

# 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

February 6, 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>41</b>
2.1	Properties of traces . . . . .	41
2.2	Describing intermediate configurations . . . . .	44
2.3	Trace-related lemmas . . . . .	46
<b>3</b>	<b>Utilities</b>	<b>53</b>
<b>4</b>	<b>Swap lemmas</b>	<b>62</b>
<b>5</b>	<b>The Chandy–Lamport algorithm</b>	<b>95</b>
5.1	The computation locale . . . . .	96
5.2	Termination . . . . .	100
5.3	Correctness . . . . .	106
5.3.1	Pre- and postrecording events . . . . .	109
5.3.2	Event swapping . . . . .	115
5.3.3	Relating configurations and the computed snapshot . . . . .	133
5.3.4	Relating process states . . . . .	149
5.3.5	Relating channel states . . . . .	152

5.4	Obtaining the desired traces . . . . .	199
5.5	Stable property detection . . . . .	217
<b>6</b>	<b>Extension to infinite traces</b>	<b>218</b>
<b>7</b>	<b>Example</b>	<b>226</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*:

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

**by** (*meson event.distinct-disc event.exhaust-disc regular-event*)

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

**assumes**

$p \neq q$

$occurs-on\ ev = p$

**shows**

$occurs-on\ ev \neq q$

using *assms* by (*cases ev*, *auto*)

## 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. msgs* *c i = Msg* *m # xs*)  $\wedge$   
*recv* *i p q s s' m*

**proof** –

**have**  $\exists$  *xs. msgs* *c i = Msg* *m # xs*

**using** *assms can-occur-def*

**by** (*metis* (*mono-tags, lifting*) *event.case*( $\exists$ ) *hd-Cons-tl length-greater-0-conv*)

**then show** *?thesis* **using** *assms can-occur-def* **by** *auto*

**qed**

**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'*. *states* *c p' = states* *c' p'*)  
 $\wedge$  ( $\forall$  *p'*. (*p'  $\neq$  p*)  $\longrightarrow$  *ps* *c' p' = ps* *c p*)  
 $\wedge$  ( $\forall$  *i. ( $\exists$  q. channel* *i = Some* (*p, q*))  $\longrightarrow$  *msgs* *c' i = msgs* *c i* @ [*Marker*])  
 $\wedge$  ( $\forall$  *i. ( $\exists$  q. channel* *i = Some* (*q, p*))  $\longrightarrow$  *channel-snapshot* *c' i = (fst* (*channel-snapshot*  
*c i*), *Recording*))  
 $\wedge$  ( $\forall$  *i. ( $\nexists$  q. channel* *i = Some* (*p, q*))  $\longrightarrow$  *msgs* *c' i = msgs* *c i*)  
 $\wedge$  ( $\forall$  *i. ( $\nexists$  q. channel* *i = Some* (*q, p*))  $\longrightarrow$  *channel-snapshot* *c' i = 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. states* *c r = states* *c' r*)  
 $\wedge$  ( $\forall$  *r. (r  $\neq$  p)  $\longrightarrow$  process-snapshot* *c r = process-snapshot* *c' r*)  
 $\wedge$  (*Marker # msgs* *c' i = msgs* *c i*)  
 $\wedge$  (*channel-snapshot* *c' i = (fst* (*channel-snapshot* *c i*), *Done*))  
 $\wedge$  (*if has-snapshotted* *c p*  
*then* (*process-snapshot* *c p = process-snapshot* *c' p*)  
 $\wedge$  ( $\forall$  *i'. (i'  $\neq$  i)  $\longrightarrow$  msgs* *c' i' = msgs* *c i'*)  
 $\wedge$  ( $\forall$  *i'. (i'  $\neq$  i)  $\longrightarrow$  channel-snapshot* *c i' = channel-snapshot* *c' i'*)  
*else* (*process-snapshot* *c' p = Some* (*states* *c p*))  
 $\wedge$  ( $\forall$  *i'. i'  $\neq$  i  $\wedge$  ( $\exists$  r. channel* *i' = Some* (*p, r*))  
 $\longrightarrow$  *msgs* *c' i' = msgs* *c i'* @ [*Marker*])  
 $\wedge$  ( $\forall$  *i'. i'  $\neq$  i  $\wedge$  ( $\exists$  r. channel* *i' = Some* (*r, p*))  
 $\longrightarrow$  *channel-snapshot* *c' i' = (fst* (*channel-snapshot* *c i'*), *Recording*))  
 $\wedge$  ( $\forall$  *i'. i'  $\neq$  i  $\wedge$  ( $\nexists$  r. channel* *i' = Some* (*p, r*))  
 $\longrightarrow$  *msgs* *c' i' = msgs* *c i'*)  
 $\wedge$  ( $\forall$  *i'. i'  $\neq$  i  $\wedge$  ( $\nexists$  r. channel* *i' = Some* (*r, p*))  
 $\longrightarrow$  *channel-snapshot* *c' i' = channel-snapshot* *c i'*))

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

*check-trans-occur* *c c' p s s'*  $\equiv$

$$\begin{aligned}
& (\text{can-occur } (\text{Trans } p \ s \ s') \ c) \\
& \wedge (\text{states } c' \ p = s') \\
& \wedge (\forall r. (r \neq p) \longrightarrow \text{states } c' \ r = \text{states } c \ r) \\
& \wedge (\forall i. \text{msgs } c' \ i = \text{msgs } c \ i) \\
& \wedge (\text{no-snapshot-change } c \ c')
\end{aligned}$$

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

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

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

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

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**

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

$c \vdash ev \mapsto c''$

**shows**

$c' = c''$

**proof** (*cases ev*)

**case** (*Snapshot p*)

**then have** *states*  $c' = \text{states } c''$  **using** *assms* **by** *auto*

**moreover have** *msgs*  $c' = \text{msgs } c''$

**proof** (*rule ext*)

**fix** *i*

**show** *msgs*  $c' i = \text{msgs } c'' i$

**proof** (*cases channel i = None*)

**case** *True*

**then show** *?thesis* **using** *Snapshot assms* **by** *auto*

**next**

**case** *False*

**then obtain** *r s* **where** *channel i = Some (r, s)* **by** *auto*

**with** *assms Snapshot* **show** *?thesis* **by** (*cases r = p, simp-all*)

**qed**

**qed**

**moreover have** *process-snapshot*  $c' = \text{process-snapshot } c''$  **by** (*metis Snapshot assms next-snapshot ext*)

**moreover have** *channel-snapshot*  $c' = \text{channel-snapshot } c''$

**proof** (*rule ext*)

**fix** *i*

**show** *channel-snapshot*  $c' i = \text{channel-snapshot } c'' i$

**proof** (*cases channel i = None*)

**case** *True*

**then show** *?thesis* **using** *assms Snapshot* **by** *simp*

**next**

**case** *False*

**then obtain** *r s* **where** *channel i = Some (r, s)* **by** *auto*

**with** *assms Snapshot* **show** *?thesis* **by** (*cases s = p, simp-all*)

**qed**

**qed**

**ultimately show**  $c' = c''$  **by** *simp*

**next**

**case** (*RecvMarker i p*)

**then have** *states*  $c' = \text{states } c''$  **using** *assms* **by** *auto*

**moreover have** *msgs*  $c' = \text{msgs } c''$

**proof** (*rule ext*)

**fix** *i'*

**show** *msgs*  $c' i' = \text{msgs } c'' i'$

**proof** (*cases i' = i*)

**case** *True*

**then have** *Marker # msgs*  $c' i' = \text{msgs } c i'$  **using** *assms RecvMarker* **by**

```

simp
  also have ... = Marker # msgs c'' i' using assms RecvMarker ⟨i' = i⟩ by
simp
  finally show ?thesis by simp
next
case False
then show ?thesis
proof (cases has-snapshotted c p)
  case True
  then show ?thesis using assms RecvMarker ⟨i' ≠ i⟩ by simp
next
case no-snap: False
then show ?thesis
proof (cases channel i' = None)
  case True
  then show ?thesis using assms RecvMarker ⟨i' ≠ i⟩ no-snap by simp
next
case False
then obtain r s where channel i' = Some (r, s) by auto
  with assms RecvMarker no-snap ⟨i' ≠ i⟩ show ?thesis by (cases r = p;
simp-all)
  qed
  qed
  qed
  qed
moreover have process-snapshot c' = process-snapshot c''
proof (rule ext)
  fix r
  show ps c' r = ps c'' r
  proof (cases r ≠ p)
    case True
    then show ?thesis using assms RecvMarker by simp
  next
  case False
  with assms RecvMarker ⟨ $\sim$  r ≠ p⟩ show ?thesis by (cases has-snapshotted c
r, auto)
  qed
  qed
moreover have channel-snapshot c' = channel-snapshot c''
proof (rule ext)
  fix i'
  show cs c' i' = cs c'' i'
  proof (cases i' = i)
    case True
    then show ?thesis using assms RecvMarker by simp
  next
  case False
  then show ?thesis
  proof (cases has-snapshotted c p)

```

```

    case True
    then show ?thesis using assms RecvMarker <i' ≠ i> by simp
next
case no-snap: False
then show ?thesis
proof (cases channel i' = None)
  case True
  then show ?thesis using assms RecvMarker <i' ≠ i> no-snap by simp
next
case False
then obtain r s where channel i' = Some (r, s) by auto
  with assms RecvMarker no-snap <i' ≠ i> show ?thesis by (cases s = p;
simp-all)
  qed
  qed
  qed
  qed
ultimately show c' = c'' by simp
next
case (Trans p s s')
then have states c' = states c'' by (metis (no-types, lifting) assms next-trans
ext)
  moreover have msgs c' = msgs c'' using assms Trans by auto
  moreover have process-snapshot c' = process-snapshot c'' using assms Trans
by auto
  moreover have channel-snapshot c' = channel-snapshot c'' using assms Trans
by auto
  ultimately show c' = c'' by simp
next
case (Send i p s s' m)
then have states c' = states c'' by (metis (no-types, lifting) assms next-send
ext)
  moreover have msgs c' = msgs c''
proof (rule ext)
  fix i'
  from assms Send show msgs c' i' = msgs c'' i' by (cases i' = i, simp-all)
qed
  moreover have process-snapshot c' = process-snapshot c'' using assms Send by
auto
  moreover have channel-snapshot c' = channel-snapshot c'' using assms Send
by auto
  ultimately show c' = c'' by simp
next
case (Recv i p s s' m)
then have states c' = states c'' by (metis (no-types, lifting) assms next-recv ext)
  moreover have msgs c' = msgs c''
proof (rule ext)
  fix i'
  from assms Recv show msgs c' i' = msgs c'' i' by (cases i' = i, simp-all)

```

```

qed
moreover have process-snapshot  $c' = \text{process-snapshot } c''$  using assms Recv by
auto
moreover have channel-snapshot  $c' = \text{channel-snapshot } c''$ 
proof (rule ext)
  fix  $i'$ 
  show  $cs\ c'\ i' = cs\ c''\ i'$ 
  proof (cases  $i' \neq i$ )
    case True
      then show ?thesis using assms Recv by simp
    next
      case False
        with assms Recv show ?thesis by (cases snd ( $cs\ c\ i'$ ) = Recording, auto)
  qed
qed
ultimately show  $c' = c''$  by simp
qed

lemma exists-next-if-can-occur:
  assumes
    can-occur ev c
  shows
     $\exists c'. c \vdash ev \mapsto c'$ 
proof (cases ev)
  case (Snapshot p)
    let  $?c = \langle \text{states} = \text{states } c,$ 
       $\text{msgs} = \%i. \text{if } (\exists q. \text{channel } i = \text{Some } (p, q)) \text{ then } \text{msgs } c\ i\ @\ [\text{Marker}]$ 
       $\text{else } \text{msgs } c\ i,$ 
       $\text{process-snapshot} = \%r. \text{if } r = p \text{ then } \text{Some } (\text{states } c\ p) \text{ else } \text{ps } c\ r,$ 
       $\text{channel-snapshot} = \%i. \text{if } (\exists q. \text{channel } i = \text{Some } (q, p)) \text{ then } (\text{fst } (cs$ 
       $c\ i), \text{Recording}) \text{ else } cs\ c\ i \rangle$ 
    have  $c \vdash ev \mapsto ?c$  using Snapshot assms by auto
    then show ?thesis by blast
  next
    case (RecvMarker i p q)
      show ?thesis
      proof (cases has-snapshotted c p)
        case True
          let  $?c = \langle \text{states} = \text{states } c,$ 
             $\text{msgs} = \%i'. \text{if } i = i' \text{ then } \text{tl } (\text{msgs } c\ i') \text{ else } \text{msgs } c\ i',$ 
             $\text{process-snapshot} = \text{ps } c,$ 
             $\text{channel-snapshot} = \%i'. \text{if } i = i' \text{ then } (\text{fst } (cs\ c\ i'), \text{Done}) \text{ else } cs\ c$ 
             $i' \rangle$ 
          have  $\text{msgs } c\ i = \text{Marker} \# \text{msgs } ?c\ i$ 
            using assms can-occur-def RecvMarker hd-Cons-tl by fastforce
          then have  $c \vdash ev \mapsto ?c$  using True RecvMarker assms by auto
          then show ?thesis by blast
        next
          case False

```

```

let ?c = (| states = states c,
          msgs = %i'. if i' = i
              then tl (msgs c i')
              else if (∃ r. channel i' = Some (p, r))
                  then msgs c i' @ [Marker]
                  else msgs c i',
          process-snapshot = %r. if r = p then Some (states c r) else ps c r,
          channel-snapshot = %i'. if i = i' then (fst (cs c i'), Done)
              else if (∃ r. channel i' = Some (r, p))
                  then (fst (cs c i'), Recording)
                  else cs c i' )

have msgs c i = Marker # msgs ?c i
  using assms can-occur-def RecvMarker hd-Cons-tl by fastforce
moreover have ps ?c p = Some (states c p) by simp
ultimately have c ⊢ ev ⇨ ?c using RecvMarker assms False by auto
then show ?thesis by blast

qed
next
case (Trans p s s')
let ?c = (| states = %r. if r = p then s' else states c r,
          msgs = msgs c,
          process-snapshot = ps c,
          channel-snapshot = cs c )
have c ⊢ ev ⇨ ?c
  using Trans assms by auto
then show ?thesis by blast
next
case (Send i p q s s' msg)
let ?c = (| states = %r. if r = p then s' else states c r,
          msgs = %i'. if i = i' then msgs c i' @ [Msg msg] else msgs c i',
          process-snapshot = ps c,
          channel-snapshot = cs c )
have c ⊢ ev ⇨ ?c
  using Send assms by auto
then show ?thesis by blast
next
case (Recv i p q s s' msg)
then show ?thesis
proof (cases snd (cs c i))
case Recording
let ?c = (| states = %r. if r = p then s' else states c r,
          msgs = %i'. if i = i' then tl (msgs c i') else msgs c i',
          process-snapshot = ps c,
          channel-snapshot = %i'. if i = i'
                                then (fst (cs c i') @ [msg], Recording)
                                else cs c i' )
have c ⊢ ev ⇨ ?c
  using Recv Recording assms can-occur-Recv by fastforce
then show ?thesis by blast

```

```

next
  case Done
  let ?c = (| states = %r. if r = p then s' else states c r,
            msgs = %i'. if i = i' then tl (msgs c i') else msgs c i',
            process-snapshot = ps c,
            channel-snapshot = cs c |)
  have c ⊢ ev ↦ ?c
  using Done Recv assms can-occur-Recv by fastforce
  then show ?thesis by blast
next
  case NotStarted
  let ?c = (| states = %r. if r = p then s' else states c r,
            msgs = %i'. if i = i' then tl (msgs c i') else msgs c i',
            process-snapshot = ps c,
            channel-snapshot = cs c |)
  have c ⊢ ev ↦ ?c
  using NotStarted Recv assms can-occur-Recv by fastforce
  then show ?thesis by blast
qed
qed

lemma exists-exactly-one-following-state:
  can-occur ev c ⇒ ∃!c'. c ⊢ ev ↦ c'
  using exists-next-if-can-occur state-and-event-determine-next by blast

lemma no-state-change-if-no-event:
  assumes
    c ⊢ ev ↦ c' and
    occurs-on ev ≠ p
  shows
    states c p = states c' p ∧ process-snapshot c p = process-snapshot c' p
  using assms by (cases ev, auto)

lemma no-msgs-change-if-no-channel:
  assumes
    c ⊢ ev ↦ c' and
    channel i = None
  shows
    msgs c i = msgs c' i
  using assms proof (cases ev)
  case (RecvMarker cid p)
  then have cid ≠ i using assms RecvMarker can-occur-def by fastforce
  with assms RecvMarker show ?thesis by (cases has-snapshotted c p, auto)
next
  case (Send cid p s s' m)
  then have cid ≠ i using assms Send can-occur-def by fastforce
  then show ?thesis using assms Send by auto
next
  case (Recv cid p s s' m)

```

**then have**  $cid \neq i$  **using** *assms Recv can-occur-def* **by** *fastforce*  
**then show** *?thesis* **using** *assms Recv* **by** *simp*  
**qed** *simp-all*

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

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $channel\ i = None$   
**shows**  
 $cs\ c\ i = cs\ c'\ i$   
**using** *assms* **proof** (*cases ev*)  
**case** (*RecvMarker cid p*)  
**then have**  $cid \neq i$  **using** *assms RecvMarker can-occur-def* **by** *fastforce*  
**with** *assms RecvMarker* **show** *?thesis* **by** (*cases has-snapshotted c p, auto*)  
**next**  
**case** (*Send cid p s s' m*)  
**then have**  $cid \neq i$  **using** *assms Send can-occur-def* **by** *fastforce*  
**then show** *?thesis* **using** *assms Send* **by** *auto*  
**next**  
**case** (*Recv cid p s s' m*)  
**then have**  $cid \neq i$  **using** *assms Recv can-occur-def* **by** *fastforce*  
**then show** *?thesis* **using** *assms Recv* **by** *simp*  
**qed** *simp-all*

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

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $isSend\ ev \longrightarrow getId\ ev \neq i$  **and**  
 $isRecv\ ev \longrightarrow getId\ ev \neq i$  **and**  
 $regular-event\ ev$   
**shows**  
 $msgs\ c\ i = msgs\ c'\ i$   
**proof** (*cases channel i = None*)  
**case** *True*  
**then show** *?thesis* **using** *assms no-msgs-change-if-no-channel* **by** *simp*  
**next**  
**have**  $isTrans\ ev \vee isSend\ ev \vee isRecv\ ev$  **using** *assms* **by** *simp*  
**then show** *?thesis*  
**proof** (*elim disjE*)  
**assume**  $isTrans\ ev$   
**then show** *?thesis*  
**by** (*metis assms(1) event.collapse(1) next-trans*)  
**next**  
**assume**  $isSend\ ev$   
**then obtain**  $i' r s u u' m$  **where**  $Send: ev = Send\ i'\ r\ s\ u\ u'\ m$  **by** (*meson isSend-def*)  
**then show** *?thesis* **using** *Send assms* **by** *auto*  
**next**  
**assume**  $isRecv\ ev$

**then obtain**  $i' r s u u' m$  **where**  $ev = Recv\ i' r s u u' m$  **by** (*meson isRecv-def*)  
**then show** *?thesis* **using** *assms* **by** *auto*  
**qed**  
**qed**

**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** –  
**have**  $isTrans\ ev \vee isSend\ ev \vee isRecv\ ev$  **using** *assms* **by** *simp*  
**then show** *?thesis*  
**proof** (*elim disjE*)  
**assume**  $isTrans\ ev$   
**then show** *?thesis*  
**by** (*metis assms(1) event.collapse(1) next-trans*)  
**next**  
**assume**  $isSend\ ev$   
**then obtain**  $i' r s u u' m$  **where**  $ev = Send\ i' r s u u' m$  **by** (*meson isSend-def*)  
**then show** *?thesis* **using** *assms* **by** *auto*  
**next**  
**assume**  $isRecv\ ev$   
**then obtain**  $i r s u u' m$  **where**  $ev = Recv\ i r s u u' m$  **by** (*meson isRecv-def*)  
**then show** *?thesis* **using** *assms* **by** *auto*  
**qed**  
**qed**

**lemma** *happen-implies-can-occur*:

**assumes**  
 $c \vdash ev \mapsto c'$   
**shows**  
 $can-occur\ ev\ c$   
**proof** –  
**show** *?thesis* **using** *assms* **by** (*cases ev, auto*)  
**qed**

**lemma** *snapshot-increases-message-length*:

**assumes**  
 $ev = Snapshot\ p$  **and**  
 $c \vdash ev \mapsto c'$  **and**  
 $channel\ i = Some\ (q, r)$   
**shows**  
 $length\ (msgs\ c\ i) \leq length\ (msgs\ c'\ i)$   
**using** *assms* **by** (*cases p = q, auto*)

**lemma** *recv-marker-changes-head-only-at-i*:

```

assumes
   $ev = \text{RecvMarker } i \ p \ q$  and
   $c \vdash ev \mapsto c'$  and
   $i' \neq i$ 
shows
   $\text{msgs } c \ i' = [] \vee \text{hd } (\text{msgs } c \ i') = \text{hd } (\text{msgs } c' \ i')$ 
proof (cases channel  $i' = \text{None}$ )
  case True
  then show ?thesis using assms no-msgs-change-if-no-channel by presburger
next
  case False
  then show ?thesis
  proof (cases  $\text{msgs } c \ i'$ )
    case Nil
    then show ?thesis by simp
  next
    case (Cons m xs)
    then obtain  $r \ s$  where channel  $i' = \text{Some } (r, s)$  using False by auto
    then show ?thesis
    proof (cases has-snapshotted c p)
      case True
      then show ?thesis using assms by auto
    next
      case False
      with assms show ?thesis by (cases  $r = p$ , auto)
    qed
  qed
qed

