

# A formal proof of the Chandy–Lamport distributed snapshot algorithm

Ben Fiedler<sup>1</sup> and Dmitriy Traytel<sup>1</sup>

<sup>1</sup>ETH Zürich

May 16, 2026

## Abstract

We provide a suitable distributed system model and implementation of 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

<b>1</b>	<b>Modelling distributed systems</b>	<b>2</b>
1.1	The distributed system locale . . . . .	4
1.1.1	State transitions . . . . .	4
<b>2</b>	<b>Traces</b>	<b>14</b>
2.1	Properties of traces . . . . .	15
2.2	Describing intermediate configurations . . . . .	16
2.3	Trace-related lemmas . . . . .	17
<b>3</b>	<b>Utilities</b>	<b>19</b>
<b>4</b>	<b>Swap lemmas</b>	<b>21</b>
<b>5</b>	<b>The Chandy–Lamport algorithm</b>	<b>31</b>
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
<b>6</b>	<b>Extension to infinite traces</b>	<b>50</b>
<b>7</b>	<b>Example</b>	<b>54</b>

## 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 its 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 synonyms for commonly used terms.

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

**abbreviation** *ps* **where**  $ps \equiv process-snapshot$

**abbreviation** *cs* **where**  $cs \equiv channel-snapshot$

**abbreviation** *no-snapshot-change* **where**

$no-snapshot-change\ c\ c' \equiv ((\forall p'. ps\ c\ p' = ps\ c'\ p') \wedge (\forall i'. cs\ c\ i' = cs\ c'\ i'))$

**abbreviation** *has-snapshot* **where**

$has-snapshot\ c\ p \equiv process-snapshot\ c\ p \neq None$

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]*:

$regular-event\ ev \equiv (isTrans\ ev \vee isSend\ ev \vee isRecv\ ev)$

**lemma** *nonregular-event*:

$\sim\ regular-event\ ev = (isSnapshot\ ev \vee isRecvMarker\ ev)$

$\langle proof \rangle$

**lemma** *event-occurs-on-unique*:

**assumes**

$p \neq q$

$occurs-on\ ev = p$

**shows**

$occurs-on\ ev \neq q$

*<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*  $\Rightarrow$  (*p* \* *p*) *option* **and**  
  *trans* :: *'p*  $\Rightarrow$  *'s*  $\Rightarrow$  *'s*  $\Rightarrow$  *bool* **and**  
  *send* :: *channel-id*  $\Rightarrow$  *'p*  $\Rightarrow$  *'p*  $\Rightarrow$  *'s*  $\Rightarrow$  *'s*  $\Rightarrow$  *'m*  $\Rightarrow$  *bool* **and**  
  *recv* :: *channel-id*  $\Rightarrow$  *'p*  $\Rightarrow$  *'p*  $\Rightarrow$  *'s*  $\Rightarrow$  *'s*  $\Rightarrow$  *'m*  $\Rightarrow$  *bool*  
**assumes**  
  *no-self-channel*:  
     $\forall i. \nexists p. \text{channel } i = \text{Some } (p, p)$   
**begin**

### 1.1.1 State transitions

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

**definition** *src* **where**  
*src i p*  $\equiv$  ( $\exists q. \text{channel } i = \text{Some } (p, q)$ )

**definition** *dest* **where**  
*dest i q*  $\equiv$  ( $\exists p. \text{channel } i = \text{Some } (p, q)$ )

**lemma** *can-occur-Recv*:

**assumes**

*can-occur (Recv i p q s s' m) c*

**shows**

*states c p = s*  $\wedge$  *channel i = Some (q, p)*  $\wedge$   $(\exists xs. \text{msgs } c \ i = \text{Msg } m \ \# \ xs)$   $\wedge$   
*recv i p q s s' m*  
 $\langle \text{proof} \rangle$

**abbreviation** *check-snapshot-occur where*

*check-snapshot-occur c c' p*  $\equiv$   
 (*can-occur (Snapshot p) c*  $\wedge$   
 (*ps c' p = Some (states c p)*)  
 $\wedge$   $(\forall p'. \text{states } c \ p' = \text{states } c' \ p')$   
 $\wedge$   $(\forall p'. (p' \neq p) \longrightarrow \text{ps } c' \ p' = \text{ps } c \ p')$   
 $\wedge$   $(\forall i. (\exists q. \text{channel } i = \text{Some } (p, q)) \longrightarrow \text{msgs } c' \ i = \text{msgs } c \ i \ @ \ [\text{Marker}])$   
 $\wedge$   $(\forall i. (\exists q. \text{channel } i = \text{Some } (q, p)) \longrightarrow \text{channel-snapshot } c' \ i = (\text{fst } (\text{channel-snapshot } c \ i), \text{Recording}))$   
 $\wedge$   $(\forall i. (\nexists q. \text{channel } i = \text{Some } (p, q)) \longrightarrow \text{msgs } c' \ i = \text{msgs } c \ i)$   
 $\wedge$   $(\forall i. (\nexists q. \text{channel } i = \text{Some } (q, p)) \longrightarrow \text{channel-snapshot } c' \ i = \text{channel-snapshot } c \ i))$

**abbreviation** *check-recv-marker-occur where*

*check-recv-marker-occur c c' i p q*  $\equiv$   
 (*can-occur (RecvMarker i p q) c*  
 $\wedge$   $(\forall r. \text{states } c \ r = \text{states } c' \ r)$   
 $\wedge$   $(\forall r. (r \neq p) \longrightarrow \text{process-snapshot } c \ r = \text{process-snapshot } c' \ r)$   
 $\wedge$   $(\text{Marker} \ \# \ \text{msgs } c' \ i = \text{msgs } c \ i)$   
 $\wedge$   $(\text{channel-snapshot } c' \ i = (\text{fst } (\text{channel-snapshot } c \ i), \text{Done}))$   
 $\wedge$   $(\text{if has-snapshotted } c \ p$   
   *then*  $(\text{process-snapshot } c \ p = \text{process-snapshot } c' \ p)$   
    $\wedge$   $(\forall i'. (i' \neq i) \longrightarrow \text{msgs } c' \ i' = \text{msgs } c \ i')$   
    $\wedge$   $(\forall i'. (i' \neq i) \longrightarrow \text{channel-snapshot } c \ i' = \text{channel-snapshot } c' \ i')$   
   *else*  $(\text{process-snapshot } c' \ p = \text{Some } (\text{states } c \ p))$   
    $\wedge$   $(\forall i'. i' \neq i \wedge (\exists r. \text{channel } i' = \text{Some } (p, r))$   
    $\longrightarrow \text{msgs } c' \ i' = \text{msgs } c \ i' \ @ \ [\text{Marker}])$   
    $\wedge$   $(\forall i'. i' \neq i \wedge (\exists r. \text{channel } i' = \text{Some } (r, p))$   
    $\longrightarrow \text{channel-snapshot } c' \ i' = (\text{fst } (\text{channel-snapshot } c \ i'), \text{Recording}))$   
    $\wedge$   $(\forall i'. i' \neq i \wedge (\nexists r. \text{channel } i' = \text{Some } (p, r))$   
    $\longrightarrow \text{msgs } c' \ i' = \text{msgs } c \ i')$   
    $\wedge$   $(\forall i'. i' \neq i \wedge (\nexists r. \text{channel } i' = \text{Some } (r, p))$   
    $\longrightarrow \text{channel-snapshot } c' \ i' = \text{channel-snapshot } c \ i'))$

**abbreviation** *check-trans-occur where*

*check-trans-occur c c' p s s'*  $\equiv$   
 (*can-occur (Trans p s s') c*  
 $\wedge$   $(\text{states } c' \ p = s')$   
 $\wedge$   $(\forall r. (r \neq p) \longrightarrow \text{states } c' \ r = \text{states } c \ r)$   
 $\wedge$   $(\forall i. \text{msgs } c' \ i = \text{msgs } c \ i)$   
 $\wedge$   $(\text{no-snapshot-change } c \ c')$

**abbreviation** *check-send-occur where*

$$\begin{aligned}
& \text{check-send-occur } c \ c' \ i \ p \ q \ s \ s' \ \text{msg} \equiv \\
& \quad (\text{can-occur } (\text{Send } i \ p \ q \ s \ s' \ \text{msg}) \ c \\
& \quad \wedge (\text{states } c' \ p = s') \\
& \quad \wedge (\forall r. (r \neq p) \longrightarrow \text{states } c' \ r = \text{states } c \ r) \\
& \quad \wedge (\text{msgs } c' \ i = \text{msgs } c \ i \ @ \ [\text{Msg } \text{msg}]) \\
& \quad \wedge (\forall i'. i \neq i' \longrightarrow \text{msgs } c' \ i' = \text{msgs } c \ i') \\
& \quad \wedge (\text{no-snapshot-change } c \ c')
\end{aligned}$$

**abbreviation** *check-recv-occur where*

$$\begin{aligned}
& \text{check-recv-occur } c \ c' \ i \ p \ q \ s \ s' \ \text{msg} \equiv \\
& \quad (\text{can-occur } (\text{Recv } i \ p \ q \ s \ s' \ \text{msg}) \ c \\
& \quad \wedge (\text{states } c \ p = s \ \wedge \ \text{states } c' \ p = s') \\
& \quad \wedge (\forall r. (r \neq p) \longrightarrow \text{states } c' \ r = \text{states } c \ r) \\
& \quad \wedge (\text{msgs } c \ i = \text{Msg } \text{msg} \ # \ \text{msgs } c' \ i) \\
& \quad \wedge (\forall i'. i \neq i' \longrightarrow \text{msgs } c' \ i' = \text{msgs } c \ i') \\
& \quad \wedge (\forall r. \text{process-snapshot } c \ r = \text{process-snapshot } c' \ r) \\
& \quad \wedge (\forall i'. i' \neq i \longrightarrow \text{channel-snapshot } c \ i' = \text{channel-snapshot } c' \ i') \\
& \quad \wedge (\text{if } \text{snd } (\text{channel-snapshot } c \ i) = \text{Recording} \\
& \quad \quad \text{then } \text{channel-snapshot } c' \ i = (\text{fst } (\text{channel-snapshot } c \ i) \ @ \ [\text{msg}], \ \text{Recording}) \\
& \quad \quad \text{else } \text{channel-snapshot } c \ i = \text{channel-snapshot } c' \ i)
\end{aligned}$$

