    |)

abbreviation t3 where t3 ≡ Snapshot Q

abbreviation s4 :: (PType, SType, MType) configuration where
  s4 ≡ (|
      states = (%p. if p = P then S-Wait else T-Wait),
       msgs = (%d. if d = 0 then [Marker, Msg M] else if d = 1 then [Msg M',
```

Marker] *else* []),
process-snapshot = (%*p*. *if p* = *P then Some S-Send else Some T-Wait*),
channel-snapshot = (%*d*. *if d* = *1 then* ([], *Recording*) *else if d* = *0 then* ([], *Recording*) *else* ([], *NotStarted*))
⦈

abbreviation *t4* **where** *t4* ≡ *RecvMarker 0 Q P*

abbreviation *s5* :: (*PType*, *SType*, *MType*) *configuration* **where**
s5 ≡ ⦇
states = (%*p*. *if p* = *P then S-Wait else T-Wait*),
msgs = (%*d*. *if d* = *0 then* [*Msg M*] *else if d* = *1 then* [*Msg M′*, *Marker*] *else* []),
process-snapshot = (%*p*. *if p* = *P then Some S-Send else Some T-Wait*),
channel-snapshot = (%*d*. *if d* = *0 then* ([], *Done*) *else if d* = *1 then* ([], *Recording*) *else* ([], *NotStarted*))
⦈

abbreviation *t5* **where** *t5* ≡ *Recv 1 P Q S-Wait S-Send M′*

abbreviation *s6* :: (*PType*, *SType*, *MType*) *configuration* **where**
s6 ≡ ⦇
states = (%*p*. *if p* = *P then S-Send else T-Wait*),
msgs = (%*d*. *if d* = *0 then* [*Msg M*] *else if d* = *1 then* [*Marker*] *else* []),
process-snapshot = (%*p*. *if p* = *P then Some S-Send else Some T-Wait*),
channel-snapshot = (%*d*. *if d* = *0 then* ([], *Done*) *else if d* = *1 then* ([*M′*], *Recording*) *else* ([], *NotStarted*))
⦈

abbreviation *t6* **where** *t6* ≡ *RecvMarker 1 P Q*

abbreviation *s7* :: (*PType*, *SType*, *MType*) *configuration* **where**
s7 ≡ ⦇
states = (%*p*. *if p* = *P then S-Send else T-Wait*),
msgs = (%*d*. *if d* = *0 then* [*Msg M*] *else if d* = *1 then* [] *else* []),
process-snapshot = (%*p*. *if p* = *P then Some S-Send else Some T-Wait*),
channel-snapshot = (%*d*. *if d* = *0 then* ([], *Done*) *else if d* = *1 then* ([*M′*], *Done*) *else* ([], *NotStarted*))
⦈

lemma *s7-no-marker*:
shows
∀ *cid*. *Marker* ∉ *set* (*msgs s7 cid*)
⟨*proof*⟩

interpretation *computation chan trans send recv init s7*
⟨*proof*⟩

end

References

[1] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[2] A. Schlichtkrull, J. C. Blanchette, D. Traytel, and U. Waldmann. Formalization of bachmair and ganzinger's ordered resolution prover. *Archive of Formal Proofs*, Jan. 2018. http://isa-afp.org/entries/Ordered_Resolution_Prover.html, Formal proof development.