```

**lemma** *recv-marker-other-channels-not-shrinking*:

```

assumes
   $ev = \text{RecvMarker } i \ p \ q$  and
   $c \vdash ev \mapsto c'$ 
shows
   $\text{length } (\text{msgs } c \ i') \leq \text{length } (\text{msgs } c' \ i') \iff i \neq i'$ 
proof (rule iffI)
  show  $\text{length } (\text{msgs } c \ i') \leq \text{length } (\text{msgs } c' \ i') \implies i \neq i'$ 
  proof (rule ccontr)
    assume asm:  $\sim i \neq i' \ \text{length } (\text{msgs } c \ i') \leq \text{length } (\text{msgs } c' \ i')$ 
    then have  $\text{msgs } c \ i = \text{Marker } \# \ \text{msgs } c' \ i$  using assms by auto
    then have  $\text{length } (\text{msgs } c \ i) > \text{length } (\text{msgs } c' \ i)$  by simp
    then have  $\text{length } (\text{msgs } c \ i') > \text{length } (\text{msgs } c' \ i')$  using asm by simp
    then show False using asm by simp
  qed
next
  show  $i \neq i' \implies \text{length } (\text{msgs } c \ i') \leq \text{length } (\text{msgs } c' \ i')$ 
  proof –
    assume  $i \neq i'$ 
    then show ?thesis

```

```

proof (cases channel i' = None)
  case True
  then show ?thesis using assms no-msgs-change-if-no-channel by presburger
next
  case False
  then obtain r s where chan: channel i' = Some (r, s) by auto
  then show ?thesis
  proof (cases has-snapshot c p)
    case True
    with assms <i ≠ i'> show ?thesis by auto
  next
    case no-snap: False
    then show ?thesis
    proof (cases p = r)
      case True
      then have msgs c' i' = msgs c i' @ [Marker] using <i ≠ i'> assms no-snap
chan by auto
      then show ?thesis by auto
    next
      case False
      then show ?thesis using assms <i ≠ i'> chan no-snap by auto
    qed
  qed
qed
qed
qed

```

```

lemma regular-event-cannot-induce-snapshot:
  assumes
    ~ has-snapshot c p and
    c ⊢ ev ↦ c'
  shows
    regular-event ev ⟶ ~ has-snapshot c' p
proof (cases ev)
  case (Trans q s s')
    then show ?thesis using assms(1) assms(2) by auto
next
  case (Send q r s s' m)
    then show ?thesis using assms by auto
next
  case (Recv q r s s' m)
    then show ?thesis using assms by auto
qed simp-all

```

```

lemma regular-event-preserves-process-snapshots:
  assumes
    c ⊢ ev ↦ c'
  shows
    regular-event ev ⟹ ps c r = ps c' r

```

```

proof (cases ev)
  case (Trans p s s')
  then show ?thesis
    using assms by auto
next
  case (Send p q s s' m)
  then show ?thesis
    using assms by auto
next
  case (Recv p q s s' m)
  then show ?thesis
    using assms by auto
qed simp-all

```

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

```

assumes
  ~ regular-event ev and
  c ⊢ ev ⇨ c'
shows
  states c p = states c' p
proof –
  have isSnapshot ev ∨ isRecvMarker ev using nonregular-event assms by auto
  then show ?thesis
  proof (elim disjE, goal-cases)
    case 1
    then obtain q where ev = Snapshot q
      by (meson isSnapshot-def)
    then show ?thesis
      using assms(2) by auto
    next
    case 2
    then obtain i q r where ev = RecvMarker i q r
      by (meson isRecvMarker-def)
    then show ?thesis using assms(2) by auto
  qed
qed

```

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

```

assumes
  ~ has-snapshotted c p and
  c ⊢ ev ⇨ c' and
  occurs-on ev = p and
  ~ regular-event ev
shows
  ~ regular-event ev ⟶ has-snapshotted c' p
proof (cases ev)
  case (Snapshot q)
  then have q = p using assms by auto
  then show ?thesis using Snapshot assms(2) by auto

```

```

next
  case (RecvMarker i q r)
  then have  $q = p$  using assms by auto
  then show ?thesis using RecvMarker assms by auto
qed (simp-all add: assms)

lemma snapshot-state-unchanged:
  assumes
    step:  $c \vdash ev \mapsto c'$  and
    has-snapshot  $c\ p$ 
  shows
     $ps\ c\ p = ps\ c'\ p$ 
proof (cases occurs-on  $ev = p$ )
  case False
  then show ?thesis
    using local.step no-state-change-if-no-event by auto
next
  case True
  then show ?thesis
proof (cases regular-event  $ev$ )
  case True
  then show ?thesis
    using local.step regular-event-preserves-process-snapshots by auto
next
  case False
  have isRecvMarker  $ev$ 
proof (rule ccontr)
  have  $isSnapshot\ ev \vee isRecvMarker\ ev$ 
    using False nonregular-event by blast
  moreover assume  $\sim isRecvMarker\ ev$ 
  ultimately have  $isSnapshot\ ev$  by simp
  then have  $ev = Snapshot\ p$  by (metis True event.collapse(4))
  then have can-occur  $ev\ c$ 
    using happen-implies-can-occur local.step by blast
  then have  $\sim has-snapshot\ c\ p$  unfolding can-occur-def
    by (simp add: ev = Snapshot p)
  then show False using assms by auto
qed
then show ?thesis
proof -
  have  $\exists n\ pa. c \vdash RecvMarker\ n\ p\ pa \mapsto c'$ 
    by (metis True isRecvMarker ev event.collapse(5) local.step)
  then show ?thesis
    using assms(2) by force
qed
qed
qed
qed

```

lemma *message-must-be-delivered*:

```

assumes
  valid:  $c \vdash ev \mapsto c'$  and
  delivered:  $(\text{msgs } c \ i \neq [] \wedge \text{hd } (\text{msgs } c \ i) = m) \wedge (\text{msgs } c' \ i = [] \vee \text{hd } (\text{msgs } c' \ i) \neq m)$ 
shows
   $(\exists p \ q. \quad ev = \text{RecvMarker } i \ p \ q \quad \wedge \quad m = \text{Marker})$ 
   $\vee (\exists p \ q \ s \ s' \ m'. \ ev = \text{Recv } i \ p \ q \ s \ s' \ m' \wedge \quad m = \text{Msg } m')$ 
proof (cases ev)
  case (Snapshot p)
  then show ?thesis
  proof (cases msgs c i)
    case Nil
    then show ?thesis using delivered by simp
  next
  case (Cons m xs)
  with assms Snapshot show ?thesis
  proof (cases channel i = None)
    case True
    then show ?thesis using assms Snapshot by auto
  next
  case False
  then obtain r s where chan: channel i = Some (r, s) by auto
  then show ?thesis
  proof (cases r = p)
    case True
    then have  $\text{msgs } c' \ i = \text{msgs } c \ i \ @ \ [\text{Marker}]$  using assms(1) Snapshot chan
  by auto
    then show ?thesis using delivered by auto
  next
  case False
  then have  $\text{msgs } c' \ i = \text{msgs } c \ i$  using assms Snapshot chan by simp
  then show ?thesis using delivered Cons by simp
  qed
  qed
  qed
next
  case (RecvMarker i' p q)
  then have  $i' = i$ 
  by (metis assms(1) delivered le-0-eq length-greater-0-conv list.size(3) recv-marker-changes-head-only-at-i
recv-marker-other-channels-not-shrinking)
  moreover have  $\text{Marker} = m$ 
  using  $\langle i' = i \rangle \text{RecvMarker}$  assms(1) can-occur-def delivered by auto
  moreover have  $\text{channel } i = \text{Some } (q, p)$ 
  using RecvMarker assms(1) calculation(1) can-occur-def by auto
  ultimately show ?thesis using RecvMarker by simp
next
  case (Trans p' s s')
  then show ?thesis
  using valid delivered by auto

```

```

next
  case (Send p' q' s s' m')
  then show ?thesis
    by (metis (no-types, lifting) delivered distributed-system.next.simps(4) dis-
tributed-system-axioms hd-append2 snoc-eq-iff-butlast valid)
next
  case (Recv i' p q s s' m')
  then have i = i'
    using assms(1) delivered by auto
  also have m = Msg m'
    by (metis (no-types, lifting) Recv delivered list.sel(1) next-recv valid)
  ultimately show ?thesis using Recv by auto
qed

lemma message-must-be-delivered-2:
  assumes
    c ⊢ ev ↦ c'
    m : set (msgs c i)
    m ∉ set (msgs c' i)
  shows
    (∃ p q. ev = RecvMarker i p q ∧ m = Marker) ∨ (∃ p q s s' m'. ev = Recv i p
q s s' m' ∧ m = Msg m')
  proof -
    have uneq-sets: set (msgs c i) ≠ set (msgs c' i)
      using assms(2) assms(3) by blast
    then obtain p q where chan: channel i = Some (p, q)
      using assms no-msgs-change-if-no-channel by fastforce
    then show ?thesis
      proof (cases ev)
        case (Snapshot p')
          with Snapshot assms chan have set (msgs c' i) = set (msgs c i) by (cases p'
= p, auto)
          then show ?thesis using uneq-sets by simp
        next
          case (Trans p' s s')
            then show ?thesis using uneq-sets assms by simp
        next
          case (Send i' p' q' s s' m)
            then show ?thesis
              by (metis (no-types, lifting) UnCI assms(1) assms(2) assms(3) local.next.simps(4)
set-append)
        next
          case (RecvMarker i' p' q')
            have i' = i
              proof (rule ccontr)
                assume ~ i' = i
                show False using assms chan RecvMarker
              proof (cases has-snapshot c p')
                case True

```

```

    then show False using assms chan RecvMarker  $\langle \sim i' = i \rangle$  by simp
  next
    case False
    then show False using assms chan RecvMarker  $\langle \sim i' = i \rangle$  by (cases  $p' =$ 
p, simp-all)
    qed
  qed
  moreover have  $m = \text{Marker}$ 
  proof -
    have  $\text{msgs } c \ i' = \text{Marker} \# \text{msgs } c' \ i'$  using assms chan RecvMarker by
auto
    then show ?thesis using assms  $\langle i' = i \rangle$  by simp
  qed
  ultimately show ?thesis using RecvMarker by simp
next
case (Recv  $i' \ p' \ q' \ s' \ m'$ )
have  $i' = i$ 
proof (rule ccontr)
  assume  $\sim i' = i$ 
  then show False
    using Recv assms(1) uneq-sets by auto
  qed
  then have  $i' = i \wedge m = \text{Msg } m'$ 
    using Recv assms by auto
  then show ?thesis using Recv by simp
qed
qed

```

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

```

  assumes
     $ev = \text{RecvMarker } i \ p \ q$  and
     $c \vdash ev \mapsto c'$ 
  shows
    has-snapshotted  $c' \ p$ 
  using assms snapshot-state-unchanged by (cases has-snapshotted  $c \ p$ , auto)

```

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

```

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

```

```

have  $\exists p q. ev = \text{RecvMarker } i \ p \ q$ 
  using assms message-must-be-delivered-2 by blast
then obtain  $r \ s$  where  $\text{RecvMarker}: ev = \text{RecvMarker } i \ r \ s$ 
  by blast
then have  $r = p$ 
  using assms(1) assms(4) can-occur-def by auto
then show ?thesis
  using recv-marker-means-snapshotted-1 assms RecvMarker by blast
qed

```

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

```

assumes
   $c \vdash ev \mapsto c'$  and
   $\text{occurs-on } ev \neq \text{occurs-on } ev'$  and
   $\text{can-occur } ev' \ c$ 
shows
   $\text{can-occur } ev' \ c'$ 
proof (cases ev')
  case (Trans p s s')
  have  $\text{states } c \ p = \text{states } c' \ p$ 
    using Trans assms(1) assms(2) no-state-change-if-no-event by auto
  moreover have  $\text{states } c \ p = s$  using can-occur-def assms Trans by simp
  ultimately have  $\text{states } c' \ p = s$  by simp
  moreover have  $\text{trans } p \ s \ s'$ 
    using Trans assms(3) can-occur-def by auto
  ultimately show ?thesis
    by (simp add: Trans can-occur-def)
next
  case (Recv i p q s s' m)
  then have  $hd \ (\text{msgs } c \ i) = \text{Msg } m$ 
  proof –
    from Recv have  $\text{length} \ (\text{msgs } c \ i) > 0$  using assms(3) can-occur-def by auto
    then obtain  $m' \ xs$  where  $\text{mcqp}: \text{msgs } c \ i = m' \ \# \ xs$ 
      by (metis list.size(3) nat-less-le neq-Nil-conv)
    then have  $\text{Msg } m = m'$ 
    proof (cases m', auto)
      case Marker
      then have  $\text{msgs } c \ i = \text{Marker } \# \ xs$  by (simp add:mcqp)
      then have  $\sim \text{can-occur } ev' \ c$  using Recv can-occur-def by simp
      then show False using assms(3) by simp
    next
      case (Msg msg)
      then have  $\text{msgs } c \ i = \text{Msg } msg \ \# \ xs$  by (simp add: mcqp)
      then show  $m = msg$  using Recv can-occur-def assms(3) by simp
    qed
  then show ?thesis by (simp add: mcqp)
qed
show ?thesis
proof (rule ccontr)

```

**assume**  $asm: \sim \text{can-occur } ev' c'$   
**then have**  $msgs\ c' i = [] \vee hd\ (msgs\ c' i) \neq Msg\ m$   
**using**  $Recv\ assms\ \text{can-occur-def no-state-change-if-no-event distributed-system-axioms}$   
*list.case-eq-if* **by** *fastforce*  
**then obtain**  $i' p' q' s'' s''' m'$  **where**  $RMoR: ev = RecvMarker\ i' p' q' \vee ev = Recv\ i\ p' q' s'' s''' m'$   
**by**  $(metis\ Recv\ \langle hd\ (msgs\ c\ i) = Msg\ m \rangle\ assms(1)\ assms(3)\ \text{can-occur-Recv}\ \text{list.discI}\ \text{message-must-be-delivered})$   
**then have**  $occurs-on\ ev = p$   
**proof** –  
**have**  $f1: states\ c\ p = s \wedge channel\ i = Some\ (q, p) \wedge recv\ i\ p\ q\ s\ s'\ m \wedge 0 < length\ (msgs\ c\ i) \wedge hd\ (msgs\ c\ i) = Msg\ m$   
**using**  $Recv\ assms(3)\ \text{can-occur-def}$  **by** *force*  
**have**  $f2: RecvMarker\ i' p' q' = ev \vee states\ c\ p' = s'' \wedge channel\ i = Some\ (q', p') \wedge recv\ i\ p' q' s'' s''' m' \wedge 0 < length\ (msgs\ c\ i) \wedge hd\ (msgs\ c\ i) = Msg\ m'$   
**using**  $RMoR\ assms(1)\ \text{can-occur-def}$  **by** *force*  
**have**  $\forall e\ n\ c. \exists p\ pa\ s\ sa\ m. \forall ca\ cb. (\neg\ c \vdash e \mapsto ca \vee msgs\ ca\ n \neq [] \vee hd\ (msgs\ c\ n) = Marker \vee msgs\ c\ n = [] \vee Recv\ n\ p\ pa\ s\ sa\ m = e) \wedge (\neg\ c \vdash e \mapsto cb \vee hd\ (msgs\ c\ n) = Marker \vee hd\ (msgs\ cb\ n) = hd\ (msgs\ c\ n) \vee msgs\ c\ n = [] \vee Recv\ n\ p\ pa\ s\ sa\ m = e)$   
**by**  $(metis\ (\text{no-types})\ \text{message-must-be-delivered})$   
**then show** *?thesis*  
**using**  $f2\ f1$  **by**  $(metis\ RMoR\ \langle msgs\ c' i = [] \vee hd\ (msgs\ c' i) \neq Msg\ m \rangle\ assms(1)\ \text{event.disc}(13,15)\ \text{event.sel}(3,5)\ \text{length-greater-0-conv}\ \text{message.distinct}(1)\ \text{option.inject}\ \text{prod.inject})$   
**qed**  
**then show** *False* **using**  $assms\ Recv$  **by** *simp*  
**qed**  
**next**  
**case**  $(Send\ i\ p\ q\ s'\ m)$   
**then have**  $states\ c\ p = states\ c'\ p$  **using**  $assms\ \text{no-state-change-if-no-event}$  **by** *auto*  
**then show**  $\text{can-occur } ev' c'$  **using**  $assms\ assms(3)\ \text{can-occur-def Send}$  **by** *auto*  
**next**  
**case**  $(RecvMarker\ i\ p\ q)$   
**then have**  $msgs-ci: hd\ (msgs\ c\ i) = Marker \wedge length\ (msgs\ c\ i) > 0$   
**proof** –  
**from**  $RecvMarker$  **have**  $length\ (msgs\ c\ i) > 0$  **using**  $assms(3)\ \text{can-occur-def}$  **by** *auto*  
**then obtain**  $m'\ xs$  **where**  $mci: msgs\ c\ i = m' \# xs$   
**by**  $(metis\ list.size(3)\ \text{nat-less-le}\ \text{neq-Nil-conv})$   
**then have**  $m\text{-mark}: Marker = m'$   
**proof**  $(cases\ m',\ auto)$   
**case**  $(Msg\ msg)$   
**then have**  $msgs\ c\ i = Msg\ msg \# xs$  **by**  $(simp\ add:mci)$   
**then have**  $\sim \text{can-occur } ev' c$  **using**  $RecvMarker\ \text{can-occur-def}$  **by** *simp*  
**then show** *False* **using**  $assms(3)$  **by** *simp*  
**qed**  
**then show** *?thesis* **by**  $(simp\ add:\ mci)$

**qed**  
**show** *?thesis*  
**proof** (*rule ccontr*)  
    **assume** *asm*:  $\sim$  *can-occur* *ev'* *c'*  
    **then have** *msgs* *c' i* = []  $\vee$  *hd* (*msgs* *c' i*)  $\neq$  *Marker*  
    **using** *RecvMarker* *assms*(3) *can-occur-def* *list.case-eq-if* **by** *fastforce*  
    **then have**  $\exists p q. ev = \text{RecvMarker } i p q \wedge \text{Marker} = \text{Marker}$  **using** *message-must-be-delivered* *msgs-ci* *assms* **by** *blast*  
    **then obtain** *r s* **where** *RecvMarker-ev*:  $ev = \text{RecvMarker } i r s$  **by** *blast*  
    **then have**  $p = r \wedge q = s$   
    **using** *RecvMarker* *assms*(1) *assms*(3) *can-occur-def* **by** *auto*  
    **then have** *occurs-on*  $ev = p$  **using** *assms* *RecvMarker-ev* **by** *auto*  
    **then show** *False* **using** *assms* **using** *RecvMarker* **by** *auto*  
**qed**  
**next**  
    **case** (*Snapshot* *p*)  
    **then have**  $\sim$  *has-snapshotted* *c p* **using** *assms* *assms*(3) *can-occur-def* **by** *simp*  
    **show** *?thesis*  
    **proof** (*rule ccontr*)  
    **assume** *asm*:  $\sim$  *can-occur* *ev'* *c'*  
    **then have** *has-snapshotted* *c' p* **using** *can-occur-def* *Snapshot* **by** *simp*  
    **then have** *occurs-on*  $ev = p$   
    **using**  $\langle \neg \text{has-snapshotted } c p \rangle$  *assms*(1) *no-state-change-if-no-event* **by** *fastforce*  
    **then show** *False* **using** *assms*(2) *Snapshot* **by** *auto*  
**qed**  
**qed**

**lemma** *msgs-unchanged-for-other-is*:  
**assumes**  
     $c \vdash ev \mapsto c'$  **and**  
    *regular-event* *ev* **and**  
     $\text{getId } ev = i$  **and**  
     $i' \neq i$   
**shows**  
     $\text{msgs } c i' = \text{msgs } c' i'$   
**proof** –  
    **have**  $\text{isTrans } ev \vee \text{isSend } ev \vee \text{isRecv } ev$  **using** *assms* **by** *simp*  
    **then show** *?thesis*  
    **proof** (*elim disjE*, *goal-cases*)  
    **case** 1  
    **then obtain** *p s s'* **where**  $ev = \text{Trans } p s s'$  **by** (*meson isTrans-def*)  
    **then show** *?thesis* **using** *assms* **by** *simp*  
**next**  
    **case** 2  
    **then obtain**  $i' p q s s' m$  **where**  $ev = \text{Send } i' p q s s' m$  **by** (*meson isSend-def*)  
    **then show** *?thesis* **using** *assms* **by** *simp*  
**next**  
    **case** 3

**then obtain**  $i' p q s s' m$  **where**  $ev = Recv\ i' p q s s' m$  **by** (*meson isRecv-def*)  
**with** *assms* **show** *?thesis* **by** *auto*  
**qed**  
**qed**

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

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $ev = RecvMarker\ i\ p\ q$  **and**  
*has-snapshotted*  $c\ p$  **and**  
 $i' \neq i$   
**shows**  
 $msgs\ c\ i' = msgs\ c'\ i'$   
**using** *assms* **by** *auto*