The *next* predicate lets us express configuration transitions using events. The predicate  $\text{next}(s_1, e, s_2)$  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* ::

$$\begin{aligned}
& ('p, 's, 'm) \ \text{configuration} \\
& \Rightarrow ('p, 's, 'm) \ \text{event} \\
& \Rightarrow ('p, 's, 'm) \ \text{configuration} \\
& \Rightarrow \text{bool} \\
& (\langle - \vdash - \mapsto - \rangle \ [70, 70, 70]) \ \mathbf{where} \\
& \quad \text{next-snapshot: } c \vdash \text{Snapshot } p \mapsto c' = \\
& \quad \quad \text{check-snapshot-occur } c \ c' \ p \\
& \quad | \ \text{next-recv-marker: } c \vdash \text{RecvMarker } i \ p \ q \mapsto c' = \\
& \quad \quad \text{check-recv-marker-occur } c \ c' \ i \ p \ q \\
& \quad | \ \text{next-trans: } c \vdash \text{Trans } p \ s \ s' \mapsto c' = \\
& \quad \quad \text{check-trans-occur } c \ c' \ p \ s \ s' \\
& \quad | \ \text{next-send: } c \vdash \text{Send } i \ p \ q \ s \ s' \ \text{msg} \mapsto c' = \\
& \quad \quad \text{check-send-occur } c \ c' \ i \ p \ q \ s \ s' \ \text{msg} \\
& \quad | \ \text{next-recv: } c \vdash \text{Recv } i \ p \ q \ s \ s' \ \text{msg} \mapsto c' = \\
& \quad \quad \text{check-recv-occur } c \ c' \ i \ p \ q \ s \ s' \ \text{msg}
\end{aligned}$$

Useful lemmas about state transitions

**lemma** *state-and-event-determine-next:*

**assumes**

$$\begin{aligned}
& c \vdash \text{ev} \mapsto c' \ \mathbf{and} \\
& c \vdash \text{ev} \mapsto c''
\end{aligned}$$

**shows**  
 $c' = c''$   
 $\langle proof \rangle$

**lemma** *exists-next-if-can-occur*:

**assumes**  
 $can\text{-occur } ev \ c$   
**shows**  
 $\exists c'. c \vdash ev \mapsto c'$   
 $\langle proof \rangle$

**lemma** *exists-exactly-one-following-state*:

$can\text{-occur } ev \ c \implies \exists! c'. c \vdash ev \mapsto c'$   
 $\langle proof \rangle$

**lemma** *no-state-change-if-no-event*:

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $occurs\text{-on } ev \neq p$   
**shows**  
 $states \ c \ p = states \ c' \ p \wedge process\text{-snapshot } c \ p = process\text{-snapshot } c' \ p$   
 $\langle proof \rangle$

**lemma** *no-msgs-change-if-no-channel*:

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $channel \ i = None$   
**shows**  
 $msgs \ c \ i = msgs \ c' \ i$   
 $\langle proof \rangle$

**lemma** *no-cs-change-if-no-channel*:

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $channel \ i = None$   
**shows**  
 $cs \ c \ i = cs \ c' \ i$   
 $\langle proof \rangle$

**lemma** *no-msg-change-if-no-event*:

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $isSend \ ev \implies getId \ ev \neq i$  **and**  
 $isRecv \ ev \implies getId \ ev \neq i$  **and**  
 $regular\text{-event } ev$   
**shows**  
 $msgs \ c \ i = msgs \ c' \ i$   
 $\langle proof \rangle$

**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**

$regular\text{-}event\ ev \implies ps\ c\ r = ps\ c'\ r$

$\langle proof \rangle$

**lemma** *no-state-change-if-nonregular-event:*

**assumes**

$\sim regular\text{-}event\ ev$  **and**

$c \vdash ev \mapsto c'$

**shows**

$states\ c\ p = states\ c'\ p$

$\langle proof \rangle$

**lemma** *nonregular-event-induces-snapshot:*

**assumes**

$\sim has\text{-}snapshotted\ c\ p$  **and**

$c \vdash ev \mapsto c'$  **and**

$occurs\text{-}on\ ev = p$  **and**

$\sim regular\text{-}event\ ev$

**shows**

$\sim regular\text{-}event\ ev \longrightarrow has\text{-}snapshotted\ c'\ p$

$\langle proof \rangle$

**lemma** *snapshot-state-unchanged:*

**assumes**

*step:*  $c \vdash ev \mapsto c'$  **and**

$has\text{-}snapshotted\ c\ p$

**shows**

$ps\ c\ p = ps\ c'\ p$

$\langle proof \rangle$

**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**

$(\exists p\ q. ev = RecvMarker\ i\ p\ q \wedge m = Marker)$

$\vee (\exists p\ q\ s\ s'\ m'. ev = Recv\ i\ p\ q\ s\ s'\ m' \wedge m = Msg\ m')$

$\langle proof \rangle$

**lemma** *message-must-be-delivered-2:*

**assumes**

$c \vdash ev \mapsto c'$

$m : set\ (msgs\ c\ i)$

$m \notin set\ (msgs\ c'\ i)$

**shows**

$(\exists p q. ev = RecvMarker\ i\ p\ q \wedge m = Marker) \vee (\exists p q s s' m'. ev = Recv\ i\ p\ q\ s\ s'\ m' \wedge m = Msg\ m')$   
 $\langle proof \rangle$

**lemma** *recv-marker-means-snapshotted-1*:

**assumes**

$ev = RecvMarker\ i\ p\ q$  **and**  
 $c \vdash ev \mapsto c'$

**shows**

$has-snapshotted\ c'\ p$   
 $\langle proof \rangle$

**lemma** *recv-marker-means-snapshotted-2*:

**fixes**

$c\ c' :: ('p, 's, 'm)$  configuration **and**  
 $ev :: ('p, 's, 'm)$  event **and**  
 $i :: channel-id$

**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**

$has-snapshotted\ c'\ p$   
 $\langle proof \rangle$

**lemma** *event-stays-valid-if-no-occurrence*:

**assumes**

$c \vdash ev \mapsto c'$  **and**  
 $occurs-on\ ev \neq occurs-on\ ev'$  **and**  
 $can-occur\ ev'\ c$

**shows**

$can-occur\ ev'\ c'$   
 $\langle proof \rangle$

**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**

$msgs\ c\ i' = msgs\ c'\ i'$   
 $\langle proof \rangle$

**lemma** *msgs-unchanged-if-snapshotted-RecvMarker-for-other-is*:

**assumes**

$c \vdash ev \mapsto c'$  **and**

$ev = \text{RecvMarker } i \ p \ q$  **and**  
 $\text{has-snapshotted } c \ p$  **and**  
 $i' \neq i$   
**shows**  
 $\text{msgs } c \ i' = \text{msgs } c' \ i'$   
 <proof>

**lemma** *event-can-go-back-if-no-sender:*

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $\text{occurs-on } ev \neq \text{occurs-on } ev'$  **and**  
 $\text{can-occur } ev' \ c'$  **and**  
 $\sim \text{isRecvMarker } ev'$  **and**  
 $\sim \text{isSend } ev$   
**shows**  
 $\text{can-occur } ev' \ c$   
 <proof>

**lemma** *nonregular-event-can-go-back-if-in-distinct-processes:*

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $\text{regular-event } ev$  **and**  
 $\sim \text{regular-event } ev'$  **and**  
 $\text{can-occur } ev' \ c'$  **and**  
 $\text{occurs-on } ev \neq \text{occurs-on } ev'$   
**shows**  
 $\text{can-occur } ev' \ c$   
 <proof>

**lemma** *same-state-implies-same-result-state:*

**assumes**  
 $\text{states } c \ p = \text{states } d \ p$   
 $c \vdash ev \mapsto c'$  **and**  
 $d \vdash ev \mapsto d'$   
**shows**  
 $\text{states } d' \ p = \text{states } c' \ p$   
 <proof>

**lemma** *same-snapshot-state-implies-same-result-snapshot-state:*

**assumes**  
 $\text{ps } c \ p = \text{ps } d \ p$  **and**  
 $\text{states } c \ p = \text{states } d \ p$  **and**  
 $c \vdash ev \mapsto c'$  **and**  
 $d \vdash ev \mapsto d'$   
**shows**  
 $\text{ps } d' \ p = \text{ps } c' \ p$   
 <proof>

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

**assumes**  
 $msgs\ c\ i = msgs\ d\ i$   
 $c \vdash ev \mapsto c'$  **and**  
 $d \vdash ev \mapsto d'$  **and**  
*regular-event*  $ev$   
**shows**  
 $msgs\ c'\ i = msgs\ d'\ i$   
⟨*proof*⟩

**lemma** *Trans-msg*:  
**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
*isTrans*  $ev$   
**shows**  
 $msgs\ c\ i = msgs\ c'\ i$   
⟨*proof*⟩

**lemma** *new-msg-in-set-implies-occurrence*:  
**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $m \notin set\ (msgs\ c\ i)$  **and**  
 $m \in set\ (msgs\ c'\ i)$  **and**  
 $channel\ i = Some\ (p, q)$   
**shows**  
*occurs-on*  $ev = p$  (**is**  $?P$ )  
⟨*proof*⟩

**lemma** *new-Marker-in-set-implies-nonregular-occurrence*:  
**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $Marker \notin set\ (msgs\ c\ i)$  **and**  
 $Marker \in set\ (msgs\ c'\ i)$  **and**  
 $channel\ i = Some\ (p, q)$   
**shows**  
 $\sim$  *regular-event*  $ev$  (**is**  $?P$ )  
⟨*proof*⟩

**lemma** *RecvMarker-implies-Marker-in-set*:  
**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $ev = RecvMarker\ cid\ p\ q$   
**shows**  
 $Marker \in set\ (msgs\ c\ cid)$   
⟨*proof*⟩

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

$channel\ cid = Some\ (p, q)$  **and**  
 $can\ occur\ ev\ c$   
**shows**  
 $ev = RecvMarker\ cid\ q\ p$   
 $\langle proof \rangle$

**lemma** *Recv-given-channel:*

**assumes**  
 $isRecv\ ev$  **and**  
 $getId\ ev = cid$  **and**  
 $channel\ cid = Some\ (p, q)$  **and**  
 $can\ occur\ ev\ c$   
**shows**  
 $\exists s\ s'\ m. ev = Recv\ cid\ q\ p\ s\ s'\ m$   
 $\langle proof \rangle$

**lemma** *same-cs-if-not-recv:*

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $\sim isRecv\ ev$   
**shows**  
 $fst\ (cs\ c\ cid) = fst\ (cs\ c'\ cid)$   
 $\langle proof \rangle$

**lemma** *done-only-from-recv-marker:*

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $channel\ cid = Some\ (p, q)$  **and**  
 $snd\ (cs\ c\ cid) \neq Done$  **and**  
 $snd\ (cs\ c'\ cid) = Done$   
**shows**  
 $ev = RecvMarker\ cid\ q\ p$   
 $\langle proof \rangle$

**lemma** *cs-not-not-started-stable:*

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $snd\ (cs\ c\ cid) \neq NotStarted$  **and**  
 $channel\ cid = Some\ (p, q)$   
**shows**  
 $snd\ (cs\ c'\ cid) \neq NotStarted$   
 $\langle proof \rangle$

**lemma** *fst-cs-changed-by-recv-recording:*

**assumes**  
 $step: c \vdash ev \mapsto c'$  **and**  
 $fst\ (cs\ c\ cid) \neq fst\ (cs\ c'\ cid)$  **and**  
 $channel\ cid = Some\ (p, q)$   
**shows**

$snd (cs\ c\ cid) = Recording \wedge (\exists p\ q\ u\ u'\ m. ev = Recv\ cid\ q\ p\ u\ u'\ m)$   
 <proof>

**lemma** *no-marker-and-snapshotted-implies-no-more-markers:*

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
*has-snapshotted*  $c\ p$  **and**  
 $Marker \notin set\ (msgs\ c\ cid)$  **and**  
 $channel\ cid = Some\ (p, q)$   
**shows**  
 $Marker \notin set\ (msgs\ c'\ cid)$   
 <proof>

**lemma** *same-messages-if-no-occurrence:*

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $\sim\ occurs-on\ ev = p$  **and**  
 $\sim\ occurs-on\ ev = q$  **and**  
 $channel\ cid = Some\ (p, q)$   
**shows**  
 $msgs\ c\ cid = msgs\ c'\ cid \wedge cs\ c\ cid = cs\ c'\ cid$   
 <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*:  $\llbracket c \vdash ev \mapsto c'; trace\ c'\ t\ c'' \rrbracket$   
 $\implies trace\ c\ (ev\ \# \ t)\ c''$

## 2.1 Properties of traces

**lemma** *trace-trans*:

**shows**

$$\begin{aligned} & \llbracket \text{trace } c \ t \ c'; \\ & \quad \text{trace } c' \ t' \ c'' \\ & \rrbracket \Longrightarrow \text{trace } c \ (t \ @ \ t') \ c'' \end{aligned}$$

*<proof>*

**lemma** *trace-decomp-head*:

**assumes**

$$\text{trace } c \ (ev \ \# \ t) \ c'$$

**shows**

$$\exists c''. \ c \vdash \ ev \mapsto c'' \wedge \text{trace } c'' \ t \ c'$$

*<proof>*

**lemma** *trace-decomp-tail*:

**shows**

$$\text{trace } c \ (t \ @ \ [ev]) \ c' \Longrightarrow \exists c''. \ \text{trace } c \ t \ c'' \wedge c'' \vdash \ ev \mapsto c'$$

*<proof>*

**lemma** *trace-snoc*:

**assumes**

$$\text{trace } c \ t \ c' \ \mathbf{and}$$

$$c' \vdash \ ev \mapsto c''$$

**shows**

$$\text{trace } c \ (t \ @ \ [ev]) \ c''$$

*<proof>*

**lemma** *trace-rev-induct* [consumes 1, case-names *tr-rev-init tr-rev-step*]:

$$\begin{aligned} & \llbracket \text{trace } c \ t \ c'; \\ & \quad (\bigwedge c. \ P \ c \ \llbracket \ c); \\ & \quad (\bigwedge c \ t \ c' \ ev \ c''. \ \text{trace } c \ t \ c' \Longrightarrow P \ c \ t \ c' \Longrightarrow c' \vdash \ ev \mapsto c'' \Longrightarrow P \ c \ (t \ @ \ [ev]) \\ & \quad c'') \\ & \rrbracket \Longrightarrow P \ c \ t \ c' \end{aligned}$$

*<proof>*

**lemma** *trace-and-start-determines-end*:

**shows**

$$\text{trace } c \ t \ c' \Longrightarrow \text{trace } c \ t \ d' \Longrightarrow c' = d'$$

*<proof>*

**lemma** *suffix-split-trace*:

**shows**

$$\begin{aligned} & \llbracket \text{trace } c \ t \ c'; \\ & \quad \text{suffix } t' \ t \\ & \rrbracket \Longrightarrow \exists c''. \ \text{trace } c'' \ t' \ c' \end{aligned}$$

*<proof>*

**lemma** *prefix-split-trace*:

**fixes**

$c :: ('p, 's, 'm)$  configuration and

$t :: ('p, 's, 'm)$  trace

**shows**

$\llbracket \exists c'. \text{trace } c \ t \ c';$

prefix  $t' \ t$

$\rrbracket \implies \exists c''. \text{trace } c \ t' \ c''$

$\langle \text{proof} \rangle$

**lemma** *split-trace*:

**shows**

$\llbracket \text{trace } c \ t \ c';$

$t = t' \ @ \ t''$

$\rrbracket \implies \exists c''. \text{trace } c \ t' \ c'' \wedge \text{trace } c'' \ t'' \ c'$

$\langle \text{proof} \rangle$

## 2.2 Describing intermediate configurations

**definition** *construct-fun-from-rel* ::  $('a * 'b)$  set  $\Rightarrow 'a \Rightarrow 'b$  where

*construct-fun-from-rel*  $R \ x = (\text{THE } y. (x, y) \in R)$

**definition** *trace-rel* where

*trace-rel*  $\equiv \{(x, t'), y). \text{trace } x \ t' \ y\}$

**lemma** *fun-must-admit-trace*:

**shows**

*single-valued*  $R \implies x \in \text{Domain } R$

$\implies (x, \text{construct-fun-from-rel } R \ x) \in R$

$\langle \text{proof} \rangle$

**lemma** *single-valued-trace-rel*:

**shows**

*single-valued* *trace-rel*

$\langle \text{proof} \rangle$

**definition** *run-trace* where

*run-trace*  $\equiv \text{construct-fun-from-rel } \text{trace-rel}$

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

$s \ c \ t \ i = (\text{THE } c'. \text{trace } c \ (\text{take } i \ t) \ c')$

**lemma** *s-is-partial-execution*:

**shows**

$s \ c \ t \ i = \text{run-trace } (c, \text{take } i \ t)$

$\langle \text{proof} \rangle$

**lemma** *exists-trace-for-any-i:*

**assumes**

$\exists c'. \text{trace } c \ t \ c'$

**shows**

$\text{trace } c \ (\text{take } i \ t) \ (s \ c \ t \ i)$

$\langle \text{proof} \rangle$

**lemma** *exists-trace-for-any-i-j:*

**assumes**

$\exists c'. \text{trace } c \ t \ c'$  **and**

$i \leq j$

**shows**

$\text{trace } (s \ c \ t \ i) \ (\text{take } (j - i) \ (\text{drop } i \ t)) \ (s \ c \ t \ j)$

$\langle \text{proof} \rangle$

**lemma** *step-Suc:*

**assumes**

$i < \text{length } t$  **and**

*valid:*  $\text{trace } c \ t \ c'$

**shows**  $(s \ c \ t \ i) \vdash (t \ ! \ i) \mapsto (s \ c \ t \ (\text{Suc } i))$

$\langle \text{proof} \rangle$

## 2.3 Trace-related lemmas

**lemma** *snapshot-state-unchanged-trace:*

**assumes**

$\text{trace } c \ t \ c'$  **and**

$ps \ c \ p = \text{Some } u$

**shows**

$ps \ c' \ p = \text{Some } u$

$\langle \text{proof} \rangle$

**lemma** *no-state-change-if-only-nonregular-events:*

**shows**

$\llbracket \text{trace } c \ t \ c';$

$\nexists ev. ev \in \text{set } t \wedge \text{regular-event } ev \wedge \text{occurs-on } ev = p;$

$\text{states } c \ p = st$

$\rrbracket \implies \text{states } c' \ p = st$

$\langle \text{proof} \rangle$

**lemma** *message-must-be-delivered-2-trace:*

**assumes**

$\text{trace } c \ t \ c'$  **and**

$m : \text{set } (\text{msgs } c \ i)$  **and**

$m \notin \text{set } (\text{msgs } c' \ i)$  **and**

$\text{channel } i = \text{Some } (q, p)$

**shows**

$\exists ev \in \text{set } t. (\exists p \ q. ev = \text{RecvMarker } i \ p \ q \wedge m = \text{Marker}) \vee (\exists p \ q \ s \ s' \ m'. ev = \text{Recv } i \ q \ p \ s \ s' \ m' \wedge m = \text{Msg } m')$

$\langle \text{proof} \rangle$

**lemma** *marker-must-be-delivered-2-trace*:

**assumes**

$\text{trace } c \ t \ c'$  **and**

$\text{Marker} : \text{set } (\text{msgs } c \ i)$  **and**

$\text{Marker} \notin \text{set } (\text{msgs } c' \ i)$  **and**

$\text{channel } i = \text{Some } (p, q)$

**shows**

$\exists ev \in \text{set } t. (\exists p \ q. ev = \text{RecvMarker } i \ p \ q)$

$\langle \text{proof} \rangle$

**lemma** *snapshot-stable*:

**shows**

$\llbracket \text{trace } c \ t \ c';$

$\text{has-snapshotted } c \ p$

$\rrbracket \implies \text{has-snapshotted } c' \ p$

$\langle \text{proof} \rangle$

**lemma** *snapshot-stable-2*:

**shows**

$\text{trace } c \ t \ c' \implies \sim \text{has-snapshotted } c' \ p \implies \sim \text{has-snapshotted } c \ p$

$\langle \text{proof} \rangle$

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

**shows**

$\llbracket \text{trace } c \ t \ c';$

$\forall p. \text{has-snapshotted } c \ p;$

$\text{Marker} \notin \text{set } (\text{msgs } c \ i)$

$\rrbracket \implies \text{Marker} \notin \text{set } (\text{msgs } c' \ i)$

$\langle \text{proof} \rangle$

**lemma** *event-stays-valid-if-no-occurrence-trace*:

**shows**

$\llbracket \text{trace } c \ t \ c';$

$\text{list-all } (\lambda ev. \text{occurs-on } ev \neq \text{occurs-on } ev') \ t;$

$\text{can-occur } ev' \ c$

$\rrbracket \implies \text{can-occur } ev' \ c'$

$\langle \text{proof} \rangle$

**lemma** *event-can-go-back-if-no-sender-trace*:

**shows**

$\llbracket \text{trace } c \ t \ c';$

$\text{list-all } (\lambda ev. \text{occurs-on } ev \neq \text{occurs-on } ev') \ t;$

$\text{can-occur } ev' \ c';$

$\sim \text{isRecvMarker } ev';$

$\text{list-all } (\lambda ev. \sim \text{isSend } ev) \ t$

$\rrbracket \implies \text{can-occur } ev' \ c$

$\langle \text{proof} \rangle$

**lemma** *done-only-from-recv-marker-trace*:

**assumes**

*trace c t c'* **and**

$t \neq []$  **and**

$snd (cs\ c\ cid) \neq Done$  **and**

$snd (cs\ c'\ cid) = Done$  **and**

$channel\ cid = Some\ (p, q)$

**shows**

$RecvMarker\ cid\ q\ p \in set\ t$

*<proof>*

**lemma** *cs-not-not-started-stable-trace*:

**shows**

$\llbracket trace\ c\ t\ c';\ snd\ (cs\ c\ cid) \neq NotStarted;\ channel\ cid = Some\ (p, q) \rrbracket \implies$