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

**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
*occurs-on*  $ev \neq occurs-on\ ev'$  **and**  
*can-occur*  $ev'\ c'$  **and**  
 $\sim isRecvMarker\ ev'$  **and**  
 $\sim isSend\ ev$   
**shows**  
*can-occur*  $ev'\ c$   
**proof** (*cases ev'*)  
**case** (*Snapshot p*)  
**then have**  $\sim has-snapshotted\ c'\ p$  **using** *assms(3)* *can-occur-def* **by** *simp*  
**then have**  $\sim has-snapshotted\ c\ p$  **using** *assms(1)* *snapshot-state-unchanged* **by**  
*force*  
**then show** *?thesis* **using** *can-occur-def Snapshot* **by** *simp*  
**next**  
**case** (*RecvMarker i p q*)  
**then show** *?thesis* **using** *assms(4)* **by** *auto*  
**next**  
**case** (*Trans p s s'*)  
**then show** *?thesis*  
**using** *assms(1)* *assms(2)* *can-occur-def no-state-change-if-no-event* *assms(3)*  
**by** *auto*  
**next**  
**case** (*Send p q s s' m*)  
**then show** *?thesis*  
**using** *assms(1)* *assms(2)* *can-occur-def no-state-change-if-no-event* *assms(3)*  
**by** *auto*  
**next**  
**case** (*Recv i p q s s' m*)  
**have**  $msgs\ c'\ i \neq Nil$  **using** *Recv can-occur-def* *assms* **by** *auto*  
**moreover have**  $hd\ (msgs\ c'\ i) = Msg\ m \wedge length\ (msgs\ c'\ i) > 0$   
**proof** –  
**from** *Recv* **have**  $length\ (msgs\ c'\ i) > 0$  **using** *assms(3)* *can-occur-def* **by** *auto*

```

then obtain  $m'$   $xs$  where  $mcqp: msgs\ c'\ i = m' \# xs$ 
  by ( $metis\ list.size(3)\ nat-less-le\ neq-Nil-conv$ )
then have  $Msg\ m = m'$ 
proof ( $cases\ m',\ auto$ )
  case  $Marker$ 
    then have  $msgs\ c'\ i = Marker \# xs$  by ( $simp\ add:mcqp$ )
    then have  $\sim\ can-occur\ ev'\ c'$  using  $Recv\ can-occur-def$  by  $simp$ 
    then show  $False$  using  $assms(3)$  by  $simp$ 
  next
    case ( $Msg\ msg$ )
    then have  $msgs\ c'\ i = Msg\ msg \# xs$  by ( $simp\ add:mcqp$ )
    then show  $m = msg$  using  $Recv\ can-occur-def\ assms(3)$  by  $simp$ 
  qed
then show  $?thesis$  by ( $simp\ add:mcqp$ )
qed
moreover have  $msgs\ c\ i \neq Nil \wedge hd\ (msgs\ c'\ i) = hd\ (msgs\ c\ i)$ 
proof ( $cases\ ev$ )
  case ( $Snapshot\ p'$ )
    then have  $p' \neq p$  using  $assms\ Recv$  by  $simp$ 
    have  $chan: channel\ i = Some\ (q, p)$ 
    by ( $metis\ Recv\ assms(3)\ distributed-system.can-occur-Recv\ distributed-system-axioms$ )
    with  $Snapshot\ assms$  have  $length\ (msgs\ c\ i) > 0 \wedge hd\ (msgs\ c\ i) = hd\ (msgs$ 
 $c'\ i)$ 
    proof ( $cases\ q = p'$ )
      case  $True$ 
        then have  $msgs\ c'\ i = msgs\ c\ i @ [Marker]$  using  $Snapshot\ chan\ assms$  by
 $simp$ 
        then show  $?thesis$ 
        by ( $metis\ append-self-conv2\ calculation(2)\ hd-append2\ length-greater-0-conv$ 
 $list.sel(1)\ message.simps(3)$ )
      next
        case  $False$ 
        then have  $msgs\ c'\ i = msgs\ c\ i$  using  $Snapshot\ chan\ assms$  by  $simp$ 
        then show  $?thesis$  using  $calculation$  by  $simp$ 
    qed
    then show  $?thesis$  by  $simp$ 
  next
    case ( $RecvMarker\ i'\ p'\ q'$ )
    then have  $i' \neq i$ 
    using  $Recv\ assms(1)\ assms(2)\ assms(3)\ can-occur-def$  by  $force$ 
    then show  $?thesis$ 
proof ( $cases\ has-snapshotted\ c\ p'$ )
  case  $True$ 
    then have  $msgs\ c\ i = msgs\ c'\ i$  using  $\langle i' \neq i \rangle\ RecvMarker\ assms$  by  $simp$ 
    then show  $?thesis$  using  $calculation$  by  $simp$ 
  next
    case  $no-snap: False$ 
    then have  $chan: channel\ i = Some\ (q, p)$ 
    by ( $metis\ Recv\ assms(3)\ distributed-system.can-occur-Recv\ distributed-system-axioms$ )

```

```

then show ?thesis
proof (cases q = p')
  case True
    then have msgs c' i = msgs c i @ [Marker]
      using no-snap RecvMarker ⟨i' ≠ i⟩ assms(1) chan by auto
    then show ?thesis
      by (metis append-self-conv2 calculation(2) hd-append2 list.sel(1) mes-
sage.simps(3))
    next
      case False
        then have msgs c' i = msgs c i using RecvMarker no-snap False chan
assms ⟨i' ≠ i⟩ by simp
        then show ?thesis using calculation by simp
      qed
    qed
  next
    case (Trans p' s'' s''')
      then show ?thesis using assms(1) ⟨msgs c' i ≠ Nil⟩ by auto
    next
      case (Send i' p' q' s'' s''' m'')
        have p' ≠ p
          using Recv Send assms(2) by auto
        then show ?thesis
          using Recv Send assms(1) assms(5) calculation(1) by auto
      next
        case (Recv i' p' q' s'' s''' m'')
          then have i' ≠ i using assms ⟨ev' = Recv i p q s' m⟩
            by (metis distributed-system.can-occur-Recv distributed-system-axioms event.sel(3)
next-recv option.inject prod.inject)
          have msgs c i = msgs c' i using msgs-unchanged-for-other-is Recv ⟨i' ≠ i⟩
assms(1) by auto
          then show ?thesis using ⟨msgs c' i ≠ Nil⟩ by simp
        qed
      moreover have states c p = states c' p using no-state-change-if-no-event assms
Recv by simp
      ultimately show ?thesis
        using Recv assms(3) can-occur-def list.case-eq-if by fastforce
    qed

```

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

```

assumes
  c ⊢ ev ⇔ c' and
  regular-event ev and
  ~ regular-event ev' and
  can-occur ev' c' and
  occurs-on ev ≠ occurs-on ev'
shows
  can-occur ev' c
proof –

```

```

let ?p = occurs-on ev
let ?q = occurs-on ev'
have isTrans ev  $\vee$  isSend ev  $\vee$  isRecv ev using assms by simp
moreover have isSnapshot ev'  $\vee$  isRecvMarker ev' using assms nonregular-event
by auto
ultimately show ?thesis
proof (elim disjE, goal-cases)
  case 1
  then show ?case
  using assms(1) assms(4) assms(5) event-can-go-back-if-no-sender by blast
next
  case 2
  then obtain s s' where Trans: ev = Trans ?p s s'
  by (metis event.collapse(1))
  obtain i r where RecvMarker: ev' = RecvMarker i ?q r
  using 2 by (metis event.collapse(5))
  have msgs c i = msgs c' i
  using 2(1) assms(1) assms(2) no-msg-change-if-no-event by blast
  moreover have can-occur ev' c' using assms by simp
  ultimately show ?thesis using can-occur-def RecvMarker
  by (metis (mono-tags, lifting) 2(2) event.case-eq-if event.distinct-disc(13)
event.distinct-disc(17) event.distinct-disc(19) event.distinct-disc(7) event.sel(10))
next
  case 3
  then have ev' = Snapshot ?q
  by (metis event.collapse(4))
  have  $\sim$  has-snapshotted c' ?q
  by (metis (mono-tags, lifting) 3(1) assms(4) can-occur-def event.case-eq-if
event.distinct-disc(11) event.distinct-disc(16) event.distinct-disc(6))
  then have  $\sim$  has-snapshotted c ?q
  using assms(1) assms(2) regular-event-preserves-process-snapshots by auto
  then show ?case unfolding can-occur-def using  $\langle$ ev' = Snapshot ?q $\rangle$ 
  by (metis (mono-tags, lifting) event.simps(29))
next
  case 4
  then have ev' = Snapshot ?q
  by (metis event.collapse(4))
  have  $\sim$  has-snapshotted c' ?q
  by (metis (mono-tags, lifting)  $\langle$ ev' = Snapshot (occurs-on ev') $\rangle$  assms(4)
can-occur-def event.simps(29))
  then have  $\sim$  has-snapshotted c ?q
  using assms(1) assms(2) regular-event-preserves-process-snapshots by auto
  then show ?case unfolding can-occur-def
  by (metis (mono-tags, lifting)  $\langle$ ev' = Snapshot (occurs-on ev') $\rangle$  event.simps(29))
next
  case 5
  then obtain i s u u' m where ev = Send i ?p s u u' m
  by (metis event.collapse(2))
  from 5 obtain i' r where ev' = RecvMarker i' ?q r

```

```

    by (metis event.collapse(5))
  then have pre: hd (msgs c' i') = Marker ∧ length (msgs c' i') > 0
    by (metis (mono-tags, lifting) assms(4) can-occur-def event.simps(30))
  have hd (msgs c i') = Marker ∧ length (msgs c i') > 0
  proof (cases i' = i)
    case False
    then have msgs c i' = msgs c' i'
      by (metis ⟨ev = Send i (occurs-on ev) s u u' m⟩ assms(1) assms(2)
event.sel(8) msgs-unchanged-for-other-is)
    then show ?thesis using pre by auto
  next
  case True
  then have msgs c' i' = msgs c i' @ [Msg m]
    by (metis ⟨ev = Send i (occurs-on ev) s u u' m⟩ assms(1) next-send)
  then have length (msgs c' i') > 1
    using pre by fastforce
  then have length (msgs c i') > 0
    by (simp add: ⟨msgs c' i' = msgs c i' @ [Msg m]⟩)
  then show ?thesis
    using ⟨msgs c' i' = msgs c i' @ [Msg m]⟩ pre by auto
  qed
  then show ?case unfolding can-occur-def using ⟨ev' = RecvMarker i' ?q r⟩
    by (metis (mono-tags, lifting) assms(4) can-occur-def event.simps(30))
  next
  case 6
  then obtain i s u u' m where ev = Recv i ?p s u u' m
    by (metis event.collapse(3))
  from 6 obtain i' r where ev' = RecvMarker i' ?q r
    by (metis event.collapse(5))
  then have i' ≠ i
  proof -
    have ?p ≠ ?q using assms by simp
    moreover have channel i = Some (s, ?p)
    by (metis ⟨ev = Recv i (occurs-on ev) s u u' m⟩ assms(1) distributed-system.can-occur-Recv
distributed-system-axioms happen-implies-can-occur)
    moreover have channel i' = Some (r, ?q)
    by (metis (mono-tags, lifting) ⟨ev' = RecvMarker i' (occurs-on ev') r⟩
assms(4) can-occur-def event.case-eq-if event.disc(5,10,15,20) event.sel(5,10,13))
    ultimately show ?thesis by auto
  qed
  then show ?case
  by (metis (mono-tags, lifting) 6(1) ⟨ev = Recv i (occurs-on ev) s u u' m⟩ ⟨ev' =
RecvMarker i' (occurs-on ev') r⟩ assms(1) assms(4) can-occur-def event.case-eq-if
event.distinct-disc(13) event.distinct-disc(17) event.distinct-disc(7) event.sel(10)
next-recv)
  qed
  qed

```

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

```

assumes
  states c p = states d p
  c ⊢ ev ↦ c' and
  d ⊢ ev ↦ d'
shows
  states d' p = states c' p
proof (cases occurs-on ev = p)
  case False
  then show ?thesis
  by (metis assms(1-3) distributed-system.no-state-change-if-no-event distributed-system-axioms)
next
  case True
  then show ?thesis
  using assms by (cases ev, auto)
qed

```

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

```

assumes
  ps c p = ps d p and
  states c p = states d p and
  c ⊢ ev ↦ c' and
  d ⊢ ev ↦ d'
shows
  ps d' p = ps c' p
proof (cases occurs-on ev = p)
  case False
  then show ?thesis
  using assms no-state-change-if-no-event by auto
next
  case True
  then show ?thesis
proof (cases ev)
  case (Snapshot q)
  then have p = q using True by auto
  then show ?thesis
  using Snapshot assms(2) assms(3) assms(4) by auto
next
  case (RecvMarker i q r)
  then have p = q using True by auto
  then show ?thesis
proof –
  have f1: ⋀c ca. ¬ c ⊢ ev ↦ ca ∨ ps c p = None ∨ ps c p = ps ca p
  using RecvMarker ⟨p = q⟩ by force
  have ⋀c ca. ps c p ≠ None ∨ ¬ c ⊢ ev ↦ ca ∨ ps ca p = Some (states c p)
  using RecvMarker ⟨p = q⟩ by force
  then show ?thesis
  using f1 by (metis (no-types) assms(1) assms(2) assms(3) assms(4))
qed
next

```

```

    case (Trans q s s')
    then have p = q
         using True by auto
    then show ?thesis
         using Trans assms(1) assms(3) assms(4) by auto
next
    case (Send i q r u u' m)
    then have p = q using True by auto
    then show ?thesis
         using Send assms(1) assms(3) assms(4) by auto
next
    case (Recv i q r u u' m)
    then have p = q using True by auto
    then show ?thesis
         using Recv assms(1) assms(3) assms(4) by auto
qed
qed

```

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

```

assumes
  msgs c i = msgs d i
  c ⊢ ev ↦ c' and
  d ⊢ ev ↦ d' and
  regular-event ev
shows
  msgs c' i = msgs d' i
proof –
  have isTrans ev ∨ isSend ev ∨ isRecv ev using assms
    by simp
  then show ?thesis
proof (elim disjE)
    assume isTrans ev
    then show ?thesis
      by (metis assms(1) assms(2) assms(3) isTrans-def next-trans)
  next
    assume isSend ev
    then obtain i' r s u u' m where ev = Send i' r s u u' m
      by (metis event.collapse(2))
    with assms show ?thesis by (cases i = i', auto)
  next
    assume isRecv ev
    then obtain i' r s u u' m where Recv: ev = Recv i' r s u u' m
      by (metis event.collapse(3))
    with assms show ?thesis by (cases i = i', auto)
qed
qed

```

**lemma** *Trans-msg*:

```

assumes

```

```

     $c \vdash ev \mapsto c'$  and
    isTrans ev
  shows
     $msgs\ c\ i = msgs\ c'\ i$ 
  using assms(1) assms(2) no-msg-change-if-no-event regular-event by blast

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 (rule ccontr)
  assume  $\sim ?P$ 
  have  $set\ (msgs\ c'\ i) \subseteq set\ (msgs\ c\ i)$ 
  proof (cases ev)
    case (Snapshot r)
      then have  $msgs\ c'\ i = msgs\ c\ i$  using  $\langle \sim ?P \rangle$  assms by simp
      then show ?thesis by auto
    next
      case (RecvMarker i' r s)
        then show ?thesis
        proof (cases has-snapshotted c r)
          case True
            then show ?thesis
            proof (cases  $i' = i$ )
              case True
                then have  $Marker \# msgs\ c'\ i = msgs\ c\ i$  using RecvMarker True assms
                by simp
              then show ?thesis
                by (metis set-subset-Cons)
            next
              case False
                then show ?thesis using RecvMarker True assms by simp
          qed
        next
          case no-snap: False
            have  $chan: channel\ i' = Some\ (s, r)$ 
            using RecvMarker assms(1) can-occur-def by auto
            then show ?thesis
            proof (cases  $i' = i$ )
              case True
                then have  $Marker \# msgs\ c'\ i = msgs\ c\ i$  using RecvMarker assms by
                simp
              then show ?thesis by (metis set-subset-Cons)
            next
              case False

```

```

    then have  $msgs\ c'\ i = msgs\ c\ i$  using  $\langle \sim\ ?P \rangle$  RecvMarker assms no-snap
  by simp
    then show ?thesis by simp
    qed
  qed
  next
    case (Trans  $r\ u\ u'$ )
    then show ?thesis using assms  $\langle \sim\ ?P \rangle$  by simp
  next
    case (Send  $i'\ r\ s\ u\ u'\ m'$ )
    then have  $i' \neq i$  using  $\langle \sim\ ?P \rangle$  can-occur-def assms by auto
    then have  $msgs\ c\ i = msgs\ c'\ i$  using  $\langle \sim\ ?P \rangle$  assms Send by simp
    then show ?thesis by simp
  next
    case (Recv  $i'\ r\ s\ u\ u'\ m'$ )
    then show ?thesis
    by (metis (no-types, lifting) assms(1) eq-iff local.next.simps(5) set-subset-Cons)
  qed
  moreover have  $\sim\ set\ (msgs\ c'\ i) \subseteq set\ (msgs\ c\ i)$  using assms by blast
  ultimately show False by simp
qed

```

**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 (rule ccontr)
    have  $occurs-on\ ev = p$ 
      using assms new-msg-in-set-implies-occurrence by blast
    assume  $\sim\ ?P$ 
    then have  $isTrans\ ev \vee isSend\ ev \vee isRecv\ ev$  by simp
    then have  $Marker \notin set\ (msgs\ c'\ i)$ 
    proof (elim disjE, goal-cases)
      case 1
        then obtain  $r\ u\ u'$  where  $ev = Trans\ r\ u\ u'$ 
          by (metis event.collapse(1))
        then show ?thesis
          using assms(1) assms(2) by auto
      next
        case 2
        then obtain  $i'\ r\ q\ u\ u'\ m$  where  $ev = Send\ i'\ r\ q\ u\ u'\ m$ 
          by (metis event.collapse(2))
        then show ?thesis
          by (metis (no-types, lifting) Un-iff assms(1) assms(2) empty-iff empty-set
            insert-iff list.set(2) message.distinct(1) next-send set-append)
    qed
  qed

```

```

next
  case 3
  then obtain  $i' r q u u' m$  where  $ev = Recv\ i' r q u u' m$ 
    by (metis event.collapse(3))
  then show ?thesis
    by (metis assms(1) assms(2) list.set-intros(2) next-recv)
qed
then show False using assms by simp
qed

```

**lemma** *RecvMarker-implies-Marker-in-set:*

```

assumes
   $c \vdash ev \mapsto c'$  and
   $ev = RecvMarker\ cid\ p\ q$ 
shows
   $Marker \in set\ (msgs\ c\ cid)$ 
by (metis (mono-tags, lifting) assms(1) assms(2) can-occur-def distributed-system.happen-implies-can-occur
distributed-system-axioms event.simps(30) list.set-sel(1) list.size(3) nat-less-le)

```

**lemma** *RecvMarker-given-channel:*

```

assumes
  isRecvMarker  $ev$  and
  getId  $ev = cid$  and
  channel  $cid = Some\ (p, q)$  and
  can-occur  $ev\ c$ 
shows
   $ev = RecvMarker\ cid\ q\ p$ 
by (metis (mono-tags, lifting) assms(1) assms(2) assms(3) assms(4) can-occur-def
event.case-eq-if event.collapse(5) event.distinct-disc(8,14,18,20) option.inject prod.inject)

```

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

```

assumes
  isRecv  $ev$  and
  getId  $ev = cid$  and
  channel  $cid = Some\ (p, q)$  and
  can-occur  $ev\ c$ 
shows
   $\exists s\ s' m. ev = Recv\ cid\ q\ p\ s\ s'\ m$ 
by (metis assms(1) assms(2) assms(3) assms(4) distributed-system.can-occur-Recv
distributed-system-axioms event.collapse(3) option.inject prod.inject)

```

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

```

assumes
   $c \vdash ev \mapsto c'$  and
   $\sim isRecv\ ev$ 
shows
   $fst\ (cs\ c\ cid) = fst\ (cs\ c'\ cid)$ 
proof (cases channel  $cid = None$ )
  case True

```

```

then show ?thesis
  using assms(1) no-cs-change-if-no-channel by auto
next
  case False
  then obtain p q where chan: channel cid = Some (p, q) by auto
  then show ?thesis
  proof (cases ev)
    case (Snapshot r)
      with Snapshot assms chan show ?thesis by (cases r = q, auto)
    next
      case (RecvMarker cid' r s)
        then show ?thesis
        proof (cases has-snapshotted c r)
          case True
            with assms RecvMarker chan show ?thesis by (cases cid' = cid, auto)
          next
            case no-snap: False
              then show ?thesis
              proof (cases cid' = cid)
                case True
                  then show ?thesis using RecvMarker assms chan by auto
                next
                  case False
                    with assms RecvMarker chan no-snap show ?thesis by (cases r = q, auto)
                  qed
                qed
              next
                case False
                  with assms RecvMarker chan no-snap show ?thesis by (cases r = q, auto)
                qed
              qed
            next
              case False
                with assms RecvMarker chan no-snap show ?thesis by (cases r = q, auto)
              qed
            qed
          next
            case False
              with assms RecvMarker chan no-snap show ?thesis by (cases r = q, auto)
            qed
          qed
        qed
      next
        case (Trans r u u')
          then show ?thesis using assms by auto
        next
          case (Send r s u u')
            then show ?thesis using assms by auto
          qed (metis assms(2) isRecv-def)
        qed
      qed
    qed
  qed

```

**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
proof (rule ccontr)
  assume  $\sim$  ev = RecvMarker cid q p
  then show False
  proof (cases isRecvMarker ev)
    case True
      then obtain cid' s r where RecvMarker: ev = RecvMarker cid' s r by (meson)

```

```

isRecvMarker-def)
  have  $cid \neq cid'$ 
  proof (rule ccontr)
    assume  $\sim cid \neq cid'$ 
    then show False
      using  $\langle ev = RecvMarker\ cid'\ s\ r \rangle \langle ev \neq RecvMarker\ cid\ q\ p \rangle\ assms(1)$ 
  assms(2) can-occur-def by auto
  qed
  then have  $snd\ (cs\ c'\ cid) \neq Done$ 
  proof (cases has-snapshotted c s)
    case True
      then show ?thesis using RecvMarker assms  $\langle cid \neq cid' \rangle$  by simp
    next
      case False
        with RecvMarker assms  $\langle cid \neq cid' \rangle$  show ?thesis by (cases s = q, auto)
  qed
  then show False using assms by auto
next
case False
  then have  $isSnapshot\ ev \vee isTrans\ ev \vee isSend\ ev \vee isRecv\ ev$ 
  using event.exhaust-disc by blast
  then have  $snd\ (cs\ c'\ cid) \neq Done$ 
  proof (elim disjE, goal-cases)
    case 1
      then obtain r where Snapshot:  $ev = Snapshot\ r$ 
        by (meson isSnapshot-def)
      with assms show ?thesis by (cases q = r, auto)
    next
      case 2
        then obtain r u u' where  $ev = Trans\ r\ u\ u'$ 
          by (meson isTrans-def)
        then show ?case using assms by auto
    next
      case 3
        then obtain cid' r s u u' m where  $ev = Send\ cid'\ r\ s\ u\ u'\ m$ 
          by (meson isSend-def)
        then show ?thesis using assms by auto
    next
      case 4
        then obtain cid' r s u u' m where Recv:  $ev = Recv\ cid'\ r\ s\ u\ u'\ m$ 
          by (meson isRecv-def)
        show ?thesis
        using Recv assms proof (cases cid = cid')
          case True
            then have  $snd\ (cs\ c\ cid) = NotStarted \vee snd\ (cs\ c\ cid) = Recording$ 
              using assms(3) recording-state.exhaust by blast
            then show ?thesis
          proof (elim disjE, goal-cases)
            case 1

```

```

    then have  $snd (cs\ c'\ cid') = NotStarted$ 
      using  $True\ Recv\ assms(1)$  by auto
    then show  $?case$  using  $True$  by auto
  next
    case 2
    then have  $snd (cs\ c'\ cid') = Recording$ 
      using  $True\ Recv\ assms(1)$  by auto
    then show  $?case$  using  $True$  by auto
  qed
qed auto
qed
then show  $False$  using  $assms$  by auto
qed
qed

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
     $snd (cs\ c'\ cid) \neq NotStarted$ 
using  $assms$  proof (cases  $ev$ )
  case (Snapshot  $r$ )
  then show  $?thesis$ 
    by (metis  $assms(1)\ assms(2)\ next-snapshot\ recording-state.simps(2)\ sndI$ )
  next
  case (RecvMarker  $cid'\ r\ s$ )
  then show  $?thesis$ 
  proof (cases  $has-snapshotted\ c\ r$ )
    case True
    with  $RecvMarker\ assms$  show  $?thesis$  by (cases  $cid = cid'$ , auto)
  next
  case no-snap: False
  then show  $?thesis$ 
  proof (cases  $cid = cid'$ )
    case True
    then show  $?thesis$  using  $RecvMarker\ assms$  by auto
  next
  case False
  with  $RecvMarker\ assms\ no-snap$  show  $?thesis$  by (cases  $s = p$ , auto)
  qed
qed
next
  case (Recv  $cid'\ r\ s\ u\ u'\ m$ )
  then have  $snd (cs\ c\ cid) = Recording \vee snd (cs\ c\ cid) = Done$ 
    using  $assms(2)\ recording-state.exhaust$  by blast
  then show  $?thesis$ 
  proof (elim  $disjE$ , goal-cases)

```

```

case 1
then show ?thesis
  by (metis (no-types, lifting) Recv assms(1) eq-snd-iff next-recv recording-state.distinct(1))
next
case 2
with Recv assms show ?thesis by (cases cid = cid', auto)
qed
qed auto

```

**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  $cid = \text{Some } (p, q)$ 
shows
   $snd\ (cs\ c\ cid) = \text{Recording} \wedge (\exists p\ q\ u\ u'\ m. ev = \text{Recv}\ cid\ q\ p\ u\ u'\ m)$ 
proof -
  have oc-on:  $occurs\text{-on}\ ev = q$ 
  proof -
    obtain nn :: ('p, 's, 'm) event  $\Rightarrow$  nat and aa :: ('p, 's, 'm) event  $\Rightarrow$  'p and
    aaa :: ('p, 's, 'm) event  $\Rightarrow$  'p and bb :: ('p, 's, 'm) event  $\Rightarrow$  's and bba :: ('p, 's,
    'm) event  $\Rightarrow$  's and cc :: ('p, 's, 'm) event  $\Rightarrow$  'm where
    f1:  $\forall e. (\neg isRecv\ e \vee e = \text{Recv}\ (nn\ e)\ (aa\ e)\ (aaa\ e)\ (bb\ e)\ (bba\ e)\ (cc\ e)) \wedge$ 
    ( $isRecv\ e \vee (\forall n\ a\ aa\ b\ ba\ c. e \neq \text{Recv}\ n\ a\ aa\ b\ ba\ c)$ )
    by (metis isRecv-def)
    then have f2:  $c \vdash \text{Recv}\ (nn\ ev)\ (aa\ ev)\ (aaa\ ev)\ (bb\ ev)\ (bba\ ev)\ (cc\ ev) \mapsto c'$ 
    by (metis (no-types) assms(2) local.step\ same-cs-if-not-recv)
    have f3:  $\forall x0\ x1\ x7\ x8. (x0 \neq x7 \longrightarrow cs\ (x8::('p, 's, 'm)\ \text{configuration})\ x0 =$ 
     $cs\ (x1::('p, 's, -)\ \text{configuration})\ x0) = (x0 = x7 \vee cs\ x8\ x0 = cs\ x1\ x0)$ 
    by auto
    have f4:  $\forall x0\ x1\ x7\ x8. (x7 \neq x0 \longrightarrow msgs\ (x1::('p, 's, 'm)\ \text{configuration})\ x0 =$ 
     $msgs\ (x8::('p, 's, -)\ \text{configuration})\ x0) = (x7 = x0 \vee msgs\ x1\ x0 = msgs\ x8\ x0)$ 
    by auto
    have  $\forall x0\ x1\ x6\ x8. (x0 \neq x6 \longrightarrow states\ (x1::('p, 's, 'm)\ \text{configuration})\ x0 =$ 
     $states\ (x8::(-, -, 'm)\ \text{configuration})\ x0) = (x0 = x6 \vee states\ x1\ x0 = states\ x8\ x0)$ 
    by fastforce
    then have can-occur ( $\text{Recv}\ (nn\ ev)\ (aa\ ev)\ (aaa\ ev)\ (bb\ ev)\ (bba\ ev)\ (cc\ ev)$ ) c
     $\wedge$  states c ( $aa\ ev$ ) = bb ev  $\wedge$  states c' ( $aa\ ev$ ) = bba ev  $\wedge$  ( $\forall a. a = aa\ ev \vee states$ 
     $c'\ a = states\ c\ a) \wedge msgs\ c\ (nn\ ev) = \text{Msg}\ (cc\ ev)\ \# msgs\ c'\ (nn\ ev) \wedge (\forall n. nn$ 
     $ev = n \vee msgs\ c'\ n = msgs\ c\ n) \wedge (\forall a. ps\ c\ a = ps\ c'\ a) \wedge (\forall n. n = nn\ ev \vee cs$ 
     $c\ n = cs\ c'\ n) \wedge (if\ snd\ (cs\ c\ (nn\ ev)) = \text{Recording}\ then\ cs\ c'\ (nn\ ev) = (fst\ (cs\ c$ 
     $(nn\ ev))\ @\ [cc\ ev],\ \text{Recording})\ else\ cs\ c\ (nn\ ev) = cs\ c'\ (nn\ ev))$ 
    using f4 f3 f2 by force
    then show ?thesis
    using f1 by (metis (no-types) Pair-inject\ assms(2) assms(3) can-occur-Recv
    event.sel(3) local.step\ option.sel\ same-cs-if-not-recv)
  qed
  have isRecv ev (is ?P)
  proof (rule\ ccontr)

```

```

  assume  $\sim ?P$ 
  then have  $\text{fst } (cs \ c \ cid) = \text{fst } (cs \ c' \ cid)$  by (metis local.step same-cs-if-not-recv)
  then show False using assms by simp
qed
then obtain  $cid' \ r \ s \ u \ u' \ m$  where  $\text{Recv: } ev = \text{Recv } cid' \ r \ s \ u \ u' \ m$  by (meson
isRecv-def)
have  $cid = cid'$ 
proof (rule ccontr)
  assume  $\sim cid = cid'$ 
  then have  $\text{fst } (cs \ c \ cid) = \text{fst } (cs \ c' \ cid)$  using Recv step by auto
  then show False using assms by simp
qed
moreover have  $\text{snd } (cs \ c \ cid) = \text{Recording}$ 
proof (rule ccontr)
  assume  $\sim \text{snd } (cs \ c \ cid) = \text{Recording}$ 
  then have  $\text{fst } (cs \ c \ cid) = \text{fst } (cs \ c' \ cid)$  using Recv step  $\langle cid = cid' \rangle$  by auto
  then show False using assms by simp
qed
ultimately show ?thesis using Recv by simp
qed

```

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

```

  assumes
     $c \vdash ev \mapsto c'$  and
    has-snapshotted  $c \ p$  and
     $\text{Marker} \notin \text{set } (msgs \ c \ cid)$  and
     $\text{channel } cid = \text{Some } (p, q)$ 
  shows
     $\text{Marker} \notin \text{set } (msgs \ c' \ cid)$ 
proof (cases ev)
  case (Snapshot  $r$ )
  then have  $r \neq p$ 
    using assms(1) assms(2) can-occur-def by auto
  then have  $msgs \ c \ cid = msgs \ c' \ cid$  using assms Snapshot by simp
  then show ?thesis using assms by simp
next
  case (RecvMarker  $cid' \ r \ s$ )
  have  $cid \neq cid'$ 
  proof (rule ccontr)
    assume  $\sim cid \neq cid'$ 
    moreover have can-occur  $ev \ c$  using happen-implies-can-occur assms by blast
    ultimately have  $\text{Marker} : \text{set } (msgs \ c \ cid)$  using can-occur-def RecvMarker
      by (metis (mono-tags, lifting) assms(1) event.simps(30) hd-in-set list.size(3))
    then show False using assms by simp
  qed
  then have  $msgs \ c \ cid = msgs \ c' \ cid$ 
  proof (cases  $r = p$ )
    case True

```

```

    then show ?thesis
    using RecvMarker ⟨cid ≠ cid'⟩ assms(1) assms(2) msgs-unchanged-if-snapshotted-RecvMarker-for-other-is
  by blast
  next
    case False
    with RecvMarker ⟨cid ≠ cid'⟩ step assms show ?thesis by (cases has-snapshotted
  c r, auto)
  qed
  then show ?thesis using assms by simp
next
  case (Trans r u u')
  then show ?thesis using assms by auto
next
  case (Send cid' r s u u' m)
  with assms Send show ?thesis by (cases cid = cid', auto)
next
  case (Recv cid' r s u u' m)
  with assms Recv show ?thesis by (cases cid = cid', auto)
qed

lemma same-messages-if-no-occurrence:
  assumes
    c ⊢ ev ↦ c' and
    ~ occurs-on ev = p and
    ~ occurs-on ev = q and
    channel cid = Some (p, q)
  shows
    msgs c cid = msgs c' cid ∧ cs c cid = cs c' cid
proof (cases ev)
  case (Snapshot r)
  then show ?thesis using assms by auto
next
  case (RecvMarker cid' r s)
  have cid ≠ cid'
  by (metis RecvMarker-given-channel assms(1) assms(3) assms(4) RecvMarker
  event.sel(5,10) happen-implies-can-occur isRecvMarker-def)
  have ∄ a. channel cid = Some (r, q)
  using assms(2) assms(4) RecvMarker by auto
  with RecvMarker assms ⟨cid ≠ cid'⟩ show ?thesis by (cases has-snapshotted c
  r, auto)
next
  case (Trans r u u')
  then show ?thesis using assms by auto
next
  case (Send cid' r s u u' m)
  then have cid ≠ cid'
  by (metis (mono-tags, lifting) Pair-inject assms(1) assms(2) assms(4) can-occur-def
  event.sel(2) event.simps(27) happen-implies-can-occur option.inject)
  then show ?thesis using assms Send by simp

```

```

next
  case (Recv cid' r s u u' m)
  then have cid  $\neq$  cid'
    by (metis assms(1) assms(3) assms(4) distributed-system.can-occur-Recv distributed-system.happen-implies-can-occur distributed-system-axioms event.sel(3) option.inject prod.inject)
    then show ?thesis using assms Recv by simp
  qed

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:  $\llbracket c \vdash ev \mapsto c'; \text{trace } c' \ t \ c'' \rrbracket$ 
     $\implies \text{trace } c \ (ev \ \# \ t) \ c''$ 

```

### 2.1 Properties of traces

```

lemma trace-trans:

```

```

  shows

```

```

     $\llbracket \text{trace } c \ t \ c';$ 
       $\text{trace } c' \ t' \ c''$ 
     $\rrbracket \implies \text{trace } c \ (t \ @ \ t') \ c''$ 

```

```

proof (induct  $c \ t \ c'$  rule:trace.induct)

```

```

  case tr-init

```

```

    then show ?case by simp

```

```

next

```

```

  case tr-step

```

```

    then show ?case using trace.tr-step by auto

```

qed

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

**assumes**

$trace\ c\ (ev\ \#)\ t\ c'$

**shows**

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

**using** *assms trace.simps* **by** *blast*

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

**shows**

$trace\ c\ (t\ @\ [ev])\ c' \implies \exists\ c''.\ trace\ c\ t\ c''\ \wedge\ c''\ \vdash\ ev\ \mapsto\ c'$

**proof** (*induct t arbitrary: c*)

**case** *Nil*

**then show** *?case*

**by** (*metis (mono-tags, lifting) append-Nil distributed-system.trace.simps distributed-system-axioms list.discI list.sel(1) list.sel(3)*)

**next**

**case** (*Cons ev' t*)

**then obtain** *d* **where**  $c\ \vdash\ ev'\ \mapsto\ d$  **and**  $trace\ d\ (t\ @\ [ev])\ c'$  **using** *trace-decomp-head* **by** *force*

**then obtain** *d'* **where**  $tr:\ trace\ d\ t\ d'$  **and**  $d'\ \vdash\ ev\ \mapsto\ c'$  **using** *Cons.hyps* **by** *blast*

**moreover have**  $trace\ c\ (ev'\ \#)\ t\ d'$  **using** *step tr trace.tr-step* **by** *simp*

**ultimately show** *?case* **by** *auto*

qed

**lemma** *trace-snoc*:

**assumes**

$trace\ c\ t\ c'$  **and**

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

**shows**

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

**using** *assms(1) assms(2) tr-init tr-step trace-trans* **by** *auto*

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

$\llbracket\ trace\ c\ t\ c';$

$(\bigwedge\ c.\ P\ c\ \llbracket\ c);$

$(\bigwedge\ c\ t\ c'\ ev\ c''.\ trace\ c\ t\ c' \implies P\ c\ t\ c' \implies c'\ \vdash\ ev\ \mapsto\ c'' \implies P\ c\ (t\ @\ [ev])$

$c'')$

$\rrbracket \implies P\ c\ t\ c'$

**proof** (*induct t arbitrary: c' rule:rev-induct*)

**case** *Nil*

**then show** *?case*

**using** *distributed-system.trace.cases distributed-system-axioms* **by** *blast*

**next**

**case** (*snoc ev t*)

**then obtain** *c''* **where**  $trace\ c\ t\ c''\ c''\ \vdash\ ev\ \mapsto\ c'$  **using** *trace-decomp-tail* **by** *blast*

then show ?case using snoc by simp  
qed

lemma trace-and-start-determines-end:

shows

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

proof (induct c t c' arbitrary: d' rule:trace-rev-induct)

case tr-rev-init

then show ?case using trace.cases by fastforce

next

case (tr-rev-step c t c' ev c'')

then obtain d'' where  $\text{trace } c \ t \ d'' \ d'' \vdash \text{ev} \mapsto d'$  using trace-decomp-tail by blast

then show ?case using tr-rev-step state-and-event-determine-next by simp

qed

lemma suffix-split-trace:

shows

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

proof (induct t arbitrary: c)

case Nil

then have  $t' = []$  by simp

then have  $\text{trace } c' \ t' \ c'$  using tr-init by simp

then show ?case by blast

next

case (Cons ev t'')

from Cons.prem1 have  $q: \text{suffix } t' \ t'' \vee t' = \text{ev} \ \# \ t''$  by (meson suffix-Cons)

thus ?case

proof (cases suffix t' t'')

case True

then show ?thesis using Cons.hyps Cons.prem1(1) trace.simps by blast

next

case False

hence  $t' = \text{ev} \ \# \ t''$  using q by simp

thus ?thesis using Cons.hyps Cons.prem1 by blast

qed

qed

lemma prefix-split-trace:

fixes

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

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

shows

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

proof (induct t rule:rev-induct)

```

    case Nil
  then show ?case by simp
next
case (snoc ev t'')
from snoc.prem1 have q: prefix t' t''  $\vee$  t' = t'' @ [ev] by auto
thus ?case
proof (cases prefix t' t'')
  case True
  thus ?thesis using trace-decomp-tail using snoc.hyps snoc.prem1 trace.simps
by blast
next
case False
thus ?thesis using q snoc.prem1 by fast
qed
qed

```

**lemma** *split-trace*:

**shows**

```

  [ trace c t c';
    t = t' @ t''
  ]  $\implies$   $\exists$  c''. trace c t' c''  $\wedge$  trace c'' t'' c'

```

**proof** (*induct t'' arbitrary: t'*)

case Nil

then show ?case using tr-init by auto

next

case (Cons ev t'')

obtain c'' where p: trace c (t' @ [ev]) c''

using Cons.prem1 prefix-split-trace rotate1.simps(2) by force

then have trace c'' t'' c'

using Cons.hyps Cons.prem1 trace-and-start-determines-end by force

then show ?case

by (meson distributed-system.tr-step distributed-system.trace-decomp-tail distributed-system-axioms p)

qed

## 2.2 Describing intermediate configurations

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

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

**definition** *trace-rel* **where**

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

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

**shows**

```

  single-valued R  $\implies$  x  $\in$  Domain R
   $\implies$  (x, construct-fun-from-rel R x)  $\in$  R

```

**unfolding** *construct-fun-from-rel-def*

by (rule theI') (auto simp add: single-valued-def)

**lemma** *single-valued-trace-rel*:  
**shows**  
*single-valued trace-rel*  
**proof** (*rule single-valuedI*)  
**fix**  $x\ y\ y'$   
**assume**  $asm: (x, y) \in trace-rel\ (x, y') \in trace-rel$   
**then obtain**  $x'\ t$  **where**  $x = (x', t)$   
**by** (*meson surj-pair*)  
**then have**  $trace\ x'\ t\ y\ trace\ x'\ t\ y'$   
**using**  $asm\ trace-rel-def$  **by** *auto*  
**then show**  $y = y'$   
**using** *trace-and-start-determines-end* **by** *blast*  
**qed**

**definition** *run-trace* **where**  
 $run-trace \equiv construct-fun-from-rel\ trace-rel$

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

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

**lemma** *s-is-partial-execution*:  
**shows**  
 $s\ c\ t\ i = run-trace\ (c,\ take\ i\ t)$   
**unfolding**  $s-def\ run-trace-def$   
*construct-fun-from-rel-def trace-rel-def*  
**by** *auto*

**lemma** *exists-trace-for-any-i*:  
**assumes**  
 $\exists c'.\ trace\ c\ t\ c'$   
**shows**  
 $trace\ c\ (take\ i\ t)\ (s\ c\ t\ i)$   
**proof** –  
**have**  $prefix\ (take\ i\ t)\ t$  **using** *take-is-prefix* **by** *auto*  
**then obtain**  $c''$  **where**  $tr: trace\ c\ (take\ i\ t)\ c''$  **using** *assms prefix-split-trace* **by**  
*blast*  
**then show** *?thesis*  
**proof** –  
**have**  $((c,\ take\ i\ t),\ s\ c\ t\ i) \in trace-rel$   
**unfolding**  $s-def\ trace-rel-def\ construct-fun-from-rel-def$   
**by** (*metis case-prod-conv distributed-system.trace-and-start-determines-end distributed-system-axioms mem-Collect-eq the-equality tr*)  
**then show** *?thesis* **by** (*simp add: trace-rel-def*)  
**qed**  
**qed**

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

**assumes**

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

$i \leq j$

**shows**

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

**proof** –

**have**  $\text{trace } c \ (take \ j \ t) \ (s \ c \ t \ j)$  **using** *exists-trace-for-any-i* **assms** **by** *simp*

**from**  $\langle j \geq i \rangle$  **have**  $take \ j \ t = take \ i \ t \ @ \ (take \ (j - i) \ (drop \ i \ t))$

**by** (*metis le-add-diff-inverse take-add*)

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

**by** (*metis (no-types, lifting) assms(1) exists-trace-for-any-i split-trace trace-and-start-determines-end*)

**then show** *?thesis* **by** *simp*

**qed**

**lemma** *step-Suc*:

**assumes**

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

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

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

**proof** –

**have** *ex-trace*:  $\text{trace } (s \ c \ t \ i) \ (take \ (Suc \ i - i) \ (drop \ i \ t)) \ (s \ c \ t \ (Suc \ i))$

**using** *exists-trace-for-any-i-j le-less valid* **by** *blast*

**moreover have**  $Suc \ i - i = 1$  **by** *auto*

**moreover have**  $take \ 1 \ (drop \ i \ t) = [t \ ! \ i]$

**by** (*metis <Suc i - i = 1> assms(1) hd-drop-conv-nth le-add-diff-inverse lessI nat-less-le same-append-eq take-add take-hd-drop*)

**ultimately show** *?thesis*

**by** (*metis list.discI trace.simps trace-decomp-head*)

**qed**

## 2.3 Trace-related lemmas

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

**assumes**

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

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

**shows**

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

**using** *assms snapshot-state-unchanged* **by** (*induct c t c', auto*)

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

**shows**

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

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

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

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

```

proof (induct c t c' rule:trace-rev-induct)
  case (tr-rev-init c)
  then show ?case by simp
next
  case (tr-rev-step c t c' ev c'')
  then have states c' p = st
  proof -
    have  $\nexists$  ev. ev : set t  $\wedge$  regular-event ev  $\wedge$  occurs-on ev = p
    using tr-rev-step by auto
    then show ?thesis using tr-rev-step by blast
  qed
  then show ?case
  using tr-rev-step no-state-change-if-no-event no-state-change-if-nonregular-event
  by auto
qed

```

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