$snd\ (cs\ c'\ cid) \neq NotStarted$

*<proof>*

**lemma** *no-messages-introduced-if-no-channel*:

**assumes**

*trace: trace init t final* **and**

*no-msgs-if-no-channel:  $\forall i. channel\ i = None \longrightarrow msgs\ init\ i = []$*

**shows**

$channel\ cid = None \implies msgs\ (s\ init\ t\ i)\ cid = []$

*<proof>*

**end**

**end**

### 3 Utilities

**theory** *Util*

**imports**

*Main*

*HOL-Library.Sublist*

*HOL-Library.Multiset*

**begin**

**abbreviation** *swap-events* **where**

$swap-events\ i\ j\ t \equiv take\ i\ t\ @\ [t!\ j, t!\ i]\ @\ take\ (j - (i+1))\ (drop\ (i+1)\ t)\ @\ drop\ (j+1)\ t$

**lemma** *swap-neighbors-2*:

**shows**

$i+1 < length\ t \implies swap-events\ i\ (i+1)\ t = (t[i := t!\ (i+1)])[i+1 := t!\ i]$

*<proof>*

**lemma** *swap-identical-length*:

**assumes**

$i < j$  **and**  
 $j < \text{length } t$

**shows**

$\text{length } t = \text{length } (\text{swap-events } i \ j \ t)$

*<proof>*

**lemma** *swap-identical-heads*:

**assumes**

$i < j$  **and**  
 $j < \text{length } t$

**shows**

$\text{take } i \ t = \text{take } i \ (\text{swap-events } i \ j \ t)$

*<proof>*

**lemma** *swap-identical-tails*:

**assumes**

$i < j$  **and**  
 $j < \text{length } t$

**shows**

$\text{drop } (j+1) \ t = \text{drop } (j+1) \ (\text{swap-events } i \ j \ t)$

*<proof>*

**lemma** *swap-neighbors*:

**shows**

$i+1 < \text{length } l \implies (l[i := l! (i+1)])[i+1 := l! i] = \text{take } i \ l \ @ \ [l! (i+1), l! i] \ @ \ \text{drop } (i+2) \ l$

*<proof>*

**lemma** *swap-events-perm*:

**assumes**

$i < j$  **and**  
 $j < \text{length } t$

**shows**

$\text{mset } (\text{swap-events } i \ j \ t) = \text{mset } t$

*<proof>*

**lemma** *sum-eq-if-same-subterms*:

**fixes**

$i :: \text{nat}$

**shows**

$\forall k. i \leq k \wedge k < j \implies f \ k = f' \ k \implies \text{sum } f \ \{i..<j\} = \text{sum } f' \ \{i..<j\}$

*<proof>*

**lemma** *filter-neq-takeWhile*:

**assumes**

$\text{filter } ((\neq) \ a) \ l \neq \ \text{takeWhile } ((\neq) \ a) \ l$

**shows**

$\exists i j. i < j \wedge j < \text{length } l \wedge l ! i = a \wedge l ! j \neq a$  (is ?P)  
<proof>

**lemma** *util-exactly-one-element*:

**assumes**

$m \notin \text{set } l$  **and**

$l' = l @ [m]$

**shows**

$\exists ! j. j < \text{length } l' \wedge l' ! j = m$  (is ?P)

<proof>

**lemma** *exists-one-iff-filter-one*:

**shows**

$(\exists ! j. j < \text{length } l \wedge l ! j = a) \longleftrightarrow \text{length } (\text{filter } ((=) a) l) = 1$

<proof>

**end**

## 4 Swap lemmas

**theory** *Swap*

**imports**

*Distributed-System*

**begin**

**context** *distributed-system*

**begin**

**lemma** *swap-msgs-Trans-Trans*:

**assumes**

$c \vdash ev \mapsto d$  **and**

$d \vdash ev' \mapsto e$  **and**

*isTrans*  $ev$  **and**

*isTrans*  $ev'$  **and**

$c \vdash ev' \mapsto d'$  **and**

$d' \vdash ev \mapsto e'$  **and**

*occurs-on*  $ev \neq \text{occurs-on } ev'$

**shows**

$\text{msgs } e \ i = \text{msgs } e' \ i$

<proof>

**lemma** *swap-msgs-Send-Send*:

**assumes**

$c \vdash ev \mapsto d$  **and**

$d \vdash ev' \mapsto e$  **and**

*isSend*  $ev$  **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-msgs-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**  
 $msgs\ e\ i = msgs\ e'\ i$   
 ⟨*proof*⟩

**lemma** *swap-msgs-Send-Trans:*

**assumes**  
 $c \vdash ev \mapsto d$  **and**  
 $d \vdash ev' \mapsto e$  **and**  
*isSend 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-Send:*

**assumes**  
 $c \vdash ev \mapsto d$  **and**  
 $d \vdash ev' \mapsto e$  **and**  
*isTrans 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-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**

$cs\ c'\ i = cs\ d'\ i$

$\langle proof \rangle$

**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**

$cs\ e\ i = cs\ e'\ i$

$\langle proof \rangle$

**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**

$cs\ e\ i = cs\ e'\ i$

$\langle proof \rangle$

**lemma** *same-messages-2*:

**assumes**

$\forall p. has-snapshot\ c\ p = has-snapshot\ d\ p$  **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**

$msgs\ c'\ i = msgs\ d'\ i$

$\langle proof \rangle$

**lemma** *same-cs-2*:

**assumes**

$\forall p. \text{has-snapshotted } c \ p = \text{has-snapshotted } d \ p$  **and**

$\text{cs } c \ i = \text{cs } d \ i$  **and**

$c \vdash \text{ev} \mapsto c'$  **and**

$d \vdash \text{ev} \mapsto d'$

**shows**

$\text{cs } c' \ i = \text{cs } d' \ i$

$\langle \text{proof} \rangle$

**lemma** *swap-Snapshot-Trans*:

**assumes**

$c \vdash \text{ev} \mapsto d$  **and**

$d \vdash \text{ev}' \mapsto e$  **and**

*isSnapshot*  $\text{ev}$  **and**

*isTrans*  $\text{ev}'$  **and**

$c \vdash \text{ev}' \mapsto d'$  **and**

$d' \vdash \text{ev} \mapsto e'$  **and**

$\text{occurs-on } \text{ev} \neq \text{occurs-on } \text{ev}'$

**shows**

$\text{msgs } e \ i = \text{msgs } e' \ i$

$\langle \text{proof} \rangle$

**lemma** *swap-msgs-Trans-RecvMarker*:

**assumes**

$c \vdash \text{ev} \mapsto d$  **and**

$d \vdash \text{ev}' \mapsto e$  **and**

*isRecvMarker*  $\text{ev}$  **and**

*isTrans*  $\text{ev}'$  **and**

$c \vdash \text{ev}' \mapsto d'$  **and**

$d' \vdash \text{ev} \mapsto e'$  **and**

$\text{occurs-on } \text{ev} \neq \text{occurs-on } \text{ev}'$

**shows**

$\text{msgs } e' \ i = \text{msgs } e \ i$

$\langle \text{proof} \rangle$

**lemma** *swap-Trans-Snapshot*:

**assumes**

$c \vdash \text{ev} \mapsto d$  **and**

$d \vdash \text{ev}' \mapsto e$  **and**

*isTrans*  $\text{ev}$  **and**

*isSnapshot*  $\text{ev}'$  **and**

$c \vdash \text{ev}' \mapsto d'$  **and**

$d' \vdash \text{ev} \mapsto e'$  **and**

$\text{occurs-on } \text{ev} \neq \text{occurs-on } \text{ev}'$

**shows**

$\text{msgs } e \ i = \text{msgs } e' \ i$

$\langle \text{proof} \rangle$

**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**

$c \vdash ev \mapsto d$  **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**

$msgs\ e\ i = msgs\ e'\ i$   
(proof)

**lemma** *swap-RecvMarker-Recv*:

**assumes**

$c \vdash ev \mapsto d$  **and**  
 $d \vdash ev' \mapsto e$  **and**  
*isRecvMarker*  $ev$  **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-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**  
 $d' \vdash ev \mapsto e'$  **and**  
*occurs-on*  $ev \neq occurs-on\ ev'$

**shows**

$msgs\ e\ i = msgs\ e'\ i$   
(proof)

**lemma** *swap-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**

$d' \vdash ev \mapsto e'$  **and**  
 $occurs-on\ ev \neq occurs-on\ ev'$   
**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 configurations 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.  $\exists p q. \text{channel } i = \text{Some } (p, q)}$  and
  strongly-connected-raw:
     $\forall p q. (p \neq q) \longrightarrow$ 
       $(\text{tranclp } (\lambda p q. (\exists i. \text{channel } i = \text{Some } (p, q)))) p q$  and

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

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

$\forall p. \sim \text{has-snapshotted init } p$  **and**  
*no-initial-channel-snapshot:*  
 $\forall i. \text{channel-snapshot init } i = ([], \text{NotStarted})$  **and**  
  
*valid:*  $\exists t. \text{trace init } t \text{ final}$  **and**  
*l1:*  $\forall t \ i \ cid. \text{trace init } t \text{ final}$   
 $\wedge \text{Marker} \in \text{set } (\text{msgs } (s \text{ init } t \ i) \ cid)$   
 $\longrightarrow (\exists j. j \geq i \wedge \text{Marker} \notin \text{set } (\text{msgs } (s \text{ init } t \ j) \ cid))$  **and**  
*l2:*  $\forall t \ p. \text{trace init } t \text{ final}$   
 $\longrightarrow (\exists i. \text{has-snapshotted } (s \text{ init } t \ i) \ p \wedge i \leq \text{length } t)$   
**begin**

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

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

**lemma** *exists-some-channel:*  
**shows**  $\exists i \ p \ q. \text{channel } i = \text{Some } (p, q)$   
 $\langle \text{proof} \rangle$

**abbreviation** *S* **where**  
 $S \equiv s \text{ init}$

**lemma** *no-messages-if-no-channel:*  
**assumes** *trace init t final*  
**shows**  $\text{channel } cid = \text{None} \implies \text{msgs } (s \text{ init } t \ i) \ cid = []$   
 $\langle \text{proof} \rangle$

**lemma** *S-induct* [*consumes 3, case-names S-init S-step*]:  
 $\llbracket \text{trace init } t \text{ final}; i \leq j; j \leq \text{length } t;$   
 $\bigwedge i. P \ i \ i;$   
 $\bigwedge i \ j. i < j \implies j \leq \text{length } t \implies (S \ t \ i) \vdash (t \ ! \ i) \mapsto (S \ t \ (\text{Suc } i)) \implies P \ (\text{Suc } i)$   
 $\bigwedge j \implies P \ i \ j$   
 $\rrbracket \implies P \ i \ j$   
 $\langle \text{proof} \rangle$