```

assumes
  trace c t c' and
  m : set (msgs c i) and
  m  $\notin$  set (msgs c' i) and
  channel i = Some (q, p)
shows
   $\exists$  ev  $\in$  set t. ( $\exists$  p q. ev = RecvMarker i p q  $\wedge$  m = Marker)  $\vee$  ( $\exists$  p q s s' m'. ev
= Recv i q p s s' m'  $\wedge$  m = Msg m')
proof (rule ccontr)
  assume  $\sim$  ( $\exists$  ev  $\in$  set t. ( $\exists$  p q. ev = RecvMarker i p q  $\wedge$  m = Marker)  $\vee$  ( $\exists$  p q
s s' m'. ev = Recv i q p s s' m'  $\wedge$  m = Msg m')) (is ?P)
  have  $\llbracket$  trace c t c'; m : set (msgs c i); ?P  $\rrbracket \implies$  m : set (msgs c' i)
  proof (induct c t c' rule:trace-rev-induct)
    case (tr-rev-init c)
    then show ?case by simp
  next
    case (tr-rev-step c t d ev c')
    then have m-in-set: m : set (msgs d i)
    using tr-rev-step by auto
    then show ?case
    proof (cases ev)
      case (Snapshot r)
      then show ?thesis
    using distributed-system.message-must-be-delivered-2 distributed-system-axioms
m-in-set tr-rev-step.hyps(3) by blast
  next
    case (RecvMarker i' r s)
    then show ?thesis
    proof (cases m = Marker)
      case True
      then have i'  $\neq$  i using tr-rev-step RecvMarker by simp
      then show ?thesis

```

```

      using RecvMarker m-in-set message-must-be-delivered-2 tr-rev-step.hyps(3)
by blast
  next
    case False
    then show ?thesis
      using RecvMarker tr-rev-step.hyps(3) m-in-set message-must-be-delivered-2
by blast
  qed
next
  case (Trans r u u')
  then show ?thesis
    using tr-rev-step.hyps(3) m-in-set by auto
next
  case (Send i' r s u u' m')
  then show ?thesis
    using distributed-system.message-must-be-delivered-2 distributed-system-axioms
m-in-set tr-rev-step.hyps(3) by blast
  next
    case (Recv i' r s u u' m')
    then show ?thesis
    proof (cases Msg m' = m)
      case True
      then have i' ≠ i using Recv tr-rev-step by auto
      then show ?thesis
        using Recv m-in-set tr-rev-step(3) by auto
    next
      case False
      then show ?thesis
      by (metis Recv event.distinct(17) event.inject(3) m-in-set message-must-be-delivered-2
tr-rev-step.hyps(3))
    qed
  qed
  then have m : set (msgs c' i) using assms ⟨?P⟩ by auto
  then show False using assms by simp
qed

```

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

```

  assumes
    trace c t c' and
    Marker : set (msgs c i) and
    Marker ∉ set (msgs c' i) and
    channel i = Some (p, q)
  shows
    ∃ ev ∈ set t. (∃ p q. ev = RecvMarker i p q)
  proof -
    show ∃ ev ∈ set t. (∃ p q. ev = RecvMarker i p q)
      using assms message-must-be-delivered-2-trace by fast
  qed

```

**lemma** *snapshot-stable*:  
**shows**  
 $\llbracket \text{trace } c \ t \ c'; \text{ has-snapshotted } c \ p \rrbracket \implies \text{has-snapshotted } c' \ p$   
**proof** (*induct*  $c \ t \ c'$  *rule:trace-rev-induct*)  
**case** (*tr-rev-init*  $c$ )  
**then show** *?case* **by** *blast*  
**next**  
**case** (*tr-rev-step*  $c \ t \ c' \ \text{ev } c''$ )  
**then show** *?case*  
**proof** (*cases*  $\text{ev}$ )  
**case** (*Snapshot*  $q$ )  
**then have**  $p \neq q$  **using** *tr-rev-step next-snapshot can-occur-def* **by** *auto*  
**then show** *?thesis* **using** *Snapshot tr-rev-step* **by** *auto*  
**next**  
**case** (*RecvMarker*  $i \ q \ r$ )  
**with** *tr-rev-step* **show** *?thesis*  
**by** (*cases*  $p = q$ ; *auto*)  
**qed** *simp-all*  
**qed**

**lemma** *snapshot-stable-2*:  
**shows**  
 $\text{trace } c \ t \ c' \implies \sim \text{has-snapshotted } c' \ p \implies \sim \text{has-snapshotted } c \ p$   
**using** *snapshot-stable* **by** *blast*

**lemma** *no-markers-if-all-snapshotted*:  
**shows**  
 $\llbracket \text{trace } c \ t \ c'; \forall p. \text{has-snapshotted } c \ p; \text{Marker} \notin \text{set } (\text{msgs } c \ i) \rrbracket \implies \text{Marker} \notin \text{set } (\text{msgs } c' \ i)$   
**proof** (*induct*  $c \ t \ c'$  *rule:trace-rev-induct*)  
**case** (*tr-rev-init*  $c$ )  
**then show** *?case* **by** *simp*  
**next**  
**case** (*tr-rev-step*  $c \ t \ c' \ \text{ev } c''$ )  
**have** *all-snapshotted*:  $\forall p. \text{has-snapshotted } c' \ p$  **using** *snapshot-stable tr-rev-step*  
**by** *auto*  
**have** *no-marker*:  $\text{Marker} \notin \text{set } (\text{msgs } c' \ i)$  **using** *tr-rev-step* **by** *blast*  
**then show** *?case*  
**proof** (*cases*  $\text{ev}$ )  
**case** (*Snapshot*  $r$ )  
**then show** *?thesis* **using** *can-occur-def tr-rev-step all-snapshotted* **by** *auto*  
**next**  
**case** (*RecvMarker*  $i' \ r \ s$ )  
**have**  $i' \neq i$

```

proof (rule ccontr)
  assume  $\sim i' \neq i$ 
  then have Marker : set (msgs c i)
  using can-occur-def RecvMarker tr-rev-step RecvMarker-implies-Marker-in-set
by blast
  then show False using tr-rev-step by simp
  qed
  then show ?thesis using tr-rev-step all-snapshotted no-marker RecvMarker by
  auto
next
  case (Trans p s s')
  then show ?thesis using tr-rev-step no-marker by auto
next
  case (Send i' r s u u' m)
  then show ?thesis
  proof (cases i' = i)
    case True
    then have msgs c'' i = msgs c' i @ [Msg m] using tr-rev-step Send by auto
    then show ?thesis using no-marker by auto
  next
  case False
  then show ?thesis using Send tr-rev-step no-marker by auto
  qed
next
  case (Recv i' r s u u' m)
  then show ?thesis
  proof (cases i = i')
    case True
    then have msgs c'' i = tl (msgs c' i) using tr-rev-step Recv by auto
    then show ?thesis using no-marker by (metis list.sel(2) list.set-sel(2))
  next
  case False
  then show ?thesis using Recv tr-rev-step no-marker by auto
  qed
qed
qed

```

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

**shows**

```

  [ trace c t c';
    list-all ( $\lambda ev. \text{occurs-on } ev \neq \text{occurs-on } ev'$ ) t;
    can-occur ev' c
  ]  $\implies$  can-occur ev' c'

```

**proof** (induct c t c' rule:trace-rev-induct)

**case** tr-rev-init

**then show** ?case **by** blast

**next**

**case** tr-rev-step

**then show** ?case **using** event-stays-valid-if-no-occurrence **by** auto

qed

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

**shows**

$\llbracket$  *trace*  $c\ t\ c'$ ;  
  *list-all*  $(\lambda ev. \text{occurs-on } ev \neq \text{occurs-on } ev')$   $t$ ;  
  *can-occur*  $ev'\ c'$ ;  
   $\sim$  *isRecvMarker*  $ev'$ ;  
  *list-all*  $(\lambda ev. \sim \text{isSend } ev)$   $t$   
 $\rrbracket \implies \text{can-occur } ev'\ c$

**proof** (*induct*  $c\ t\ c'$  *rule:trace-rev-induct*)

**case** *tr-rev-init*

**then show** *?case* **by** *blast*

**next**

**case** *tr-rev-step*

**then show** *?case* **using** *event-can-go-back-if-no-sender* **by** *auto*

qed

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

**assumes**

*trace*  $c\ t\ c'$  **and**  
 $t \neq []$  **and**  
*snd*  $(cs\ c\ cid) \neq \text{Done}$  **and**  
*snd*  $(cs\ c'\ cid) = \text{Done}$  **and**  
*channel*  $cid = \text{Some } (p, q)$

**shows**

*RecvMarker*  $cid\ q\ p \in \text{set } t$

**proof** (*rule* *ccontr*)

**assume**  $\sim \text{RecvMarker } cid\ q\ p \in \text{set } t$

**moreover have**  $\llbracket \text{trace } c\ t\ c'; \sim \text{RecvMarker } cid\ q\ p \in \text{set } t; \text{snd } (cs\ c\ cid) \neq \text{Done}; \text{channel } cid = \text{Some } (p, q) \rrbracket$   
 $\implies \text{snd } (cs\ c'\ cid) \neq \text{Done}$

**proof** (*induct*  $t$  *arbitrary: c'* *rule:rev-induct*)

**case** *Nil*

**then show** *?case*

**by** (*metis* *list.discI* *trace.simps*)

**next**

**case**  $(\text{snoc } ev\ t)$

**then obtain**  $d$  **where** *ind: trace*  $c\ t\ d$  **and** *step: d*  $\vdash ev \mapsto c'$

**using** *trace-decomp-tail* **by** *blast*

**then have** *snd*  $(cs\ d\ cid) \neq \text{Done}$

**proof** –

**have**  $\sim \text{RecvMarker } cid\ q\ p \in \text{set } t$

**using** *snoc.premis(2)* **by** *auto*

**then show** *?thesis* **using** *snoc ind* **by** *blast*

qed

**then show** *?case*

**using** *done-only-from-recv-marker* *local.step* *snoc.premis(2)* *snoc.premis(4)* **by** *auto*

qed  
ultimately have  $snd (cs\ c'\ cid) \neq Done$  using *assms* by *blast*  
then show *False* using *assms* by *simp*  
qed

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 (induct *t* arbitrary:*c'* rule:rev-induct)  
case *Nil*  
then show ?*case*  
by (metis *list.discI trace.simps*)  
next  
case (snoc *ev t*)  
then obtain *d* where *tr*:  $trace\ c\ t\ d$  and *step*:  $d \vdash ev \mapsto c'$   
using *trace-decomp-tail* by *blast*  
then have  $snd\ (cs\ d\ cid) \neq NotStarted$  using *snoc* by *blast*  
then show ?*case* using *cs-not-not-started-stable snoc step* by *blast*  
qed

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 (induct *i*)  
case 0  
then show ?*case*  
by (metis *assms exists-trace-for-any-i no-msgs-if-no-channel take0 tr-init trace-and-start-determines-end*)  
next  
case (Suc *n*)  
have *f*:  $trace\ (s\ init\ t\ n)\ (take\ ((Suc\ n) - n)\ (drop\ n\ t))\ (s\ init\ t\ (Suc\ n))$   
using *exists-trace-for-any-i-j order-le-less trace assms* by *blast*  
then show ?*case*  
proof (cases  $drop\ n\ t = Nil$ )  
case *True*  
then show ?*thesis* using *Suc.hyps Suc.prem*  
by (metis *f tr-init trace-and-start-determines-end take-Nil*)  
next  
case *False*  
have *suc-n-minus-n*:  $Suc\ n - n = 1$  by *auto*  
then have  $length\ (take\ ((Suc\ n) - n)\ (drop\ n\ t)) = 1$  using *False* by *auto*  
then obtain *ev* where  $ev \# Nil = take\ ((Suc\ n) - n)\ (drop\ n\ t)$   
by (metis *False One-nat-def suc-n-minus-n length-greater-0-conv self-append-conv2 take-eq-Nil take-hd-drop*)  
then have *g*:  $(s\ init\ t\ n) \vdash ev \mapsto (s\ init\ t\ (Suc\ n))$   
by (metis *f tr-init trace-and-start-determines-end trace-decomp-head*)

```

then show ?thesis
proof (cases ev)
  case (Snapshot r)
  then show ?thesis
    using Suc.hyps Suc.prem1 g by auto
next
  case (RecvMarker cid' sr r)
  have cid' ≠ cid using RecvMarker can-occur-def g Suc by auto
  with RecvMarker Suc g show ?thesis by (cases has-snapshotted (s init t n)
sr, auto)
next
  case (Trans r u u')
  then show ?thesis
    by (metis Suc.hyps Suc.prem1 g next-trans)
next
  case (Send cid' r s u u' m)
  have cid' ≠ cid using Send can-occur-def g Suc by auto
  then show ?thesis using Suc g Send by simp
next
  case (Recv cid' s r u u' m)
  have cid' ≠ cid using Recv can-occur-def g Suc by auto
  then show ?thesis using Suc g Recv by simp
qed
qed
qed
end
end

```

### 3 Utilities

theory Util

imports

Main

HOL-Library.Sublist

HOL-Library.Multiset

begin

**abbreviation** swap-events where

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

**lemma** swap-neighbors-2:

shows

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

**proof** (induct i arbitrary: t)

case 0

**then show**  $?case$   
**by** (*metis One-nat-def Suc-eq-plus1 add-lessD1 append.left-neutral append-Cons cancel-comm-monoid-add-class.diff-cancel drop-update-cancel length-list-update numeral-One take-0 take-Cons-numeral upd-conv-take-nth-drop zero-less-Suc*)  
**next**  
**case** (*Suc n*)  
**let**  $?t = tl\ t$   
**have**  $t = hd\ t \# ?t$   
**by** (*metis Suc.prem hd-Cons-tl list.size(3) not-less-zero*)  
**moreover have**  $swap\ events\ n\ (n+1)\ ?t = (?t[n := ?t!\ (n+1)])[n+1 := ?t!\ n]$   
**by** (*metis Suc.hyps Suc.prem Suc-eq-plus1 length-tl less-diff-conv*)  
**ultimately show**  $?case$   
**by** (*metis Suc-eq-plus1 append-Cons diff-self-eq-0 drop-Suc-Cons list-update-code(3) nth-Cons-Suc take-Suc-Cons*)  
**qed**

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

**assumes**  
 $i < j$  **and**  
 $j < length\ t$   
**shows**  
 $length\ t = length\ (swap\ events\ i\ j\ t)$   
**proof** –  
**have**  $length\ (take\ i\ t\ @\ [t!\ j,\ t!\ i]\ @\ take\ (j - (i+1))\ (drop\ (i+1)\ t))$   
 $= length\ (take\ i\ t) + length\ [t!\ j,\ t!\ i] + length\ (take\ (j - (i+1))\ (drop\ (i+1)\ t))$   
**by** *simp*  
**then have**  $\dots = j+1$   
**using** *assms(1) assms(2)* **by** *auto*  
**then show**  $?thesis$  **using** *assms(2)* **by** *auto*  
**qed**

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

**assumes**  
 $i < j$  **and**  
 $j < length\ t$   
**shows**  
 $take\ i\ t = take\ i\ (swap\ events\ i\ j\ t)$   
**using** *assms* **by** *auto*

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

**assumes**  
 $i < j$  **and**  
 $j < length\ t$   
**shows**  
 $drop\ (j+1)\ t = drop\ (j+1)\ (swap\ events\ i\ j\ t)$   
**proof** –  
**have**  $length\ (take\ i\ t\ @\ [t!\ j,\ t!\ i]\ @\ take\ (j - (i+1))\ (drop\ (i+1)\ t))$

```

      = length (take i t) + length [t ! j, t ! i] + length (take (j - (i+1)) (drop
(i+1) t))
    by simp
    then have ... = j+1
      using assms(1) assms(2) by auto
    then show ?thesis
      by (metis ‹length (take i t @ [t ! j, t ! i] @ take (j - (i + 1)) (drop (i + 1)
t)) = length (take i t) + length [t ! j, t ! i] + length (take (j - (i + 1)) (drop (i
+ 1) t))› append.assoc append-eq-conv-conj)
    qed

```

**lemma** *swap-neighbors*:

```

  shows
    i+1 < length l ==> (l[i := l ! (i+1)])[i+1 := l ! i] = take i l @ [l ! (i+1), l !
i] @ drop (i+2) l
  proof (induct i arbitrary: l)
    case 0
    then show ?case
      by (metis One-nat-def add.left-neutral add-lessD1 append-Cons append-Nil
drop-update-cancel length-list-update one-add-one plus-1-eq-Suc take0 take-Suc-Cons
upd-conv-take-nth-drop zero-less-two)
    next
      case (Suc n)
      let ?l = tl l
      have (l[Suc n := l ! (Suc n + 1)])[Suc n + 1 := l ! Suc n] = hd l # (?l[n := ?l
! (n+1)])[n+1 := ?l ! n]
      by (metis Suc.prem1 add.commute add-less-same-cancel2 list.collapse list.size(3)
list-update-code(3) not-add-less2 nth-Cons-Suc plus-1-eq-Suc)
      have n + 1 < length ?l using Suc.prem1 by auto
      then have (?l[n := ?l ! (n+1)])[n+1 := ?l ! n] = take n ?l @ [?l ! (n+1), ?l !
n] @ drop (n+2) ?l
      using Suc.hyps by simp
      then show ?case
        by (cases l) auto
    qed

```

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

```

  assumes
    i < j and
    j < length t
  shows
    mset (swap-events i j t) = mset t
  proof -
    have swap: swap-events i j t
      = take i t @ [t ! j, t ! i] @ (take (j - (i+1)) (drop (i+1) t)) @ (drop (j+1)
t)
    by auto
    have reg: t = take i t @ (take ((j+1) - i) (drop i t)) @ drop (j+1) t
      by (metis add-diff-inverse-nat add-lessD1 append.assoc append-take-drop-id

```

```

assms(1) less-imp-add-positive less-not-refl take-add
  have mset (take i t) = mset (take i t) by simp
  moreover have mset (drop (j+1) t) = mset (drop (j+1) t) by simp
  moreover have mset ([t ! j, t ! i] @ (take (j - (i+1)) (drop (i+1) t))) = mset
(take ((j+1) - i) (drop i t))
  proof -
    let ?l = take (j - (i+1)) (drop (i+1) t)
    have take ((j+1) - i) (drop i t) = t ! i # ?l @ [t ! j]
    proof -
      have f1: Suc (j - Suc i) = j - i
        by (meson Suc-diff-Suc assms(1))
      have f2: i < length t
        using assms(1) assms(2) by linarith
      have f3: j - (i + 1) + (i + 1) = j
        using <i < j> by simp
      then have drop (j - (i + 1)) (drop (i + 1) t) = drop j t
        by (metis drop-drop)
      then show ?thesis
        using f3 f2 f1 by (metis (no-types) Cons-nth-drop-Suc Suc-diff-le Suc-eq-plus1
assms(1) assms(2) hd-drop-conv-nth length-drop less-diff-conv nat-less-le take-Suc-Cons
take-hd-drop)
      qed
    then show ?thesis by fastforce
  qed
  ultimately show ?thesis using swap reg
  by simp (metis mset-append union-mset-add-mset-left union-mset-add-mset-right)
qed

```

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

**fixes**

*i :: nat*

**shows**

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

**by** *auto*

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

**assumes**

*filter (( $\neq$ ) a) l  $\neq$  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** (*rule ccontr*)

**assume**  $\sim ?P$

**then have** *asm:  $\forall i j. i < j \wedge j < \text{length } l \longrightarrow l ! i \neq a \vee l ! j = a$*  (**is** *?Q*) **by** *simp*

**then have** *filter (( $\neq$ ) a) l = takeWhile (( $\neq$ ) a) l*

**proof** (*cases a : set l*)

**case** *False*

**then have**  $\forall i. i < \text{length } l \longrightarrow l ! i \neq a$  **by** *auto*

**then show** *?thesis*

```

    by (metis (mono-tags, lifting) False filter-True takeWhile-eq-all-conv)
next
case True
then have ex-j:  $\exists j. j < \text{length } l \wedge l ! j = a$ 
  by (simp add: in-set-conv-nth)
let ?j = Min {j. j < length l  $\wedge$  l ! j = a}
have fin-j: finite {j. j < length l  $\wedge$  l ! j = a}
  using finite-nat-set-iff-bounded by blast
moreover have {j. j < length l  $\wedge$  l ! j = a}  $\neq$  empty using ex-j by blast
ultimately have ?j < length l
  using Min-less-iff by blast
have tail-all-a:  $\forall j. j < \text{length } l \wedge j \geq ?j \longrightarrow l ! j = a$ 
proof (rule allI, rule impI)
  fix j
  assume j < length l  $\wedge$  j  $\geq$  ?j
  moreover have  $\llbracket ?Q; j < \text{length } l \wedge j \geq ?j \rrbracket \implies \forall k. k \geq ?j \wedge k \leq j \longrightarrow l$ 
! j = a
  proof (induct j - ?j)
    case 0
    then have j = ?j using 0 by simp
    then show ?case
      using Min-in  $\langle \{j. j < \text{length } l \wedge l ! j = a\} \neq \{\} \rangle$  fin-j by blast
  next
  case (Suc n)
  then have  $\forall k. k \geq ?j \wedge k < j \longrightarrow l ! j = a$ 
    by (metis (mono-tags, lifting) Min-in  $\langle \{j. j < \text{length } l \wedge l ! j = a\} \neq \{\} \rangle$ 
fin-j le-eq-less-or-eq mem-Collect-eq)
  then show ?case
    using Suc.hyps(2) by auto
qed
ultimately show l ! j = a using asm by blast
qed
moreover have head-all-not-a:  $\forall i. i < ?j \longrightarrow l ! i \neq a$  using asm calculation

  by (metis (mono-tags, lifting) Min-le  $\langle \text{Min } \{j. j < \text{length } l \wedge l ! j = a\} <$ 
length l  $\rangle$  fin-j leD less-trans mem-Collect-eq)
ultimately have takeWhile (( $\neq$ ) a) l = take ?j l
proof -
  have length (takeWhile (( $\neq$ ) a) l) = ?j
  proof -
    have length (takeWhile (( $\neq$ ) a) l)  $\leq$  ?j (is ?S)
    proof (rule ccontr)
      assume  $\neg ?S$ 
      then have l ! ?j  $\neq$  a
        by (metis (mono-tags, lifting) not-le-imp-less nth-mem set-takeWhileD
takeWhile-nth)
      then show False
        using  $\langle \text{Min } \{j. j < \text{length } l \wedge l ! j = a\} < \text{length } l \rangle$  tail-all-a by blast
    qed
  qed

```

**moreover have**  $\text{length } (\text{takeWhile } ((\neq) a) l) \geq ?j$  (**is**  $?T$ )  
**proof** (*rule ccontr*)  
**assume**  $\neg ?T$   
**then have**  $\exists j. j < ?j \wedge l ! j = a$   
**by** (*metis (mono-tags, lifting) ‹Min {j. j < length l ‹ l ! j = a} < length l›* *calculation le-less-trans not-le-imp-less nth-length-takeWhile*)  
**then show** *False*  
**using** *head-all-not-a* **by** *blast*  
**qed**  
**ultimately show** *?thesis* **by** *simp*  
**qed**  
**moreover have**  $\text{length } (\text{take } ?j l) = ?j$   
**by** (*metis calculation takeWhile-eq-take*)  
**ultimately show** *?thesis*  
**by** (*metis takeWhile-eq-take*)

**qed**  
**moreover have**  $\text{filter } ((\neq) a) l = \text{take } ?j l$   
**proof** –  
**have**  $\text{filter } ((\neq) a) l = \text{filter } ((\neq) a) (\text{take } ?j l) @ \text{filter } ((\neq) a) (\text{drop } ?j l)$   
**by** (*metis append-take-drop-id filter-append*)  
**moreover have**  $\text{filter } ((\neq) a) (\text{take } ?j l) = \text{take } ?j l$  **using** *head-all-not-a*  
**by** (*metis ‹takeWhile ((≠) a) l = take (Min {j. j < length l ‹ l ! j = a}›* *l› filter-id-conv set-takeWhileD*)  
**moreover have**  $\text{filter } ((\neq) a) (\text{drop } ?j l) = []$   
**proof** –  
**have**  $\forall j. j \geq ?j \wedge j < \text{length } l \longrightarrow l ! j = \text{drop } ?j l ! (j - ?j)$   
**by** *simp*  
**then have**  $\forall j. j < \text{length } l - ?j \longrightarrow \text{drop } ?j l ! j = a$  **using** *tail-all-a*  
**by** (*metis (no-types, lifting) Groups.add-ac(2) ‹Min {j. j < length l ‹ l ! j = a} < length l›* *less-diff-conv less-imp-le-nat not-add-less2 not-le nth-drop*)  
**then show** *?thesis*  
**proof** –  
**obtain**  $aa :: ('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a$  **where**  
 $\forall x0 x1. (\exists v2. v2 \in \text{set } x1 \wedge x0 v2) = (aa x0 x1 \in \text{set } x1 \wedge x0 (aa x0 x1))$   
**by** *moura*  
**then have**  $f1: \forall as p. aa p as \in \text{set } as \wedge p (aa p as) \vee \text{filter } p as = []$   
**by** (*metis (full-types) filter-False*)  
**obtain**  $nn :: 'a \text{ list} \Rightarrow 'a \Rightarrow \text{nat}$  **where**  
 $f2: \forall x0 x1. (\exists v2 < \text{length } x0. x0 ! v2 = x1) = (nn x0 x1 < \text{length } x0 \wedge x0 ! nn x0 x1 = x1)$   
**by** *moura*  
**{ assume**  $\text{drop } (\text{Min } \{n. n < \text{length } l \wedge l ! n = a\}) l ! nn (\text{drop } (\text{Min } \{n. n < \text{length } l \wedge l ! n = a\}) l) (aa ((\neq) a) (\text{drop } (\text{Min } \{n. n < \text{length } l \wedge l ! n = a\}) l)) = a$   
**then have**  $\text{filter } ((\neq) a) (\text{drop } (\text{Min } \{n. n < \text{length } l \wedge l ! n = a\}) l) = [] \vee \neg nn (\text{drop } (\text{Min } \{n. n < \text{length } l \wedge l ! n = a\}) l) (aa ((\neq) a) (\text{drop } (\text{Min } \{n. n < \text{length } l \wedge l ! n = a\}) l)) < \text{length } (\text{drop } (\text{Min } \{n. n < \text{length } l \wedge l ! n = a\}) l)$

$l) \vee \text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l! nn (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l) (aa ((\neq) a) (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l)) \neq aa ((\neq) a) (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l)$   
**using f1 by (metis (full-types)) }**  
**moreover**  
**{ assume  $\neg nn (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l) (aa ((\neq) a) (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l)) < \text{length } l - \text{Min} \{n. n < \text{length } l \wedge l! n = a\}$**   
**then have  $\neg nn (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l) (aa ((\neq) a) (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l)) < \text{length} (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l) \vee \text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l! nn (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l) (aa ((\neq) a) (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l)) \neq aa ((\neq) a) (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l)$**   
**by simp }**  
**ultimately have filter  $((\neq) a) (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l) = [] \vee \neg nn (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l) (aa ((\neq) a) (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l)) < \text{length} (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l) \vee \text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l! nn (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l) (aa ((\neq) a) (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l)) \neq aa ((\neq) a) (\text{drop} (\text{Min} \{n. n < \text{length } l \wedge l! n = a\}) l)$**   
**using  $\langle \forall j < \text{length } l - \text{Min} \{j. j < \text{length } l \wedge l! j = a\}. \text{drop} (\text{Min} \{j. j < \text{length } l \wedge l! j = a\}) l! j = a \rangle$  by blast**  
**then show ?thesis**  
**using f2 f1 by (meson in-set-conv-nth)**  
**qed**  
**qed**  
**ultimately show ?thesis by simp**  
**qed**  
**ultimately show ?thesis by simp**  
**qed**  
**then show False using assms by simp**  
**qed**

**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 -**

**have  $\forall j. j < \text{length } l' - 1 \longrightarrow l'! j \neq m$**

**by (metis assms(1) assms(2) butlast-snoc length-butlast nth-append nth-mem)**

**then have one-j:  $\forall j. j < \text{length } l' \wedge l'! j = m \longrightarrow j = (\text{length } l' - 1)$**

**by (metis (no-types, opaque-lifting) diff-Suc-1 lessE)**

**show ?thesis**

**proof (rule ccontr)**

**assume  $\sim ?P$**

**then obtain  $i j$  where  $i \neq j$   $i < \text{length } l'$   $j < \text{length } l'$**

$l'! i = m$   $l'! j = m$

```

    using assms by auto
    then show False using one-j by blast
  qed
qed

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 (rule iffI)
    assume  $\exists! j. j < \text{length } l \wedge l ! j = a$ 
    then obtain j where  $j < \text{length } l \wedge l ! j = a$ 
      by blast
    moreover have  $\forall k. k \neq j \wedge k < \text{length } l \longrightarrow l ! k \neq a$ 
      using  $\langle \exists! j. j < \text{length } l \wedge l ! j = a \rangle \langle j < \text{length } l \rangle \langle l ! j = a \rangle$  by blast
    moreover have  $l = \text{take } j \ l @ [l ! j] @ \text{drop } (j+1) \ l$ 
      by (metis Cons-eq-appendI Cons-nth-drop-Suc Suc-eq-plus1 append-self-conv2
append-take-drop-id calculation(1) calculation(2))
    moreover have  $\text{filter } ((=) a) (\text{take } j \ l) = []$ 
      proof -
        have  $\forall k. k < \text{length } (\text{take } j \ l) \longrightarrow (\text{take } j \ l) ! k \neq a$ 
          using calculation(3) by auto
        then show ?thesis
          by (metis (full-types) filter-False in-set-conv-nth)
      qed
    moreover have  $\text{filter } ((=) a) (\text{drop } (j+1) \ l) = []$ 
      proof -
        have  $\forall k. k < \text{length } (\text{drop } (j+1) \ l) \longrightarrow (\text{drop } (j+1) \ l) ! k \neq a$ 
          using calculation(3) by auto
        then show ?thesis
          by (metis (full-types) filter-False in-set-conv-nth)
      qed
    ultimately show  $\text{length } (\text{filter } ((=) a) l) = 1$ 
      by (metis (mono-tags, lifting) One-nat-def Suc-eq-plus1 append-Cons append-self-conv2
filter.simps(2) filter-append list.size(3) list.size(4))
  next
    assume asm:  $\text{length } (\text{filter } ((=) a) l) = 1$ 
    then have  $\text{filter } ((=) a) l = [a]$ 
      proof -
        let ?xs =  $\text{filter } ((=) a) l$ 
        have  $\text{length } ?xs = 1$ 
          using asm by blast
        then show ?thesis
          by (metis (full-types) Cons-eq-filterD One-nat-def length-0-conv length-Suc-conv)
      qed
    then have  $\exists j. j < \text{length } l \wedge l ! j = a$ 
      by (metis (full-types) filter-False in-set-conv-nth list.discI)
    then obtain j where  $j < \text{length } l \wedge l ! j = a$  by blast
    moreover have  $\forall k. k < \text{length } l \wedge k \neq j \longrightarrow l ! k \neq a$ 
      proof (rule allI, rule impI)

```

```

fix k
assume assm:  $k < \text{length } l \wedge k \neq j$ 
then have  $\langle k < \text{length } l \rangle ..$ 
show  $l ! k \neq a$ 
proof (rule ccontr)
  assume lka:  $\neg l ! k \neq a$ 
  then have  $\langle l ! k = a \rangle$ 
    by simp
  show False
  proof (cases  $k < j$ )
    let ?xs = take (k+1) l
    let ?ys = drop (k+1) l
    case True
    then have  $\text{length } (\text{filter } ((=) a) ?xs) > 0$ 
      using  $\langle k < \text{length } l \rangle \langle l ! k = a \rangle$  by (auto simp add: filter-empty-conv
in-set-conv-nth)
    moreover have  $\text{length } (\text{filter } ((=) a) ?ys) > 0$ 
    proof -
      have  $?ys ! (j - (k+1)) = l ! j$ 
        using True assm by auto
      moreover have  $j - (k+1) < \text{length } ?ys$ 
        using True  $\langle j < \text{length } l \rangle$  by auto
      ultimately show ?thesis
        by (metis (full-types)  $\langle l ! j = a \rangle$  filter-empty-conv length-greater-0-conv
nth-mem)
    qed
    moreover have  $?xs @ ?ys = l$ 
      using append-take-drop-id by blast
    ultimately have  $\text{length } (\text{filter } ((=) a) l) > 1$ 
      by (metis (no-types, lifting) One-nat-def Suc-eq-plus1 asm filter-append
length-append less-add-eq-less less-one nat-neq-iff)
    then show False using asm by simp
  next
  let ?xs = take (j+1) l
  let ?ys = drop (j+1) l
  case False
  then have  $\text{length } (\text{filter } ((=) a) ?xs) > 0$ 
    using  $\langle k < \text{length } l \rangle \langle l ! j = a \rangle$  by (auto simp add: filter-empty-conv
in-set-conv-nth)
  moreover have  $\text{length } (\text{filter } ((=) a) ?ys) > 0$ 
  proof -
    have  $?ys ! (k - (j+1)) = l ! k$ 
      using False assm by auto
    moreover have  $k - (j+1) < \text{length } ?ys$ 
      using False assm by auto
    ultimately show ?thesis
      by (metis (full-types) filter-empty-conv length-greater-0-conv lka nth-mem)
  qed
  moreover have  $?xs @ ?ys = l$ 

```

```

    using append-take-drop-id by blast
    ultimately have length (filter ((=) a) l) > 1
      by (metis (no-types, lifting) One-nat-def Suc-eq-plus1 asm filter-append
length-append less-add-eq-less less-one nat-neq-iff)
    then show False using asm by simp
  qed
qed
qed
ultimately show  $\exists!j. j < \text{length } l \wedge l ! j = a$  by blast
qed
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  $\neq$  occurs-on ev'
  shows
     $\text{msgs } e \ i = \text{msgs } e' \ i$ 
proof -
  let  $?p = \text{occurs-on } ev$ 
  let  $?q = \text{occurs-on } ev'$ 
  obtain  $u \ u'$  where  $ev = \text{Trans } ?p \ u \ u'$ 
    by (metis assms(3) event.collapse(1))
  obtain  $u'' \ u'''$  where  $ev' = \text{Trans } ?q \ u'' \ u'''$ 
    by (metis assms(4) event.collapse(1))
  then have  $\text{msgs } d' \ i = \text{msgs } d \ i$ 
    by (metis Trans-msg assms(1) assms(3) assms(4) assms(5))
  then have  $\text{msgs } e \ i = \text{msgs } e' \ i$ 
    using Trans-msg assms(2) assms(3) assms(4) assms(6) by auto
  then show ?thesis by blast

```

qed

**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** –

**let**  $?p =$  *occurs-on*  $ev$

**let**  $?q =$  *occurs-on*  $ev'$

**obtain**  $i' r u u' m$  **where** *Send-ev*:  $ev = \text{Send } i' ?p r u u' m$

**by** (*metis* *assms*(3) *event.collapse*(2))

**obtain**  $i'' s u'' u''' m'$  **where** *Send-ev'*:  $ev' = \text{Send } i'' ?q s u'' u''' m'$

**by** (*metis* *assms*(4) *event.collapse*(2))

**have**  $i' \neq i''$

**by** (*metis* (*mono-tags*, *lifting*)  $\langle ev = \text{Send } i' (\text{occurs-on } ev) r u u' m \rangle \langle ev' = \text{Send } i'' (\text{occurs-on } ev') s u'' u''' m' \rangle$  *assms*(1) *assms*(2) *assms*(7) *can-occur-def* *event.simps*(27) *happen-implies-can-occur* *option.simps*(1) *prod.simps*(1))

**then show**  $?thesis$

**proof** (*cases*  $i = i' \vee i = i''$ )

**case** *True*

**then show**  $?thesis$

**proof** (*elim* *disjE*)

**assume**  $i = i'$

**then have**  $msgs\ d\ i = msgs\ c\ i @ [Msg\ m]$

**by** (*metis*  $\langle ev = \text{Send } i' (\text{occurs-on } ev) r u u' m \rangle$  *assms*(1) *next-send*)

**moreover have**  $msgs\ e\ i = msgs\ d\ i$

**by** (*metis*  $\langle ev' = \text{Send } i'' (\text{occurs-on } ev') s u'' u''' m' \rangle \langle i = i' \rangle \langle i' \neq i'' \rangle$  *assms*(2) *assms*(4) *event.sel*(8) *msgs-unchanged-for-other-is* *regular-event*)

**moreover have**  $msgs\ d'\ i = msgs\ c\ i$

**by** (*metis*  $\langle ev' = \text{Send } i'' (\text{occurs-on } ev') s u'' u''' m' \rangle \langle i = i' \rangle \langle i' \neq i'' \rangle$  *assms*(4) *assms*(5) *event.sel*(8) *msgs-unchanged-for-other-is* *regular-event*)

**moreover have**  $msgs\ e'\ i = msgs\ d'\ i @ [Msg\ m]$

**by** (*metis*  $\langle ev = \text{Send } i' (\text{occurs-on } ev) r u u' m \rangle \langle i = i' \rangle$  *assms*(6) *next-send*)

**ultimately show**  $?thesis$  **by** *simp*

**next**

**assume**  $i = i''$

**then have**  $msgs\ d\ i = msgs\ c\ i$

**by** (*metis* *Send-ev*  $\langle i' \neq i'' \rangle$  *assms*(1) *assms*(3) *event.sel*(8) *msgs-unchanged-for-other-is* *regular-event*)

**moreover have**  $msgs\ e\ i = msgs\ d\ i @ [Msg\ m]$

**by** (*metis* *Send-ev'*  $\langle i = i'' \rangle$  *assms*(2) *next-send*)

**moreover have**  $msgs\ d'\ i = msgs\ c\ i @ [Msg\ m]$

by (metis Send-ev'  $\langle i = i'' \rangle$  assms(5) next-send)  
 moreover have  $msgs\ e'\ i = msgs\ d'\ i$   
 by (metis Send-ev  $\langle i = i'' \rangle \langle i' \neq i'' \rangle$  assms(3) assms(6) event.sel(8)  
 msgs-unchanged-for-other-is regular-event)  
 ultimately show ?thesis by simp  
 qed  
 next  
 case False  
 then have  $msgs\ e\ i = msgs\ d\ i$  using Send-ev' assms  
 by (metis event.sel(8) msgs-unchanged-for-other-is regular-event)  
 also have  $\dots = msgs\ c\ i$   
 by (metis False Send-ev assms(1) assms(3) event.sel(8) msgs-unchanged-for-other-is  
 regular-event)  
 also have  $\dots = msgs\ d'\ i$   
 by (metis (no-types, opaque-lifting)  $\langle msgs\ d\ i = msgs\ c\ i \rangle$  assms(2) assms(4)  
 assms(5) calculation regular-event same-messages-imply-same-resulting-messages)  
 also have  $\dots = msgs\ e'\ i$   
 by (metis (no-types, opaque-lifting)  $\langle msgs\ c\ i = msgs\ d'\ i \rangle \langle msgs\ d\ i = msgs\ c\ i \rangle$   
 assms(1) assms(3) assms(6) regular-event same-messages-imply-same-resulting-messages)  
 finally show ?thesis by simp  
 qed  
 qed

**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** –  
 let ?p = *occurs-on*  $ev$   
 let ?q = *occurs-on*  $ev'$   
**obtain**  $i'\ r\ u\ u'\ m$  **where** *Recv-ev*:  $ev = Recv\ i'\ ?p\ r\ u\ u'\ m$   
 by (metis assms(3) event.collapse(3))  
**obtain**  $i''\ s\ u''\ u''' m'$  **where** *Recv-ev'*:  $ev' = Recv\ i''\ ?q\ s\ u''\ u''' m'$   
 by (metis assms(4) event.collapse(3))  
**have**  $i' \neq i''$   
 by (metis *Recv-ev Recv-ev'* assms(1) assms(2) assms(7) *can-occur-Recv hap-*  
*pen-implies-can-occur option.simps(1) prod.simps(1)*)  
**show** ?thesis  
**proof** (cases  $i = i' \vee i = i''$ )  
 case True  
**then show** ?thesis  
**proof** (elim disjE)

**assume**  $i = i'$   
**then have**  $\text{Msg } m \# \text{ msgs } d \ i = \text{ msgs } c \ i$  **using** *Recv-ev* *assms* **by** (*metis* *next-recv*)  
**moreover have**  $\text{msgs } e \ i = \text{ msgs } d \ i$   
**by** (*metis* *Recv-ev'*  $\langle i = i' \rangle \langle i' \neq i'' \rangle$  *assms*(2) *assms*(4) *event.sel*(9) *msgs-unchanged-for-other-is* *regular-event*)  
**moreover have**  $\text{msgs } d' \ i = \text{ msgs } c \ i$   
**by** (*metis* *Recv-ev'*  $\langle i = i' \rangle \langle i' \neq i'' \rangle$  *assms*(4) *assms*(5) *event.sel*(9) *msgs-unchanged-for-other-is* *regular-event*)  
**moreover have**  $\text{Msg } m \# \text{ msgs } e' \ i = \text{ msgs } d' \ i$   
**by** (*metis* *Recv-ev*  $\langle i = i' \rangle$  *assms*(6) *next-recv*)  
**ultimately show** *?thesis* **by** (*metis* *list.inject*)  
**next**  
**assume**  $i = i''$   
**then have**  $\text{msgs } d \ i = \text{ msgs } c \ i$   
**by** (*metis* *Recv-ev*  $\langle i' \neq i'' \rangle$  *assms*(1) *assms*(3) *event.sel*(9) *msgs-unchanged-for-other-is* *regular-event*)  
**moreover have**  $\text{Msg } m' \# \text{ msgs } e \ i = \text{ msgs } d \ i$   
**by** (*metis* *Recv-ev'*  $\langle i = i'' \rangle$  *assms*(2) *next-recv*)  
**moreover have**  $\text{Msg } m' \# \text{ msgs } d' \ i = \text{ msgs } c \ i$   
**by** (*metis* *Recv-ev'*  $\langle i = i'' \rangle$  *assms*(5) *next-recv*)  
**moreover have**  $\text{msgs } e' \ i = \text{ msgs } d' \ i$   
**by** (*metis* *Recv-ev*  $\langle i = i'' \rangle \langle i' \neq i'' \rangle$  *assms*(3) *assms*(6) *event.sel*(9) *msgs-unchanged-for-other-is* *regular-event*)  
**ultimately show** *?thesis* **by** (*metis* *list.inject*)  
**qed**  
**next**  
**case** *False*  
**then have**  $\text{msgs } e \ i = \text{ msgs } d \ i$   
**by** (*metis* *Recv-ev'* *assms*(2) *assms*(4) *event.sel*(9) *msgs-unchanged-for-other-is* *regular-event*)  
**also have**  $\dots = \text{ msgs } c \ i$   
**by** (*metis* *False* *Recv-ev* *assms*(1) *assms*(3) *event.sel*(9) *msgs-unchanged-for-other-is* *regular-event*)  
**also have**  $\dots = \text{ msgs } d' \ i$   
**by** (*metis* (*no-types*, *opaque-lifting*)  $\langle \text{msgs } d \ i = \text{ msgs } c \ i \rangle$  *assms*(2) *assms*(4) *assms*(5) *calculation* *regular-event* *same-messages-imply-same-resulting-messages*)  
**also have**  $\dots = \text{ msgs } e' \ i$   
**by** (*metis* (*no-types*, *lifting*)  $\langle \text{msgs } c \ i = \text{ msgs } d' \ i \rangle \langle \text{msgs } d \ i = \text{ msgs } c \ i \rangle$  *assms*(1) *assms*(3) *assms*(6) *regular-event* *same-messages-imply-same-resulting-messages*)  
**finally show** *?thesis* **by** *simp*  
**qed**  
**qed**

**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 ⊢ ev' ↦ d' and*  
*d' ⊢ ev ↦ e' and*  
*occurs-on ev ≠ occurs-on ev'*  
**shows**  
*msgs e i = msgs e' i*  
**proof** –  
**let** *?p = occurs-on ev*  
**let** *?q = occurs-on ev'*  
**obtain** *i' r u u' m* **where** *Send: ev = Send i' ?p r u u' m*  
**by** (*metis assms(3) event.collapse(2)*)  
**obtain** *u'' u'''* **where** *Trans: ev' = Trans ?q u'' u'''*  
**by** (*metis assms(4) event.collapse(1)*)  
**show** *?thesis*  
**proof** (*cases i = i'*)  
**case** *True*  
**then have** *msgs d i = msgs c i @ [Msg m]*  
**by** (*metis Send assms(1) next-send*)  
**moreover have** *msgs e i = msgs d i*  
**using** *Trans-msg assms(2) assms(4)* **by** *auto*  
**moreover have** *msgs d' i = msgs c i*  
**using** *Trans-msg assms(4) assms(5)* **by** *auto*  
**moreover have** *msgs e' i = msgs d' i @ [Msg m]*  
**by** (*metis Send True assms(6) next-send*)  
**ultimately show** *?thesis* **by** *simp*  
**next**  
**case** *False*  
**then have** *msgs e i = msgs d i*  
**using** *Trans-msg assms(2) assms(4)* **by** *auto*  
**also have** *... = msgs c i*  
**by** (*metis False Send assms(1) assms(3) event.sel(8) msgs-unchanged-for-other-is*  
*regular-event*)  
**also have** *... = msgs d' i*  
**using** *Trans-msg assms(4) assms(5)* **by** *blast*  
**also have** *... = msgs e' i*  
**by** (*metis (no-types, lifting) ⟨msgs c i = msgs d' i⟩ ⟨msgs d i = msgs c i⟩*  
*assms(1) assms(3) assms(6) regular-event same-messages-imply-same-resulting-messages*)  
**finally show** *?thesis* **by** *simp*  
**qed**  
**qed**

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

**assumes**  
*c ⊢ ev ↦ d and*  
*d ⊢ ev' ↦ e and*  
*isTrans ev and*  
*isSend ev' and*  
*c ⊢ ev' ↦ d' and*  
*d' ⊢ ev ↦ e' and*

$occurs-on\ ev \neq occurs-on\ ev'$   
**shows**  
 $msgs\ e\ i = msgs\ e'\ i$   
**using** *assms swap-msgs-Send-Trans* **by** *auto*

**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** –  
**let**  $?p = occurs-on\ ev$   
**let**  $?q = occurs-on\ ev'$   
**obtain**  $i' r u u' m$  **where** *Recv*:  $ev = Recv\ i'\ ?p\ r\ u\ u'\ m$   
**by** (*metis assms(3) event.collapse(3)*)  
**obtain**  $u'' u'''$  **where** *Trans*:  $ev' = Trans\ ?q\ u''\ u'''$   
**by** (*metis assms(4) event.collapse(1)*)  
**show** *?thesis*  
**proof** (*cases i = i'*)  
**case** *True*  
**then have**  $Msg\ m \# msgs\ d\ i = msgs\ c\ i$   
**by** (*metis Recv assms(1) next-recv*)  
**moreover have**  $msgs\ e\ i = msgs\ d\ i$   
**using** *Trans-msg assms(2) assms(4)* **by** *auto*  
**moreover have**  $msgs\ d'\ i = msgs\ c\ i$   
**using** *Trans-msg assms(4) assms(5)* **by** *auto*  
**moreover have**  $Msg\ m \# msgs\ e'\ i = msgs\ d'\ i$   
**by** (*metis Recv True assms(6) next-recv*)  
**ultimately show** *?thesis* **by** (*metis list.inject*)

**next**  
**case** *False*  
**then have**  $msgs\ e\ i = msgs\ d\ i$   
**using** *Trans-msg assms(2) assms(4)* **by** *auto*  
**also have**  $\dots = msgs\ c\ i$   
**by** (*metis False Recv assms(1) assms(3) event.sel(9) msgs-unchanged-for-other-is-regular-event*)  
**also have**  $\dots = msgs\ d'\ i$   
**using** *Trans-msg assms(4) assms(5)* **by** *blast*  
**also have**  $\dots = msgs\ e'\ i$   
**by** (*metis False Recv assms(6) next-recv*)  
**finally show** *?thesis* **by** *simp*

**qed**  
**qed**

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

**shows**

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

**using** *assms swap-msgs-Recv-Trans* **by** *auto*

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

**shows**

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

**proof** –

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

**let**  $?q = \text{occurs-on } ev'$

**obtain**  $i' \ r \ u \ u' \ m$  **where** *Send*:  $ev = \text{Send } i' \ ?p \ r \ u \ u' \ m$

**by** (*metis assms(3) event.collapse(2)*)

**obtain**  $i'' \ s \ u'' \ u''' \ m'$  **where** *Recv*:  $ev' = \text{Recv } i'' \ ?q \ s \ u'' \ u''' \ m'$

**by** (*metis assms(4) event.collapse(3)*)

**show** *?thesis*

**proof** (*cases*  $i = i'$ ; *cases*  $i = i''$ , *goal-cases*)

**case** 1

**then have**  $\text{msgs } e' \ i = \text{msgs } d' \ i \ @ \ [\text{Msg } m]$

**by** (*metis Send assms(6) next-send*)

**moreover have**  $\text{Msg } m' \ \# \ \text{msgs } d' \ i = \text{msgs } c \ i$

**by** (*metis 1(2) Recv assms(5) next-recv*)

**moreover have**  $\text{msgs } d \ i = \text{msgs } c \ i \ @ \ [\text{Msg } m]$

**by** (*metis 1(1) Send assms(1) next-send*)

**moreover have**  $\text{Msg } m' \ \# \ \text{msgs } e \ i = \text{msgs } d \ i$

**by** (*metis 1(2) Recv assms(2) next-recv*)

**ultimately show** *?thesis*

**by** (*metis list.sel(2) list.sel(3) not-Cons-self2 tl-append2*)

**next**

**case** 2

**then have**  $\text{msgs } d \ i = \text{msgs } c \ i \ @ \ [\text{Msg } m]$

**by** (*metis Send assms(1) next-send*)

**moreover have**  $msgs\ e\ i = msgs\ d\ i$   
**by** (*metis* 2(2) *Recv* *assms*(2) *assms*(4) *event.sel*(9) *msgs-unchanged-for-other-is-regular-event*)  
**moreover have**  $msgs\ d'\ i = msgs\ c\ i$   
**by** (*metis* 2(2) *Recv* *assms*(4) *assms*(5) *event.sel*(9) *msgs-unchanged-for-other-is-regular-event*)  
**moreover have**  $msgs\ e'\ i = msgs\ d'\ i \ @ \ [Msg\ m]$   
**by** (*metis* *Send* 2(1) *assms*(6) *next-send*)  
**ultimately show** *?thesis* **by** *simp*  
**next**  
**assume**  $3: i \neq i' \ i = i''$   
**then have**  $msgs\ d\ i = msgs\ c\ i$   
**by** (*metis* *Send* *assms*(1) *assms*(3) *event.sel*(8) *msgs-unchanged-for-other-is-regular-event*)  
**moreover have**  $Msg\ m' \# msgs\ e\ i = msgs\ d\ i$  **using** 3 *Recv* *assms* **by** (*metis* *next-recv*)  
**moreover have**  $Msg\ m' \# msgs\ d'\ i = msgs\ c\ i$   
**by** (*metis* 3(2) *Recv* *assms*(5) *next-recv*)  
**moreover have**  $msgs\ e'\ i = msgs\ d'\ i$   
**by** (*metis* 3(1) *Send* *assms*(6) *next-send*)  
**ultimately show** *?thesis* **by** (*metis* *list.inject*)  
**next**  
**assume**  $4: i \neq i' \ i \neq i''$   
**then have**  $msgs\ e\ i = msgs\ d\ i$   
**by** (*metis* *Recv* *assms*(2) *assms*(4) *event.sel*(9) *msgs-unchanged-for-other-is-regular-event*)  
**also have**  $\dots = msgs\ c\ i$   
**by** (*metis* 4(1) *Send* *assms*(1) *assms*(3) *event.sel*(8) *msgs-unchanged-for-other-is-regular-event*)  
**also have**  $\dots = msgs\ d'\ i$   
**by** (*metis* 4(2) *Recv* *assms*(5) *next-recv*)  
**also have**  $\dots = msgs\ e'\ i$   
**by** (*metis* 4(1) *Send* *assms*(6) *next-send*)  
**finally show** *?thesis* **by** *simp*  
**qed**  
**qed**

**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**

$msgs\ e\ i = msgs\ e'\ i$

**using** *assms* *swap-msgs-Send-Recv* **by** *auto*

**lemma** *same-cs-implies-same-resulting-cs*:

**assumes**  
 $cs\ c\ i = cs\ d\ i$   
 $c \vdash ev \mapsto c'$  **and**  
 $d \vdash ev \mapsto d'$  **and**  
*regular-event*  $ev$

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

**proof** –  
**have**  $isTrans\ ev \vee isSend\ ev \vee isRecv\ ev$  **using** *assms* **by** *simp*  
**then show** *?thesis*  
**proof** (*elim disjE*)  
**assume**  $isTrans\ ev$   
**then show** *?thesis*  
**by** (*metis (no-types, lifting) assms(1) assms(2) assms(3) assms(4) event.distinct-disc(4)*)  
*no-cs-change-if-no-event*)  
**next**  
**assume**  $isSend\ ev$   
**then show** *?thesis*  
**by** (*metis (no-types, lifting) assms(1) assms(2) assms(3) assms(4) event.distinct-disc(10)*)  
*no-cs-change-if-no-event*)  
**next**  
**assume**  $isRecv\ ev$   
**then obtain**  $i' r s u u' m$  **where**  $Recv: ev = Recv\ i' r s u u' m$   
**by** (*meson isRecv-def*)  
**then show** *?thesis*  
**proof** (*cases i' = i*)  
**case** *True*  
**with** *assms Recv* **show** *?thesis* **by** (*cases snd (cs c i) = Recording, auto*)  
**next**  
**case** *False*  
**then show** *?thesis* **using** *assms Recv* **by** *simp*  
**qed**  
**qed**  
**qed**

**lemma** *regular-event-implies-same-channel-snapshot-Recv-Recv*:

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

**shows**  
 $cs\ e\ i = cs\ e'\ i$

**proof** –