**lemma** *exists-index:*  
**assumes**  
*trace init t final and*  
 $ev \in \text{set } (\text{take } (j - i) \ (\text{drop } i \ t))$   
**shows**  
 $\exists k. i \leq k \wedge k < j \wedge ev = t \ ! \ k$   
 $\langle \text{proof} \rangle$

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

$j \geq i$   
**shows**  
 $S\ t\ i = S\ t\ j$   
 <proof>

**lemma** *no-marker-if-no-snapshot*:  
**shows**  
 $\llbracket \text{trace } \textit{init}\ t\ \textit{final}; \text{channel } \textit{cid} = \textit{Some}\ (p, q);$   
 $\sim \textit{has-snapshotted}\ (S\ t\ i)\ p \rrbracket$   
 $\implies \textit{Marker} \notin \textit{set}\ (\textit{msgs}\ (S\ t\ i)\ \textit{cid})$   
 <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**  
 $\textit{trace}\ \textit{init}\ t\ \textit{final}$   
**shows**  
 $\exists p\ i. \textit{Snapshot}\ p = t!\ i$   
 <proof>

**lemma** *recv-marker-means-snapshotted*:  
**assumes**  
 $\textit{trace}\ \textit{init}\ t\ \textit{final}$  **and**  
 $\textit{ev} = \textit{RecvMarker}\ \textit{cid}\ p\ q$  **and**  
 $(S\ t\ i) \vdash \textit{ev} \mapsto (S\ t\ (\textit{Suc}\ i))$   
**shows**  
 $\textit{has-snapshotted}\ (S\ t\ i)\ q$   
 <proof>

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

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

$\text{Marker} : \text{set} (\text{msgs} (S t (\text{Suc } i)) \text{ cid}) \vee \text{has-snapshotted} (S t i) q$   
 $\langle \text{proof} \rangle$

**lemma** *exists-snapshot-for-all-p*:

**assumes**

$\text{trace init } t \text{ final}$

**shows**

$\exists i. \sim \text{has-snapshotted} (S t i) p \wedge \text{has-snapshotted} (S t (\text{Suc } i)) p$  (**is ?Q**)

$\langle \text{proof} \rangle$

**lemma** *all-processes-snapshotted-in-final-state*:

**assumes**

$\text{trace init } t \text{ final}$

**shows**

$\text{has-snapshotted final } p$

$\langle \text{proof} \rangle$

**definition** *next-marker-free-state* **where**

$\text{next-marker-free-state } t i \text{ cid} = (\text{LEAST } j. j \geq i \wedge \text{Marker} \notin \text{set} (\text{msgs} (S t j) \text{ cid}))$

**lemma** *exists-next-marker-free-state*:

**assumes**

$\text{channel cid} = \text{Some} (p, q)$

$\text{trace init } t \text{ final}$

**shows**

$\exists !j. \text{next-marker-free-state } t i \text{ cid} = j \wedge j \geq i \wedge \text{Marker} \notin \text{set} (\text{msgs} (S t j) \text{ cid})$

$\langle \text{proof} \rangle$

**theorem** *snapshot-algorithm-must-terminate*:

**assumes**

$\text{trace init } t \text{ final}$

**shows**

$\exists \text{phi}. ((\forall p. \text{has-snapshotted} (S t \text{phi}) p) \wedge (\forall \text{cid}. \text{Marker} \notin \text{set} (\text{msgs} (S t \text{phi}) \text{cid})))$

$\langle \text{proof} \rangle$

### 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**  $\text{trace init } t \text{ final} \implies \text{has-snapshotted} (S t i) p \implies j \geq i \implies \text{has-snapshotted} (S t j) p$

$\langle \text{proof} \rangle$

**lemma** *snapshot-stable-ver-3*:

**shows**  $\text{trace init } t \text{ final} \implies \sim \text{has-snapshotted } (S \ t \ i) \ p \implies i \geq j \implies \sim$   
 $\text{has-snapshotted } (S \ t \ j) \ p$   
 $\langle \text{proof} \rangle$

**lemma** *marker-must-stay-if-no-snapshot*:

**assumes**

$\text{trace init } t \text{ final}$  **and**  
 $\text{has-snapshotted } (S \ t \ i) \ p$  **and**  
 $\sim \text{has-snapshotted } (S \ t \ i) \ q$  **and**  
 $\text{channel } \text{cid} = \text{Some } (p, q)$

**shows**

$\text{Marker} : \text{set } (\text{msgs } (S \ t \ i) \ \text{cid})$

$\langle \text{proof} \rangle$

### 5.3.1 Pre- and postrecording events

**definition** *prerecording-event*:

$\text{prerecording-event } t \ i \equiv$   
 $i < \text{length } t \wedge \text{regular-event } (t \ ! \ i)$   
 $\wedge \sim \text{has-snapshotted } (S \ t \ i) \ (\text{occurs-on } (t \ ! \ i))$

**definition** *postrecording-event*:

$\text{postrecording-event } t \ i \equiv$   
 $i < \text{length } t \wedge \text{regular-event } (t \ ! \ i)$   
 $\wedge \text{has-snapshotted } (S \ t \ i) \ (\text{occurs-on } (t \ ! \ i))$

**abbreviation** *neighboring where*

$\text{neighboring } t \ i \ j \equiv i < j \wedge j < \text{length } t \wedge \text{regular-event } (t \ ! \ i) \wedge \text{regular-event } (t$   
 $\ ! \ j)$   
 $\wedge (\forall k. i < k \wedge k < j \longrightarrow \sim \text{regular-event } (t \ ! \ k))$

**lemma** *pre-if-regular-and-not-post*:

**assumes**

$\text{regular-event } (t \ ! \ i)$  **and**  
 $\sim \text{postrecording-event } t \ i$  **and**  
 $i < \text{length } t$

**shows**

$\text{prerecording-event } t \ i$

$\langle \text{proof} \rangle$

**lemma** *post-if-regular-and-not-pre*:

**assumes**

$\text{regular-event } (t \ ! \ i)$  **and**  
 $\sim \text{prerecording-event } t \ i$  **and**  
 $i < \text{length } t$

**shows**

*postrecording-event t i*  
 ⟨proof⟩

**lemma** *post-before-pre-different-processes:*

**assumes**

*i < j and*  
*j < length t and*  
*neighboring:  $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t ! k)$  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)  $\neq$  occurs-on (t ! j)*  
 ⟨proof⟩

**lemma** *post-before-pre-neighbors:*

**assumes**

*i < j and*  
*j < length t and*  
*neighboring:  $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t ! k)$  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.  $\sim \text{regular-event } ev \wedge \sim$*   
*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:  $\forall k. (i < k \wedge k < j)$*   
 *$\longrightarrow \sim \text{regular-event } (t ! k)$  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**  $\llbracket i < j; j < \text{length } t;$   
 $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t ! k);$   
*postrecording-event t i; prerecording-event t j;*

$$\begin{aligned}
& \text{trace init } t \text{ final } \mathbb{I} \\
\implies & \text{trace init } (\text{swap-events } i \ j \ t) \text{ final} \\
& \wedge (\forall k. k \geq j + 1 \longrightarrow S (\text{swap-events } i \ j \ t) \ k = S \ t \ k) \\
& \wedge (\forall k. k \leq i \longrightarrow S (\text{swap-events } i \ j \ t) \ k = S \ t \ k) \\
& \wedge \text{prerecording-event } (\text{swap-events } i \ j \ t) \ i \\
& \wedge \text{postrecording-event } (\text{swap-events } i \ j \ t) \ (i+1) \\
& \wedge (\forall k. k > i+1 \wedge k < j+1 \\
& \quad \longrightarrow \sim \text{regular-event } ((\text{swap-events } i \ j \ t) ! \ k)) \\
\langle \text{proof} \rangle
\end{aligned}$$

### 5.3.3 Relating configurations and the computed snapshot

**definition** *ps-equal-to-snapshot* where

$$\begin{aligned}
\text{ps-equal-to-snapshot } c \ c' & \equiv \\
& \forall p. \text{Some } (\text{states } c \ p) = \text{process-snapshot } c' \ p
\end{aligned}$$

**definition** *cs-equal-to-snapshot* where

$$\begin{aligned}
\text{cs-equal-to-snapshot } c \ c' & \equiv \\
& \forall \text{cid}. \text{channel } \text{cid} \neq \text{None} \\
& \longrightarrow \text{filter } ((\neq) \text{Marker}) (\text{msgs } c \ \text{cid}) \\
& = \text{map } \text{Msg } (\text{fst } (\text{channel-snapshot } c' \ \text{cid}))
\end{aligned}$$

**definition** *state-equal-to-snapshot* where

$$\begin{aligned}
\text{state-equal-to-snapshot } c \ c' & \equiv \\
& \text{ps-equal-to-snapshot } c \ c' \wedge \text{cs-equal-to-snapshot } c \ c'
\end{aligned}$$

**lemma** *init-is-s-t-0*:

**assumes**

$$\text{trace init } t \text{ final}$$

**shows**

$$\text{init} = (S \ t \ 0)$$

$\langle \text{proof} \rangle$

**lemma** *final-is-s-t-len-t*:

**assumes**

$$\text{trace init } t \text{ final}$$

**shows**

$$\text{final} = S \ t \ (\text{length } t)$$

$\langle \text{proof} \rangle$

**lemma** *snapshot-event*:

**assumes**

$$\text{trace init } t \text{ final} \ \mathbf{and}$$

$$\sim \text{has-snapshotted } (S \ t \ i) \ p \ \mathbf{and}$$

$$\text{has-snapshotted } (S \ t \ (i+1)) \ p$$

**shows**

$$\text{isSnapshot } (t ! i) \vee \text{isRecvMarker } (t ! i)$$

$\langle \text{proof} \rangle$

**lemma** *snapshot-state*:

**assumes**

*trace init t final and*  
*states (S t i) p = u and*  
*~ has-snapshot (S t i) p and*  
*has-snapshot (S t (i+1)) p*

**shows**

*ps (S t (i+1)) p = Some u*

*<proof>*

**lemma** *snapshot-state-unchanged-trace-2*:

**shows**

$\llbracket$  *trace init t final; i ≤ j; j ≤ length t;*  
*ps (S t i) p = Some u*  
 $\rrbracket \implies$  *ps (S t j) p = Some u*

*<proof>*

**lemma** *no-recording-cs-if-not-snapshot*:

**shows**

$\llbracket$  *trace init t final; ~ has-snapshot (S t i) p;*  
*channel cid = Some (q, p) \rrbracket \implies* *cs (S t i) cid = cs init cid*

*<proof>*

**lemma** *cs-done-implies-has-snapshot*:

**assumes**

*trace init t final and*  
*snd (cs (S t i) cid) = Done and*  
*channel cid = Some (p, q)*