```

let ?p = occurs-on ev
let ?q = occurs-on ev'
obtain i' r u u' m where Recv-ev: ev = Recv i' ?p r u u' m
  by (metis assms(3) event.collapse(3))
obtain i'' s u'' u''' m' where Recv-ev': ev' = Recv i'' ?q s u'' u''' m'
  by (metis assms(4) event.collapse(3))
have i' ≠ i''
  by (metis Recv-ev Recv-ev' assms(1) assms(5) assms(7) can-occur-Recv hap-
pen-implies-can-occur option.simps(1) prod.simps(1))
show ?thesis
proof (cases i = i' ∨ i = i'')
  case True
  then show ?thesis
  proof (elim disjE)
    assume i = i'
    then have cs d' i = cs c i
      using assms(4) assms(5) assms(7) no-cs-change-if-no-event
      by (metis Recv-ev' ⟨i' ≠ i''⟩ event.sel(9) regular-event)
    then have cs e' i = cs d i
      using assms(1) assms(3) assms(6) distributed-system.same-cs-implies-same-resulting-cs
distributed-system-axioms regular-event by blast
    then have cs d i = cs e i
      by (metis Recv-ev' ⟨i = i'⟩ ⟨i' ≠ i''⟩ assms(2) assms(4) event.sel(9)
no-cs-change-if-no-event regular-event)
    then show ?thesis
      by (simp add: ⟨cs e' i = cs d i⟩)
  next
    assume i = i''
    then have cs d i = cs c i
      by (metis Recv-ev ⟨i' ≠ i''⟩ assms(1) assms(3) event.sel(9) no-cs-change-if-no-event
regular-event)
    then have cs e i = cs d' i
      using assms(2) assms(4) assms(5) regular-event same-cs-implies-same-resulting-cs
by blast
    then have cs d' i = cs e' i
      by (metis Recv-ev ⟨i = i''⟩ ⟨i' ≠ i''⟩ assms(3) assms(6) event.sel(9)
no-cs-change-if-no-event regular-event)
    then show ?thesis
      by (simp add: ⟨cs e i = cs d' i⟩)
  qed
next
  case False
  then show ?thesis
    by (metis Recv-ev Recv-ev' assms(1) assms(2) assms(5) assms(6) next-recv)
  qed
qed

```

**lemma** regular-event-implies-same-channel-snapshot-Recv:  
**assumes**

$c \vdash ev \mapsto d$  **and**  
 $d \vdash ev' \mapsto e$  **and**  
 $\sim isRecv\ ev$  **and**  
*regular-event*  $ev$  **and**  
 $isRecv\ ev'$  **and**  
 $c \vdash ev' \mapsto d'$  **and**  
 $d' \vdash ev \mapsto e'$  **and**  
*occurs-on*  $ev \neq occurs-on\ ev'$   
**shows**  
 $cs\ e\ i = cs\ e'\ i$   
**proof** –  
**let**  $?p = occurs-on\ ev$   
**let**  $?q = occurs-on\ ev'$   
**obtain**  $i' s u'' u''' m'$  **where**  $Recv: ev' = Recv\ i'\ ?q\ s\ u''\ u''' m'$   
**by** (*metis*  $assms(5)$  *event.collapse(3)*)  
**show**  $?thesis$   
**proof** (*cases*  $i = i'$ )  
**case** *True*  
**then have**  $cs\ d\ i = cs\ c\ i$   
**using**  $assms(1)$   $assms(3)$   $assms(7)$  *no-cs-change-if-no-event*  $\langle regular-event\ ev \rangle$   $\langle \sim isRecv\ ev \rangle$  **by** *auto*  
**then have**  $cs\ e\ i = cs\ d'\ i$   
**using**  $assms(2)$   $assms(5)$   $assms(6)$  *regular-event same-cs-implies-same-resulting-cs*  
**by** *blast*  
**then have**  $cs\ d'\ i = cs\ e'\ i$   
**using** *True*  $assms(3)$   $assms(6)$   $assms(7)$  *no-cs-change-if-no-event*  $\langle regular-event\ ev \rangle$   $\langle \sim isRecv\ ev \rangle$  **by** *auto*  
**then show**  $?thesis$   
**by** (*simp* *add:*  $\langle cs\ e\ i = cs\ d'\ i \rangle$ )  
**next**  
**case** *False*  
**then have**  $cs\ d\ i = cs\ c\ i$   
**using**  $assms(1)$   $assms(3)$   $assms(4)$  *no-cs-change-if-no-event* **by** *auto*  
**then have**  $cs\ d'\ i = cs\ e\ i$   
**by** (*metis* (*no-types*, *lifting*)  $assms(2)$   $assms(5)$   $assms(6)$  *regular-event same-cs-implies-same-resulting-cs*)  
**then show**  $cs\ e\ i = cs\ e'\ i$   
**using**  $assms(3)$   $assms(4)$   $assms(7)$  *no-cs-change-if-no-event* **by** *auto*  
**qed**  
**qed**

**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
proof (cases channel i = None)
  case True
  then show ?thesis
    using assms(2) assms(3) assms(4) no-msgs-change-if-no-channel by auto
next
  case False
  then obtain p q where chan: channel i = Some (p, q) by auto
  have isSnapshot ev  $\vee$  isRecvMarker ev
    using assms(5) event.exhaust-disc by auto
  then show ?thesis
  proof (elim disjE)
    assume isSnapshot ev
    then obtain r where Snapshot: ev = Snapshot r by (meson isSnapshot-def)
    then show ?thesis
    proof (cases r = p)
      case True
      then have msgs c' i = msgs c i @ [Marker] using chan Snapshot assms by
simp
      moreover have msgs d' i = msgs d i @ [Marker] using chan Snapshot assms
True by simp
      ultimately show ?thesis using assms by simp
    next
      case False
      then have msgs c' i = msgs c i using chan Snapshot assms by simp
      moreover have msgs d' i = msgs d i using chan Snapshot assms False by
simp
      ultimately show ?thesis using assms by simp
    qed
  next
  assume isRecvMarker ev
  then obtain i' r s where RecvMarker: ev = RecvMarker i' r s
    by (meson isRecvMarker-def)
  then show ?thesis
  proof (cases has-snapshotted c r)
    case snap: True
    then show ?thesis
    proof (cases i = i')
      case True
      then have Marker # msgs c' i = msgs c i using chan RecvMarker assms
snap by simp
      moreover have Marker # msgs d' i = msgs d i using chan RecvMarker
assms snap True by simp
      ultimately show ?thesis using assms by (metis list.inject)
    next
      case False
      then have msgs d' i = msgs d i
        using RecvMarker assms(1) assms(4) snap by auto
      also have ... = msgs c i using assms by simp

```

```

    also have ... = msgs c' i
      using False RecvMarker snap assms by auto
    finally show ?thesis using snap by simp
  qed
next
case no-snap: False
then show ?thesis
proof (cases i = i')
  case True
  then have Marker # msgs c' i = msgs c i using chan RecvMarker assms
by simp
  moreover have Marker # msgs d' i = msgs d i using chan RecvMarker
assms True by simp
  ultimately show ?thesis using assms by (metis list.inject)
next
case not-i: False
then show ?thesis
proof (cases r = p)
  case True
  then have msgs c' i = msgs c i @ [Marker]
    using no-snap RecvMarker assms True chan not-i by auto
  moreover have msgs d' i = msgs d i @ [Marker]
  proof -
    have ~ has-snapshotted d r using assms no-snap True by simp
    then show ?thesis
      using no-snap RecvMarker assms True chan not-i by auto
  qed
  ultimately show ?thesis using assms by simp
next
case False
  then have msgs c i = msgs c' i using False RecvMarker no-snap chan
assms not-i by simp
  moreover have msgs d' i = msgs d i
  proof -
    have ~ has-snapshotted d r using assms no-snap False by simp
    then show ?thesis
      using False RecvMarker no-snap chan assms not-i by simp
  qed
  ultimately show ?thesis using assms by simp
qed
qed
qed
qed
qed

```

**lemma same-cs-2:**

**assumes**

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

```

    c ⊢ ev ↦ c' and
    d ⊢ ev ↦ d'
  shows
    cs c' i = cs d' i
proof (cases channel i = None)
  case True
  then show ?thesis
    using assms(2) assms(3) assms(4) no-cs-change-if-no-channel by auto
next
  case False
  then obtain p q where chan: channel i = Some (p, q) by auto
  then show ?thesis
proof (cases ev)
  case (Snapshot r)
  with assms chan show ?thesis by (cases r = q, auto)
next
  case (RecvMarker i' r s)
  then show ?thesis
proof (cases has-snapshotted c r)
  case snap: True
  then have sdr: has-snapshotted d r using assms by auto
  then show ?thesis
proof (cases i = i')
  case True
  then have cs c' i = (fst (cs c i), Done)
    using RecvMarker assms(3) next-recv-marker by blast
  also have ... = cs d' i
    using RecvMarker True assms(2) assms(4) by auto
  finally show ?thesis using True by simp
next
  case False
  then have cs c' i = cs c i using RecvMarker assms snap by auto
  also have ... = cs d' i using RecvMarker assms snap sdr False by auto
  finally show ?thesis by simp
qed
next
  case no-snap: False
  then have nsdr: ~ has-snapshotted d r using assms by blast
  show ?thesis
proof (cases i = i')
  case True
  then have cs c' i = (fst (cs c i), Done)
    using RecvMarker assms(3) next-recv-marker by blast
  also have ... = cs d' i
    using RecvMarker True assms(2) assms(4) by auto
  finally show ?thesis using True by simp
next
  case not-i: False
  with assms RecvMarker chan no-snap show ?thesis by (cases r = q, auto)

```

```

    qed
  qed
next
  case (Trans r u u')
  then show ?thesis
  by (metis assms(2) assms(3) assms(4) event.disc(1) regular-event same-cs-implies-same-resulting-cs)
next
  case (Send i' r s u u' m)
  then show ?thesis
  by (metis assms(2) assms(3) assms(4) event.disc(7) regular-event same-cs-implies-same-resulting-cs)
next
  case (Recv i' r s u u' m)
  then show ?thesis
  by (metis assms(2) assms(3) assms(4) event.disc(13) regular-event same-cs-implies-same-resulting-cs)
qed
qed

```

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

```

assumes
  c ⊢ ev ↦ d and
  d ⊢ ev' ↦ e and
  isSnapshot ev and
  isTrans ev' and
  c ⊢ ev' ↦ d' and
  d' ⊢ ev ↦ e' and
  occurs-on ev ≠ occurs-on ev'
shows
  msgs e i = msgs e' i
proof –
  let ?p = occurs-on ev
  let ?q = occurs-on ev'
  have ev = Snapshot ?p
  by (metis assms(3) event.collapse(4))
  obtain u'' u''' where ev' = Trans ?q u'' u'''
  by (metis assms(4) event.collapse(1))
  have msgs c i = msgs d' i
  using Trans-msg assms(4) assms(5) by blast
  then have msgs e' i = msgs d i
  proof –
    have ∀ p. has-snapshotted c p = has-snapshotted d' p
    using assms(4) assms(5) regular-event-preserves-process-snapshots by auto
    moreover have msgs c i = msgs d' i using ⟨msgs c i = msgs d' i⟩ by auto
    moreover have c ⊢ ev ↦ d using assms by auto
    moreover have d' ⊢ ev ↦ e' using assms by auto
    moreover have ~ regular-event ev using assms by auto
    ultimately show ?thesis by (blast intro: same-messages-2[symmetric])
  qed
  then have msgs d i = msgs e i
  using Trans-msg assms(2) assms(4) by blast

```

**then show** *?thesis*  
**by** (*simp add: <msgs e' i = msgs d i>*)  
**qed**

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

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

**shows**  
 $msgs\ e'\ i = msgs\ e\ i$

**proof** –

**have** *nr: ~ regular-event ev*  
**using** *assms(3) nonregular-event* **by** *blast*  
**let** *?p = occurs-on ev*  
**let** *?q = occurs-on ev'*  
**obtain** *i' r* **where** *RecvMarker: ev = RecvMarker i' ?p r*  
**by** (*metis assms(3) event.collapse(5)*)  
**obtain** *u'' u'''* **where** *Trans: ev' = Trans ?q u'' u'''*  
**by** (*metis assms(4) event.collapse(1)*)  
**have**  $msgs\ c\ i = msgs\ d'\ i$   
**using** *Trans-msg assms(4) assms(5)* **by** *blast*  
**then have**  $msgs\ e'\ i = msgs\ d\ i$