**shows**

*has-snapshot (S t i) q*

*<proof>*

**lemma** *exactly-one-snapshot*:

**assumes**

*trace init t final*

**shows**

$\exists! i. \sim$  *has-snapshot (S t i) p*  $\wedge$  *has-snapshot (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**

$\sim$  *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**  
 $\llbracket \text{trace init } t \text{ final}; \text{ channel } cid = \text{Some } (p, q); i < \text{length } t; \sim \text{has-snapshotted } (S \ t \ i) \ q \rrbracket$   
 $\implies cs \ (S \ t \ i) \ cid = (\llbracket, \text{NotStarted}\rrbracket)$   
 ⟨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**  
 $\exists k. i \leq k \wedge k < j \wedge (\exists p \ q \ u \ u' \ m. t ! k = \text{Recv } cid \ q \ p \ u \ u' \ m) \wedge (\text{snd } (cs \ (S \ t \ k) \ cid) = \text{Recording}) \text{ (is ?P)}$   
 ⟨proof⟩

**lemma** *cs-not-nil-implies-postrecording-event:*

**assumes**  
*trace init t final and*  
*fst (cs (S t i) cid) ≠  $\llbracket$  and*  
*i ≤ length t and*  
*channel cid = Some (p, q)*

**shows**  
 $\exists j. j < i \wedge \text{postrecording-event } t \ j$   
 $\langle \text{proof} \rangle$

### 5.3.4 Relating process states

**lemma** *snapshot-state-must-have-been-reached*:

**assumes**  
 $\text{trace init } t \ \text{final}$  **and**  
 $\text{ps final } p = \text{Some } u$  **and**  
 $\sim \text{has-snapshotted } (S \ t \ i) \ p$  **and**  
 $\text{has-snapshotted } (S \ t \ (i+1)) \ p$  **and**  
 $i < \text{length } t$   
**shows**  
 $\text{states } (S \ t \ i) \ p = u$   
 $\langle \text{proof} \rangle$

**lemma** *ps-after-all-prerecording-events*:

**assumes**  
 $\text{trace init } t \ \text{final}$  **and**  
 $\forall i'. i' \geq i \longrightarrow \sim \text{prerecording-event } t \ i'$  **and**  
 $\forall j'. j' < i \longrightarrow \sim \text{postrecording-event } t \ j'$   
**shows**  
 $\text{ps-equal-to-snapshot } (S \ t \ i) \ \text{final}$   
 $\langle \text{proof} \rangle$

### 5.3.5 Relating channel states

**lemma** *cs-when-recording*:

**shows**  
 $\llbracket i < j; j \leq \text{length } t; \text{trace init } t \ \text{final};$   
 $\text{has-snapshotted } (S \ t \ i) \ p;$   
 $\text{snd } (cs \ (S \ t \ i) \ cid) = \text{Recording};$   
 $\text{snd } (cs \ (S \ t \ j) \ cid) = \text{Done};$   
 $\text{channel } cid = \text{Some } (p, q) \rrbracket$   
 $\implies \text{map } \text{Msg } (\text{fst } (cs \ (S \ t \ j) \ cid))$   
 $= \text{map } \text{Msg } (\text{fst } (cs \ (S \ t \ i) \ cid)) @ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (S \ t \ i)$   
 $\text{cid})$   
 $\langle \text{proof} \rangle$

**lemma** *cs-when-recording-2*:

**shows**  
 $\llbracket i \leq j; \text{trace init } t \ \text{final};$   
 $\sim \text{has-snapshotted } (S \ t \ i) \ p;$   
 $\forall k. i \leq k \wedge k < j \longrightarrow \sim \text{occurs-on } (t \ ! \ k) = p;$   
 $\text{snd } (cs \ (S \ t \ i) \ cid) = \text{Recording};$   
 $\text{channel } cid = \text{Some } (p, q) \rrbracket$   
 $\implies \text{map } \text{Msg } (\text{fst } (cs \ (S \ t \ j) \ cid)) @ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (S \ t \ j)$   
 $\text{cid})$

$$= \text{map } \text{Msg } (\text{fst } (\text{cs } (S \ t \ i) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (S \ t \ i) \ \text{cid})$$

$$\wedge \ \text{snd } (\text{cs } (S \ t \ j) \ \text{cid}) = \text{Recording}$$
 <proof>

**lemma** *cs-when-recording-3*:

**shows**

$$\llbracket i \leq j; \text{trace init } t \ \text{final};$$

$$\sim \text{has-snapshotted } (S \ t \ i) \ q;$$

$$\forall k. i \leq k \wedge k < j \longrightarrow \sim \text{occurs-on } (t \ ! \ k) = q;$$

$$\text{snd } (\text{cs } (S \ t \ i) \ \text{cid}) = \text{NotStarted};$$

$$\text{has-snapshotted } (S \ t \ i) \ p;$$

$$\text{Marker} : \text{set } (\text{msgs } (S \ t \ i) \ \text{cid});$$

$$\text{channel } \text{cid} = \text{Some } (p, q) \rrbracket$$

$$\implies \text{map } \text{Msg } (\text{fst } (\text{cs } (S \ t \ j) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (S \ t \ j) \ \text{cid})$$

$$= \text{map } \text{Msg } (\text{fst } (\text{cs } (S \ t \ i) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (S \ t \ i) \ \text{cid})$$

$$\wedge \ \text{snd } (\text{cs } (S \ t \ j) \ \text{cid}) = \text{NotStarted}$$
 <proof>

**lemma** *at-most-one-marker*:

**shows**

$$\llbracket \text{trace init } t \ \text{final}; \text{channel } \text{cid} = \text{Some } (p, q) \rrbracket$$

$$\implies \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ i) \ \text{cid})$$

$$\vee (\exists ! j. j < \text{length } (\text{msgs } (S \ t \ i) \ \text{cid}) \wedge \text{msgs } (S \ t \ i) \ \text{cid} \ ! \ j = \text{Marker})$$
 <proof>

**lemma** *last-changes-implies-send-when-msgs-nonempty*:

**assumes**

$$\text{trace init } t \ \text{final} \ \mathbf{and}$$

$$\text{msgs } (S \ t \ i) \ \text{cid} \neq [] \ \mathbf{and}$$

$$\text{msgs } (S \ t \ (i+1)) \ \text{cid} \neq [] \ \mathbf{and}$$

$$\text{last } (\text{msgs } (S \ t \ i) \ \text{cid}) = \text{Marker} \ \mathbf{and}$$

$$\text{last } (\text{msgs } (S \ t \ (i+1)) \ \text{cid}) \neq \text{Marker} \ \mathbf{and}$$

$$\text{channel } \text{cid} = \text{Some } (p, q)$$

**shows**

$$(\exists u \ u' \ m. t \ ! \ i = \text{Send } \text{cid } p \ q \ u \ u' \ m)$$
 <proof>

**lemma** *no-marker-after-RecvMarker*:

**assumes**

$$\text{trace init } t \ \text{final} \ \mathbf{and}$$

$$(S \ t \ i) \vdash \text{RecvMarker } \text{cid } p \ q \mapsto (S \ t \ (i+1)) \ \mathbf{and}$$

$$\text{channel } \text{cid} = \text{Some } (q, p)$$

**shows**

$$\text{Marker} \notin \text{set } (\text{msgs } (S \ t \ (i+1)) \ \text{cid})$$
 <proof>

**lemma** *no-marker-and-snapshotted-implies-no-more-markers-trace:*

**shows**

$\llbracket$  *trace init t final; i ≤ j; j ≤ length t;*  
*has-snapshotted (S t i) p;*  
*Marker ∉ set (msgs (S t i) cid);*  
*channel cid = Some (p, q) \rrbracket  
 $\implies$  *Marker ∉ set (msgs (S t j) cid)**

*<proof>*

**lemma** *marker-not-vanishing-means-always-present:*

**shows**

$\llbracket$  *trace init t final; i ≤ j; j ≤ length t;*  
*Marker : set (msgs (S t i) cid);*  
*Marker : set (msgs (S t j) cid);*  
*channel cid = Some (p, q)*  
 $\rrbracket \implies \forall k. i \leq k \wedge k \leq j \longrightarrow$  *Marker : set (msgs (S t k) cid)*

*<proof>*

**lemma** *last-stays-if-no-recv-marker-and-no-send:*

**shows**

$\llbracket$  *trace init t final; i < j; j ≤ length t;*  
*last (msgs (S t i) cid) = Marker;*  
*Marker : set (msgs (S t i) cid);*  
*Marker : set (msgs (S t j) cid);*  
 $\forall k. i \leq k \wedge k < j \longrightarrow \sim (\exists u u' m. t ! k = \text{Send cid } p \ q \ u \ u' \ m);$   
*channel cid = Some (p, q) \rrbracket  
 $\implies$  *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 ≤ 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) ≠ Marker*  
*channel cid = Some (p, q)*

**shows**

$\exists k u u' m. i \leq k \wedge k < j \wedge t ! k = \text{Send cid } p \ q \ u \ u' \ m$

*<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 ≠ last (msgs (S t i) cid) and*  
*i ≤ length t and*  
*channel cid = Some (p, q)*

**shows**  
 $\exists j. j < i \wedge \text{postrecording-event } t \ j \ (\text{is } ?P)$   
 $\langle \text{proof} \rangle$

**lemma** *same-messages-if-no-occurrence-trace*:

**shows**  
 $\llbracket \text{trace init } t \ \text{final}; i \leq j; j \leq \text{length } t;$   
 $(\forall k. i \leq k \wedge k < j \longrightarrow \text{occurs-on } (t \ ! \ k) \neq p \wedge \text{occurs-on } (t \ ! \ k) \neq q);$   
 $\text{channel } cid = \text{Some } (p, q) \rrbracket$   
 $\implies \text{msgs } (S \ t \ i) \ cid = \text{msgs } (S \ t \ j) \ cid \wedge cs \ (S \ t \ i) \ cid = cs \ (S \ t \ j) \ cid$   
 $\langle \text{proof} \rangle$

**lemma** *snapshot-step-cs-preservation-p*:

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $\sim \text{regular-event } ev$  **and**  
 $\text{occurs-on } ev = p$  **and**  
 $\text{channel } cid = \text{Some } (p, q)$   
**shows**  
 $\text{map } \text{Msg } (\text{fst } (cs \ c \ cid)) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } \ c \ cid)$   
 $= \text{map } \text{Msg } (\text{fst } (cs \ c' \ cid)) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } \ c' \ cid)$   
 $\wedge \text{snd } (cs \ c \ cid) = \text{snd } (cs \ c' \ cid)$   
 $\langle \text{proof} \rangle$

**lemma** *snapshot-step-cs-preservation-q*:

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $\sim \text{regular-event } ev$  **and**  
 $\text{occurs-on } ev = q$  **and**  
 $\text{channel } cid = \text{Some } (p, q)$  **and**  
 $\text{Marker} \notin \text{set } (\text{msgs } \ c \ cid)$  **and**  
 $\sim \text{has-snapshotted } c \ q$   
**shows**  
 $\text{map } \text{Msg } (\text{fst } (cs \ c \ cid)) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } \ c \ cid)$   
 $= \text{map } \text{Msg } (\text{fst } (cs \ c' \ cid)) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } \ c' \ cid)$   
 $\wedge \text{snd } (cs \ c' \ cid) = \text{Recording}$   
 $\langle \text{proof} \rangle$

**lemma** *Marker-in-channel-implies-not-done*:

**assumes**  
 $\text{trace init } t \ \text{final}$  **and**  
 $\text{Marker} : \text{set } (\text{msgs } (S \ t \ i) \ cid)$  **and**  
 $\text{channel } cid = \text{Some } (p, q)$  **and**  
 $i \leq \text{length } t$   
**shows**  
 $\text{snd } (cs \ (S \ t \ i) \ cid) \neq \text{Done}$   
 $\langle \text{proof} \rangle$

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

**shows**

$\llbracket$  *trace init t final*;  $i \leq j$ ;  $j \leq \text{length } t$ ;  
*msgs* ( $S \ t \ i$ ) *cid* =  $\llbracket$ ;  
*has-snapshotted* ( $S \ t \ i$ )  $p$ ;  
*channel cid* = *Some* ( $p, q$ );  
 $\forall k. i \leq k \wedge k < j \wedge \text{regular-event } (t \ ! \ k) \longrightarrow \sim \text{occurs-on } (t \ ! \ k) = p \rrbracket$   
 $\implies \text{msgs } (S \ t \ j) \text{ cid} = \llbracket$

$\langle \text{proof} \rangle$

**lemma** *last-unchanged-or-empty-if-no-events*:

**shows**

$\llbracket$  *trace init t final*;  $i \leq j$ ;  $j \leq \text{length } t$ ;  
*msgs* ( $S \ t \ i$ ) *cid*  $\neq \llbracket$ ;  
*last* (*msgs* ( $S \ t \ i$ ) *cid*) = *Marker*;  
*has-snapshotted* ( $S \ t \ i$ )  $p$ ;  
*channel cid* = *Some* ( $p, q$ );  
 $\forall k. i \leq k \wedge k < j \wedge \text{regular-event } (t \ ! \ k) \longrightarrow \sim \text{occurs-on } (t \ ! \ k) = p \rrbracket$   
 $\implies \text{msgs } (S \ t \ j) \text{ cid} = \llbracket \vee (\text{msgs } (S \ t \ j) \text{ cid} \neq \llbracket \wedge \text{last } (\text{msgs } (S \ t \ j) \text{ cid}) =$

*Marker*)

$\langle \text{proof} \rangle$

**lemma** *cs-after-all-prerecording-events*:

**assumes**

*trace init t final* **and**  
 $\forall i'. i' \geq i \longrightarrow \sim \text{prerecording-event } t \ i'$  **and**  
 $\forall j'. j' < i \longrightarrow \sim \text{postrecording-event } t \ j'$  **and**  
 $i \leq \text{length } t$

**shows**

*cs-equal-to-snapshot* ( $S \ t \ i$ ) *final*

$\langle \text{proof} \rangle$

**lemma** *snapshot-after-all-prerecording-events*:

**assumes**

*trace init t final* **and**  
 $\forall i'. i' \geq i \longrightarrow \sim \text{prerecording-event } t \ i'$  **and**  
 $\forall j'. j' < i \longrightarrow \sim \text{postrecording-event } t \ j'$  **and**  
 $i \leq \text{length } t$

**shows**

*state-equal-to-snapshot* ( $S \ t \ i$ ) *final*

$\langle \text{proof} \rangle$

## 5.4 Obtaining the desired traces

**abbreviation** *all-prerecording-before-postrecording* **where**

*all-prerecording-before-postrecording*  $t \equiv \exists i. (\forall j. j < i \longrightarrow \sim \text{postrecording-event } t \ j)$

$\wedge (\forall j. j \geq i \longrightarrow \sim \text{prerecording-event } t \ j)$   
 $\wedge i \leq \text{length } t$   
 $\wedge \text{trace init } t \ \text{final}$

**definition** *count-violations* :: ('a, 'b, 'c) trace  $\Rightarrow$  nat **where**  
*count-violations* t = sum ( $\lambda i$ . if *postrecording-event* t i  
then card {j  $\in$  {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  $\in$  {i+1..*length* t}. *prerecording-event* t j}  
else 0)  $\geq$  0

*<proof>*

**lemma** *count-violations-ge-0*:

**shows**

*count-violations* t  $\geq$  0

*<proof>*

**lemma** *violations-0-implies-all-subterms-0*:

**assumes**

*count-violations* t = 0

**shows**

$\forall i \in \{0..*length* t\}$ . (if *postrecording-event* t i  
then card {j  $\in$  {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**

$\exists i$ . *postrecording-event* t i  $\wedge$  card {j  $\in$  {i+1..*length* t}. *prerecording-event* t j}  $>$  0 (**is** ?P)

*<proof>*

**lemma** *exists-prerecording-violation-when-card-greater-0*:

**assumes**

card {j  $\in$  {i+1..*length* t}. *prerecording-event* t j}  $>$  0

**shows**

$\exists j \in \{i+1..*length* t\}$ . *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**

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

**lemma exists-neighboring-violation-pair:**

**assumes**

*trace init t final* **and**  
*count-violations t > 0*

**shows**

$\exists i j. i < j \wedge \text{postrecording-event } t i \wedge \text{prerecording-event } t j$   
 $\wedge (\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t ! k)) \wedge j < \text{length } t$

⟨proof⟩

**lemma same-cardinality-post-swap-1:**

**assumes**

*prerecording-event t j* **and**  
*postrecording-event t i* **and**  
*i < j* **and**  
*j < length t* **and**  
*count-violations t = Suc n* **and**  
 $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t ! k)$  **and**  
*trace init t final*

**shows**

$\{k \in \{0..<i\}. \text{prerecording-event } t k\}$   
 $= \{k \in \{0..<i\}. \text{prerecording-event } (\text{swap-events } i j t) k\}$

⟨proof⟩

**lemma same-cardinality-post-swap-2:**

**assumes**

*prerecording-event t j* **and**  
*postrecording-event t i* **and**  
*i < j* **and**  
*j < length t* **and**  
*count-violations t = Suc n* **and**  
 $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t ! k)$  **and**  
*trace init t final*

**shows**

$\text{card } \{k \in \{i..<j+1\}. \text{prerecording-event } t k\}$   
 $= \text{card } \{k \in \{i..<j+1\}. \text{prerecording-event } (\text{swap-events } i j t) k\}$

⟨proof⟩

**lemma same-cardinality-post-swap-3:**

**assumes**

*prerecording-event t j* **and**  
*postrecording-event t i* **and**  
*i < j* **and**  
*j < length t* **and**  
*count-violations t = Suc n* **and**  
 $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t ! k)$  **and**  
*trace init t final*

**shows**

$\{k \in \{j+1..<length\ t\}. \text{prerecording-event } t\ k\}$   
 $= \{k \in \{j+1..<length\ (\text{swap-events } i\ j\ t)\}. \text{prerecording-event } (\text{swap-events } i\ j\ t)\ k\}$   
(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**  
 $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t\ !\ k)$  **and**  
*count-violations*  $t = Suc\ n$  **and**  
*trace* *init*  $t$  *final*

**shows**

$card\ \{k \in \{i+1..<j+1\}. \text{prerecording-event } t\ k\} = 1$   
(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**  
 $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t\ !\ k)$  **and**  
*count-violations*  $t = Suc\ n$  **and**  
*trace* *init*  $t$  *final*

**shows**

$card\ \{k \in \{i+1..<j+1\}. \text{prerecording-event } (\text{swap-events } i\ j\ t)\ k\} = 0$   
(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**  
 $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t\ !\ k)$  **and**  
*count-violations*  $t = Suc\ n$  **and**  
*trace* *init*  $t$  *final*

**shows**

*count-violations*  $(\text{swap-events } i\ j\ t) = n$   
(proof)

**lemma** *desired-trace-always-exists:*

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

**theorem** *snapshot-algorithm-is-correct:*

**assumes**  
*trace init t final*  
**shows**  
 $\exists t' i. \text{trace init } t' \text{ final} \wedge \text{mset } t' = \text{mset } t$   
 $\wedge \text{state-equal-to-snapshot } (S \ t' \ i) \text{ final} \wedge i \leq \text{length } t'$   
⟨*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*  $\equiv (\forall c. p \ c \longrightarrow (\forall t \ c'. \text{trace } c \ t \ c' \longrightarrow p \ c'))$

**lemma** *has-snapshot-stable:*

**assumes**  
*trace init t final*  
**shows**  
*stable*  $(\lambda c. \text{has-snapshotted } c \ p)$   
⟨*proof*⟩

**definition** *some-snapshot-state where*

*some-snapshot-state t*  $\equiv$   
 $SOME \ (t', i). \text{trace init } t \text{ final}$   
 $\wedge \text{trace init } t' \text{ final} \wedge \text{mset } t' = \text{mset } t$   
 $\wedge \text{state-equal-to-snapshot } (S \ t' \ i) \text{ final}$

**lemma** *split-S:*

**assumes**  
*trace init t final*  
**shows**  
*trace*  $(S \ t \ i) \ (\text{drop } i \ t) \ \text{final}$   
⟨*proof*⟩

**theorem** *Stable-Property-Detection:*

**assumes**  
*stable p and*  
*trace init t final and*  
 $(t', i) = \text{some-snapshot-state } t \ \text{and}$   
 $p \ (S \ t' \ i)$   
**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*:  $0 < i \implies ltake\ i\ (LCons\ x\ t) = x \# ltake\ (i - 1)\ t$

⟨proof⟩

**lemma** *take-ltake*:  $i \leq j \implies take\ i\ (ltake\ j\ xs) = ltake\ i\ xs$

⟨proof⟩

**lemma** *nth-ltake [simp]*:  $i < \min\ n\ (\text{length}\ xs) \implies (ltake\ n\ xs) ! i = \text{nth}\ xs\ i$

⟨proof⟩

**lemma** *length-ltake[simp]*:  $length\ (ltake\ i\ xs) = (\text{case}\ \text{length}\ xs\ \text{of}\ \infty \Rightarrow i \mid \text{enat}\ m \Rightarrow \min\ i\ m)$

⟨proof⟩

**lemma** *ltake-prepend*:

$ltake\ i\ (prepend\ xs\ t) = (if\ i \leq length\ xs\ then\ take\ i\ xs\ else\ xs\ @\ ltake\ (i - length\ xs)\ t)$   
*<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*:  $\llbracket c \vdash ev \mapsto c'; cotrace\ c' t \rrbracket \implies cotrace\ c\ (LCons\ ev\ t)$

**lemma** *cotrace-trace*:  $cotrace\ c\ t \implies \exists!c'. trace\ c\ (ltake\ i\ t)\ c'$   
*<proof>*

**lemma** *cotrace-trace'*:  $cotrace\ c\ t \implies \exists c'. trace\ c\ (ltake\ i\ t)\ c'$   
*<proof>*

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

**lemma** *cotrace-trace-cos*:  $cotrace\ c\ t \implies trace\ c\ (ltake\ i\ t)\ (cos\ c\ t\ i)$   
*<proof>*

**lemma** *s-0[simp]*:  $s\ c\ t\ 0 = c$   
*<proof>*

**lemma** *s-chop*:  $i \leq length\ t \implies s\ c\ t\ i = s\ c\ (take\ i\ t)\ i$   
*<proof>*

**lemma** *cotrace-prepend*:  $trace\ c\ t\ c' \implies cotrace\ c'\ u \implies cotrace\ c\ (prepend\ t\ u)$   
*<proof>*

**lemma** *s-Cons*:  $\exists c''. trace\ c'\ xs\ c'' \implies c \vdash ev \mapsto c' \implies s\ c\ (ev\ \# \ xs)\ (Suc\ i) = s\ c'\ xs\ i$   
*<proof>*

**lemma** *cotrace-ldrop*:  $cotrace\ c\ t \implies i \leq llength\ t \implies cotrace\ (cos\ c\ t\ i)\ (ldrop\ i\ t)$   
*<proof>*

**end**

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

*finite-channels:*

*finite*  $\{i. \exists p q. \text{channel } i = \text{Some } (p, q)\}$  **and**

*strongly-connected-raw:*

$\forall p q. (p \neq q) \longrightarrow$

$(\text{trancpl } (\lambda p q. (\exists i. \text{channel } i = \text{Some } (p, q)))) p q$  **and**

*at-least-two-processes:*

*card*  $(UNIV :: 'a \text{ set}) > 1$  **and**

*finite-processes:*

*finite*  $(UNIV :: 'a \text{ set})$  **and**

*no-initial-Marker:*

$\forall i. (\exists p q. \text{channel } i = \text{Some } (p, q))$

$\longrightarrow \text{Marker} \notin \text{set } (\text{msgs } \text{init } i)$  **and**

*no-msgs-if-no-channel:*

$\forall i. \text{channel } i = \text{None} \longrightarrow \text{msgs } \text{init } i = []$  **and**

*no-initial-process-snapshot:*

$\forall p. \neg \text{has-snapshotted } \text{init } p$  **and**

*no-initial-channel-snapshot:*

$\forall i. \text{channel-snapshot } \text{init } i = ([], \text{NotStarted})$  **and**

*valid:*  $\exists t. \text{cotrace } \text{init } t$  **and**

*l1:*  $\forall t i \text{ cid}. \text{cotrace } \text{init } t$

$\wedge \text{Marker} \in \text{set } (\text{msgs } (\text{cos } \text{init } t i) \text{ cid})$

$\longrightarrow (\exists j \leq \text{llength } t. j \geq i \wedge \text{Marker} \notin \text{set } (\text{msgs } (\text{cos } \text{init } t j) \text{ cid}))$  **and**

*l2:*  $\forall t p. \text{cotrace } \text{init } t$

$\longrightarrow (\exists i \leq \text{llength } t. \text{has-snapshotted } (\text{cos } \text{init } t i) p)$

**begin**

**abbreviation** *coS* **where**  $\text{coS} \equiv \text{cos } \text{init}$

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

**lemma** *has-snapshotted:*

$\text{cotrace } \text{init } t \Longrightarrow \text{has-snapshotted } (\text{coS } t (\text{some-snapshot } t p)) p \wedge \text{some-snapshot } t p \leq \text{llength } t$

*<proof>*

**lemma** *cotrace-cos:*  $\text{cotrace } \text{init } t \Longrightarrow j < \text{llength } t \Longrightarrow$

$(\text{coS } t j) \vdash \text{lth } t j \mapsto (\text{coS } t (\text{Suc } j))$

*<proof>*

**lemma** *snapshot-stable:*

$\text{cotrace } \text{init } t \Longrightarrow i \leq j \Longrightarrow \text{has-snapshotted } (\text{coS } t i) p \Longrightarrow \text{has-snapshotted } (\text{coS } t j) p$

*<proof>*

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

$\text{cotrace init } t \implies i \leq j \implies \forall p. \text{has-snapshotted } (\text{coS } t \ i) \ p \implies$   
 $\text{Marker} \notin \text{set } (\text{msgs } (\text{coS } t \ i) \ c) \implies \text{Marker} \notin \text{set } (\text{msgs } (\text{coS } t \ j) \ c)$   
 <proof>

**lemma** *cotrace-all-have-snapshotted*:

**assumes** *cotrace init t*  
**shows**  $\exists i \leq \text{llength } t. \forall p. \text{has-snapshotted } (\text{coS } t \ i) \ p$   
 <proof>

**lemma** *no-messages-if-no-channel*:

**assumes** *cotrace init t*  
**shows**  $\text{channel } \text{cid} = \text{None} \implies \text{msgs } (\text{coS } t \ i) \ \text{cid} = []$   
 <proof>

**lemma** *cotrace-all-have-snapshotted-and-no-markers*:

**assumes** *cotrace init t*  
**shows**  $\exists i \leq \text{llength } t. (\forall p. \text{has-snapshotted } (\text{coS } t \ i) \ p) \wedge$   
 $(\forall c. \text{Marker} \notin \text{set } (\text{msgs } (\text{coS } t \ i) \ c))$   
 <proof>

**context**

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

**definition** *final-i*  $\equiv$

$(\text{SOME } i. i \leq \text{llength } t \wedge (\forall p. \text{has-snapshotted } (\text{coS } t \ i) \ p) \wedge (\forall c. \text{Marker} \notin \text{set } (\text{msgs } (\text{coS } t \ i) \ c)))$

**definition** *final where*

*final* = *coS t final-i*

**lemma** *final-i*:  $\text{final-i} \leq \text{llength } t \ (\forall p. \text{has-snapshotted } (\text{coS } t \ \text{final-i}) \ p) \ (\forall c. \text{Marker} \notin \text{set } (\text{msgs } (\text{coS } t \ \text{final-i}) \ c))$   
 <proof>

**lemma** *final*:  $\exists t. \text{trace init } t \ \text{final} \ (\forall p. \text{has-snapshotted } \text{final} \ p) \ (\forall c. \text{Marker} \notin \text{set } (\text{msgs } \text{final} \ c))$   
 <proof>

**interpretation** *computation channel trans send recv init final*

<proof>

**definition** *coperm where*

$\text{coperm } l \ r = (\exists xs \ ys \ z. \text{mset } xs = \text{mset } ys \wedge l = \text{prepend } xs \ z \wedge r = \text{prepend } ys \ z)$

**lemma** *copermIL*:  $\text{mset } ys = \text{mset } xs \implies t = \text{prepend } xs \ z \implies \text{coperm } (\text{prepend } ys \ z) \ t$

*<proof>*

**lemma** *snapshot-algorithm-is-cocorrect*:

$\exists t' i. \text{cotrace init } t' \wedge \text{coperm } t' t \wedge \text{state-equal-to-snapshot } (\text{coS } t' i) \text{ final} \wedge i \leq \text{final-}i$   
*<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*  $\Rightarrow$  *SType*  $\Rightarrow$  *SType*  $\Rightarrow$  *bool* **where**  
*trans* *p s s'* = *False*

**fun** *send* :: *channel-id*  $\Rightarrow$  *PType*  $\Rightarrow$  *PType*  $\Rightarrow$  *SType*  
 $\Rightarrow$  *SType*  $\Rightarrow$  *MType*  $\Rightarrow$  *bool* **where**  
*send* *c p q s s' m* =  $((c = 0 \wedge p = P \wedge q = Q$   
 $\wedge s = S\text{-Send} \wedge s' = S\text{-Wait} \wedge m = M)$   
 $\vee (c = 1 \wedge p = Q \wedge q = P$   
 $\wedge s = T\text{-Send} \wedge s' = T\text{-Wait} \wedge m = M')$ )

**fun** *recv* :: *channel-id*  $\Rightarrow$  *PType*  $\Rightarrow$  *PType*  $\Rightarrow$  *SType*  
 $\Rightarrow$  *SType*  $\Rightarrow$  *MType*  $\Rightarrow$  *bool* **where**  
*recv* *c p q s s' m* =  $((c = 1 \wedge p = P \wedge q = Q$   
 $\wedge s = S\text{-Wait} \wedge s' = S\text{-Send} \wedge m = M')$   
 $\vee (c = 0 \wedge p = Q \wedge q = P$   
 $\wedge s = T\text{-Wait} \wedge s' = T\text{-Send} \wedge m = M)$ )

**fun** *chan* :: *nat*  $\Rightarrow$  (*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

$init \equiv$  (  
 states = (% $p$ . if  $p = P$  then S-Send else T-Send),  
 msgs = (% $d$ . []),  
 process-snapshot = (% $p$ . None),  
 channel-snapshot = (% $d$ . ([], NotStarted))  
 )

**abbreviation**  $t0$  where  $t0 \equiv$  Snapshot  $P$

**abbreviation**  $s1 :: (PType, SType, MType)$  configuration where

$s1 \equiv$  (  
 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  $t1 \equiv$  Send 0  $P$   $Q$  S-Send S-Wait  $M$

**abbreviation**  $s2 :: (PType, SType, MType)$  configuration where

$s2 \equiv$  (  
 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  $t2 \equiv$  Send 1  $Q$   $P$  T-Send T-Wait  $M'$

**abbreviation**  $s3 :: (PType, SType, MType)$  configuration where

$s3 \equiv$  (  
 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  $t3 \equiv$  Snapshot  $Q$

**abbreviation**  $s4 :: (PType, SType, MType)$  configuration where

$s4 \equiv$  (  
 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  $t4 \equiv \text{RecvMarker } 0 \ Q \ P$

**abbreviation**  $s5$  :: (PType, SType, MType) configuration where

$s5 \equiv$  (  
     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  $t5 \equiv \text{Recv } 1 \ P \ Q \ S\text{-Wait } S\text{-Send } M'$

**abbreviation**  $s6$  :: (PType, SType, MType) configuration where

$s6 \equiv$  (  
     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  $t6 \equiv \text{RecvMarker } 1 \ P \ Q$

**abbreviation**  $s7$  :: (PType, SType, MType) configuration where

$s7 \equiv$  (  
     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**  $s7\text{-no-marker}$ :

**shows**

$\forall cid. \text{Marker} \notin \text{set } (\text{msgs } s7 \ cid)$

$\langle \text{proof} \rangle$

**interpretation** computation chan trans send recv init  $s7$

$\langle \text{proof} \rangle$

**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](http://isa-afp.org/entries/Ordered_Resolution_Prover.html), Formal proof development.