**proof** –

**have**  $\forall p. has-snapshotted\ d'\ p = has-snapshotted\ c\ p$   
**using** *assms(4) assms(5) regular-event-preserves-process-snapshots* **by** *auto*  
**moreover have**  $\sim\ regular-event\ ev$  **using** *assms* **by** *auto*  
**moreover have**  $\forall n. msgs\ d'\ n = msgs\ c\ n$   
**by** (*metis Trans assms(5) local.next.simps(3)*)  
**ultimately show** *?thesis*  
**using** *assms(1) assms(6) same-messages-2* **by** *blast*

**qed**

**thm** *same-messages-2*

**then have**  $msgs\ d\ i = msgs\ e\ i$   
**using** *Trans-msg assms(2) assms(4)* **by** *blast*  
**then show** *?thesis*  
**by** (*simp add: <msgs e' i = msgs d i>*)

**qed**

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

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

```

    c ⊢ ev' ↦ d' and
    d' ⊢ ev ↦ e' and
    occurs-on ev ≠ occurs-on ev'
  shows
    msgs e i = msgs e' i
  using assms swap-Snapshot-Trans by auto

lemma swap-Send-Snapshot:
  assumes
    c ⊢ ev ↦ d and
    d ⊢ ev' ↦ e and
    isSend ev and
    isSnapshot ev' and
    c ⊢ ev' ↦ d' and
    d' ⊢ ev ↦ e' and
    occurs-on ev ≠ occurs-on ev'
  shows
    msgs e i = msgs e' i
  proof (cases channel i = None)
    case True
      then show ?thesis
        by (metis assms(1) assms(2) assms(5) assms(6) no-msgs-change-if-no-channel)
    next
      case False
        then obtain p q where chan: channel i = Some (p, q) by auto
        let ?p = occurs-on ev
        let ?q = occurs-on ev'
        obtain i' r u u' m where Send: ev = Send i' ?p r u u' m
          by (metis assms(3) event.collapse(2))
        have Snapshot: ev' = Snapshot ?q
          by (metis assms(4) event.collapse(4))
        show ?thesis
          proof (cases i = i'; cases p = ?q)
            assume asm: i = i' p = ?q
            then have ?p = p
              proof -
                have channel i' = Some (p, q) using chan asm by simp
                then show ?thesis using assms can-occur-def Send chan
                  by (metis (mono-tags, lifting) event.simps(27) happen-implies-can-occur
                    option.inject prod.inject)
              qed
            then show ?thesis using assms asm by simp
          next
            assume i = i' p ≠ ?q
            then have msgs d i = msgs c i @ [Msg m]
              by (metis Send assms(1) next-send)
            moreover have msgs e i = msgs d i
              by (metis Pair-inject Snapshot ⟨p ≠ occurs-on ev'⟩ assms(2) chan next-snapshot
                option.inject)
          next

```

**moreover have**  $msgs\ d'\ i = msgs\ c\ i$   
**by** (*metis Pair-inject Snapshot*  $\langle p \neq occurs-on\ ev' \rangle\ assms(5)\ chan\ next-snapshot\ option.inject$ )  
**moreover have**  $msgs\ e'\ i = msgs\ d'\ i\ @\ [Msg\ m]$   
**by** (*metis Send*  $\langle i = i' \rangle\ assms(6)\ next-send$ )  
**ultimately show** *?thesis* **by** *simp*  
**next**  
**assume**  $asm: i \neq i'\ p = ?q$   
**then have**  $msgs\ d\ i = msgs\ c\ i$   
**by** (*metis Send*  $assms(1)\ assms(3)\ event.sel(8)\ msgs-unchanged-for-other-is\ regular-event$ )  
**moreover have**  $msgs\ e\ i = msgs\ c\ i\ @\ [Marker]$   
**by** (*metis (full-types) Snapshot*  $asm(2)\ assms(2)\ calculation\ chan\ next-snapshot$ )  
**moreover have**  $msgs\ d'\ i = msgs\ c\ i\ @\ [Marker]$   
**by** (*metis (full-types) Snapshot*  $asm(2)\ assms(5)\ chan\ next-snapshot$ )  
**moreover have**  $msgs\ e'\ i = msgs\ d'\ i$   
**by** (*metis Send*  $asm(1)\ assms(6)\ next-send$ )  
**ultimately show** *?thesis* **by** *simp*  
**next**  
**assume**  $i \neq i'\ p \neq ?q$   
**then have**  $msgs\ c\ i = msgs\ d\ i$   
**by** (*metis Send*  $assms(1)\ assms(3)\ event.sel(8)\ msgs-unchanged-for-other-is\ regular-event$ )  
**then have**  $msgs\ e\ i = msgs\ d'\ i$   
**by** (*metis Pair-inject Snapshot*  $\langle p \neq occurs-on\ ev' \rangle\ assms(2,5)\ chan\ next-snapshot\ option.inject$ )  
**then show** *?thesis*  
**by** (*metis Send*  $\langle i \neq i' \rangle\ assms(6)\ next-send$ )  
**qed**  
**qed**

**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**

$msgs\ e\ i = msgs\ e'\ i$

**using** *assms swap-Send-Snapshot* **by** *auto*

**lemma** *swap-Recv-Snapshot*:

**assumes**

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

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

*isRecv*  $ev$  **and**

```

    isSnapshot ev' and
    c ⊢ ev' ↦ d' and
    d' ⊢ ev ↦ e' and
    occurs-on ev ≠ occurs-on ev'
  shows
    msgs e i = msgs e' i
proof (cases channel i = None)
  case True
  then show ?thesis
  by (metis assms(1) assms(2) assms(5) assms(6) no-msgs-change-if-no-channel)
next
  case False
  then obtain p q where chan: channel i = Some (p, q) by auto
  let ?p = occurs-on ev
  let ?q = occurs-on ev'
  obtain i' r u u' m where Recv: ev = Recv i' ?p r u u' m
  by (metis assms(3) event.collapse(3))
  have Snapshot: ev' = Snapshot ?q
  by (metis assms(4) event.collapse(4))
  show ?thesis
proof (cases i = i'; cases p = ?q)
  assume i = i' p = ?q
  then have Msg m # msgs d i = msgs c i
  by (metis Recv assms(1) next-recv)
  moreover have msgs e i = msgs d i @ [Marker]
  by (metis (full-types) Snapshot ⟨p = occurs-on ev'⟩ assms(2) chan next-snapshot)
  moreover have msgs d' i = msgs c i @ [Marker]
  by (metis (full-types) Snapshot ⟨p = occurs-on ev'⟩ assms(5) chan next-snapshot)
  moreover have Msg m # msgs e' i = msgs d' i
  by (metis Recv ⟨i = i'⟩ assms(6) next-recv)
  ultimately show ?thesis
  by (metis list.sel(3) neq-Nil-conv tl-append2)
next
  assume i = i' p ≠ ?q
  then have Msg m # msgs d i = msgs c i
  by (metis Recv assms(1) next-recv)
  moreover have msgs e i = msgs d i
  by (metis Pair-inject Snapshot ⟨p ≠ occurs-on ev'⟩ assms(2) chan next-snapshot
  option.inject)
  moreover have msgs d' i = msgs c i
  by (metis Pair-inject Snapshot ⟨p ≠ occurs-on ev'⟩ assms(5) chan next-snapshot
  option.inject)
  moreover have Msg m # msgs e' i = msgs d' i
  by (metis Recv ⟨i = i'⟩ assms(6) next-recv)
  ultimately show ?thesis by (metis list.inject)
next
  assume i ≠ i' p = ?q
  then have msgs d i = msgs c i
  by (metis Recv assms(1) next-recv)

```

**moreover have**  $msgs\ e\ i = msgs\ d\ i @ [Marker]$   
**by** (*metis* (*full-types*) *Snapshot*  $\langle p = occurs-on\ ev' \rangle\ assms(2)\ chan\ next-snapshot$ )  
**moreover have**  $msgs\ d'\ i = msgs\ c\ i @ [Marker]$   
**by** (*metis* (*full-types*) *Snapshot*  $\langle p = occurs-on\ ev' \rangle\ assms(5)\ chan\ next-snapshot$ )  
**moreover have**  $msgs\ e'\ i = msgs\ d'\ i$   
**by** (*metis* *Recv*  $\langle i \sim = i' \rangle\ assms(6)\ next-recv$ )  
**ultimately show** *?thesis* **by** *simp*  
**next**  
**assume**  $i \neq i'\ p \neq ?q$   
**then have**  $msgs\ d\ i = msgs\ c\ i$   
**by** (*metis* *Recv* *assms(1)* *next-recv*)  
**moreover have**  $msgs\ e\ i = msgs\ d\ i$   
**by** (*metis* *Pair-inject* *Snapshot*  $\langle p \neq occurs-on\ ev' \rangle\ assms(2)\ chan\ next-snapshot$   
*option.inject*)  
**moreover have**  $msgs\ d'\ i = msgs\ c\ i$   
**by** (*metis* *Pair-inject* *Snapshot*  $\langle p \neq occurs-on\ ev' \rangle\ assms(5)\ chan\ next-snapshot$   
*option.inject*)  
**moreover have**  $msgs\ e'\ i = msgs\ d'\ i$   
**by** (*metis* *Recv*  $\langle i \sim = i' \rangle\ assms(6)\ next-recv$ )  
**ultimately show** *?thesis* **by** *auto*  
**qed**  
**qed**

**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**  
 $msgs\ e\ i = msgs\ e'\ i$   
**using** *assms* *swap-Recv-Snapshot* **by** *auto*

**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** (*cases* *channel*  $i = None$ )  
**case** *True*

```

then show ?thesis
  by (metis assms(1) assms(2) assms(5) assms(6) no-msgs-change-if-no-channel)
next
  case False
  then obtain p q where chan: channel i = Some (p, q) by auto
  obtain i' p' r u u' m where Recv: ev = Recv i' p' r u u' m
    by (metis assms(3) event.collapse(3))
  obtain i'' q' s where RecvMarker: ev' = RecvMarker i'' q' s
    by (metis assms(4) event.collapse(5))
  have i' ≠ i''
  proof (rule ccontr)
    assume ~ i' ≠ i''
    then have channel i' = channel i'' by auto
    then have Some (r, p') = Some (s, q') using assms can-occur-def Recv Recv-
Marker by simp
    then show False using assms
      by (metis Recv RecvMarker event.sel(3,5) option.inject prod.inject)
  qed
  then show ?thesis
  proof (cases i = i' ∨ i = i'')
    case True
    then show ?thesis
    proof (elim disjE)
      assume i = i'
      then have pqrp: (p, q) = (r, p')
      by (metis Recv assms(1) chan distributed-system.can-occur-Recv distributed-system-axioms
next-recv option.inject)
      then show ?thesis
      proof (cases has-snapshot c q')
        case snap: True
        then have Msg m # msgs d i = msgs c i
          by (metis Recv ⟨i = i'⟩ assms(1) next-recv)
        moreover have msgs c i = msgs d' i
          using RecvMarker ⟨i = i'⟩ ⟨i' ≠ i''⟩ assms(5) msgs-unchanged-if-snapshot-RecvMarker-for-other-is
snap by blast
        moreover have msgs d i = msgs e i
          using RecvMarker ⟨i = i'⟩ ⟨i' ≠ i''⟩ assms(1) assms(2) snap snap-
shot-state-unchanged by auto
        moreover have Msg m # msgs e' i = msgs d' i
          by (metis Recv ⟨i = i'⟩ assms(6) next-recv)
        ultimately show ?thesis by (metis list.inject)
      next
      case no-snap: False
      then have msgs-d: Msg m # msgs d i = msgs c i
        by (metis Recv ⟨i = i'⟩ assms(1) next-recv)
      then show ?thesis
      proof (cases q' = r)
        case True
        then have msgs d' i = msgs c i @ [Marker]

```

```

proof –
  have channel  $i = \text{Some } (q', q)$ 
    using True chan pgrp by blast
  then show ?thesis using RecvMarker assms no-snap
    by (simp add: no-snap  $\langle i = i' \rangle \langle i' \neq i'' \rangle$ )
qed
moreover have Msg  $m \# \text{msgs } e' i = \text{msgs } d' i$ 
  by (metis Recv  $\langle i = i' \rangle$  assms(6) next-recv)
moreover have msgs  $e i = \text{msgs } d i @ [\text{Marker}]$ 
proof –
  have ps  $d q' = ps c q'$ 
    using assms(1) assms(7) no-state-change-if-no-event RecvMarker by
auto
  then show ?thesis
    using RecvMarker  $\langle i = i' \rangle \langle i' \neq i'' \rangle$  assms True chan no-snap pgrp by
simp
qed
ultimately show ?thesis using msgs-d
  by (metis append-self-conv2 list.inject list.sel(3) message.distinct(1)
tl-append2)
next
  case False
  then have msgs  $e i = \text{msgs } d i$ 
  proof –
    have  $\sim$  has-snapshotted  $d q'$ 
    using assms(1) assms(7) no-snap no-state-change-if-no-event RecvMarker
by auto
  moreover have  $\nexists r. \text{channel } i = \text{Some } (q', r)$  using chan False pgrp by
auto
  moreover have  $i \neq i''$  using  $\langle i = i' \rangle \langle i' \neq i'' \rangle$  by simp
  ultimately show ?thesis using RecvMarker assms by simp
qed
moreover have msgs  $d' i = \text{msgs } c i$ 
proof –
  have  $\nexists r. \text{channel } i = \text{Some } (q', r)$ 
    using False chan pgrp by auto
  moreover have  $i \neq i''$  using  $\langle i = i' \rangle \langle i' \neq i'' \rangle$  by simp
  ultimately show ?thesis using RecvMarker assms(5) no-snap by auto
qed
moreover have Msg  $m \# \text{msgs } e' i = \text{msgs } d' i$ 
  by (metis Recv  $\langle i = i' \rangle$  assms(6) next-recv)
ultimately show ?thesis using msgs-d
  by (metis list.inject)
qed
qed
next
  assume  $i = i''$ 
  then have msgs  $d i = \text{msgs } c i$  using assms
  by (metis Recv  $\langle i' \neq i'' \rangle$  next-recv)

```

**moreover have**  $msgs\ e\ i = msgs\ d'\ i$   
**proof** –  
**have**  $\forall p. has\ snapshotted\ c\ p = has\ snapshotted\ d\ p$   
**by** (*metis Recv assms(1) next-recv*)  
**then show** *?thesis*  
**by** (*meson assms(2) assms(5) calculation same-messages-2 same-messages-imply-same-resulting-message*)  
**qed**  
**moreover have**  $msgs\ e'\ i = msgs\ d'\ i$   
**using** *assms* **by** (*metis Recv <i' ≠ i''> <i = i''> next-recv*)  
**ultimately show** *?thesis* **by** *simp*  
**qed**  
**next**  
**assume** *asm*:  $\sim (i = i' \vee i = i'')$   
**then have**  $msgs\ c\ i = msgs\ d\ i$   
**by** (*metis Recv assms(1) assms(3) event.distinct-disc(16,18) event.sel(9)*  
*msgs-unchanged-for-other-is nonregular-event*)  
**then have**  $msgs\ d'\ i = msgs\ e\ i$   
**proof** –  
**have**  $\forall p. has\ snapshotted\ c\ p = has\ snapshotted\ d\ p$   
**using** *assms(1) assms(3) regular-event-preserves-process-snapshots* **by** *auto*  
**then show** *?thesis*  
**by** (*meson <msgs c i = msgs d i> assms(2) assms(5) same-messages-2*  
*same-messages-imply-same-resulting-messages*)  
**qed**  
**then show** *?thesis*  
**by** (*metis Recv asm assms(3) assms(6) event.distinct-disc(16,18) event.sel(9)*  
*msgs-unchanged-for-other-is nonregular-event*)  
**qed**  
**qed**

**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**  
 $msgs\ e\ i = msgs\ e'\ i$   
**using** *assms swap-msgs-Recv-RecvMarker* **by** *auto*

**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 ⊢ ev' ↦ d' and
  d' ⊢ ev ↦ e' and
  occurs-on ev ≠ occurs-on ev'
shows
  msgs e i = msgs e' i
proof (cases channel i = None)
  case True
  then show ?thesis
  by (metis assms(1) assms(2) assms(5) assms(6) no-msgs-change-if-no-channel)
next
  case False
  then obtain p q where chan: channel i = Some (p, q) by auto
  let ?p = occurs-on ev
  let ?q = occurs-on ev'
  obtain i' p' r u u' m where Send: ev = Send i' p' r u u' m
  by (metis assms(3) event.collapse(2))
  obtain i'' q' s where RecvMarker: ev' = RecvMarker i'' q' s
  by (metis assms(4) event.collapse(5))
  have p' ≠ q' using Send RecvMarker assms by simp
  show ?thesis
  proof (cases i = i'; cases i = i'', goal-cases)
  case 1
  then have msgs e' i = msgs d' i @ [Msg m]
  by (metis Send assms(6) next-send)
  moreover have Marker # msgs d' i = msgs c i using ⟨i = i''⟩ RecvMarker
  assms by simp
  moreover have msgs d i = msgs c i @ [Msg m]
  by (metis 1(1) Send assms(1) next-send)
  moreover have Marker # msgs e i = msgs d i using ⟨i = i''⟩ RecvMarker
  assms by simp
  ultimately show ?thesis
  by (metis append-self-conv2 list.inject list.sel(3) message.distinct(1) tl-append2)
  next
  case 2
  then have pqr: (p, q) = (p', r) using chan Send can-occur-def assms by simp
  then have msgs d i = msgs c i @ [Msg m]
  by (metis 2(1) Send assms(1) next-send)
  moreover have msgs e' i = msgs d' i @ [Msg m]
  by (metis 2(1) Send assms(6) next-send)
  moreover have msgs d' i = msgs c i
  proof –
  have #r. channel i = Some (q', r) using ⟨p' ≠ q'⟩ chan pqr by simp
  with RecvMarker ⟨i ≠ i''⟩ ⟨i = i'⟩ assms show ?thesis by (cases has-snapshotted
  c q', auto)
  qed
  moreover have msgs e i = msgs d i
  proof –
  have #r. channel i = Some (q', r) using ⟨p' ≠ q'⟩ chan pqr by simp
  with RecvMarker ⟨i ≠ i''⟩ ⟨i = i'⟩ assms show ?thesis by (cases has-snapshotted

```

```

d q', auto)
  qed
  ultimately show ?thesis by simp
next
  assume 3: i ≠ i' i = i''
  then have mcd: msgs c i = msgs d i
    by (metis Send assms(1) next-send)
  moreover have msgs e i = msgs d' i
  proof -
    have ∀ p. has-snapshotted c p = has-snapshotted d p
      using assms(1) assms(3) regular-event-preserves-process-snapshots by auto
    moreover have ~ regular-event ev' using assms by auto
    ultimately show ?thesis using mcd assms(2,5) by (blast intro: same-messages-2[symmetric])
  qed
  moreover have msgs e' i = msgs d' i
    by (metis 3(1) Send assms(6) next-send)
  ultimately show ?thesis by simp
next
  assume 4: i ≠ i' i ≠ i''
  have mcd: msgs c i = msgs d i
  by (metis 4(1) Send assms(1) assms(3) event.distinct-disc(12,14) event.sel(8)
  msgs-unchanged-for-other-is nonregular-event)
  have msgs d' i = msgs e i
  proof -
    have ∀ p. has-snapshotted c p = has-snapshotted d p
      using assms(1) assms(3) regular-event-preserves-process-snapshots by auto
    moreover have ~ regular-event ev' using assms by auto
    ultimately show ?thesis using mcd assms(2,5) same-messages-2 by blast
  qed
  moreover have msgs e' i = msgs d' i
    by (metis 4(1) Send assms(6) next-send)
  ultimately show ?thesis by simp
qed
qed

```

**lemma** *swap-RecvMarker-Send:*

```

assumes
  c ⊢ ev ↦ d and
  d ⊢ ev' ↦ e and
  isRecvMarker ev and
  isSend ev' and
  c ⊢ ev' ↦ d' and
  d' ⊢ ev ↦ e' and
  occurs-on ev ≠ occurs-on ev'
shows
  msgs e i = msgs e' i
using assms swap-msgs-Send-RecvMarker by auto

```

**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 (cases channel i = None)
  case True
  then show ?thesis
    by (metis assms(1) assms(2) assms(5) assms(6) no-cs-change-if-no-channel)
next
  case False
  then obtain  $p\ q$  where channel i = Some (p, q) by auto
  have  $nr: \sim\ regular\text{-}event\ ev'$ 
    using assms(4) nonregular-event by blast
  let  $?p = occurs\text{-}on\ ev$ 
  let  $?q = occurs\text{-}on\ ev'$ 
  obtain  $u''\ u'''$  where  $ev = Trans\ ?p\ u''\ u'''$ 
    by (metis assms(3) event.collapse(1))
  have  $ev' = Snapshot\ ?q$ 
    by (metis assms(4) event.collapse(4))
  have  $cs\ d\ i = cs\ c\ i$ 
    by (metis assms(1) assms(3) event.distinct-disc(4) no-cs-change-if-no-event
regular-event)
  then have  $cs\ e\ i = cs\ d'\ i$ 
  proof –
    have  $\forall p. has\text{-}snapshotted\ d\ p = has\text{-}snapshotted\ c\ p$ 
      using assms(1) assms(3) regular-event-preserves-process-snapshots by auto
    then show ?thesis
      using  $\langle cs\ d\ i = cs\ c\ i \rangle$  assms(2) assms(5) same-cs-2 by blast
  qed
  also have  $\dots = cs\ e'\ i$ 
    using assms(3) assms(6) no-cs-change-if-no-event regular-event by blast
  finally show ?thesis by simp
qed

```

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

```

using *swap-cs-Trans-Snapshot* *assms* by *auto*

**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** (*cases channel i = None*)

**case** *True*

**then show** *?thesis*

by (*metis assms(1) assms(2) assms(5) assms(6) no-cs-change-if-no-channel*)

**next**

**case** *False*

**then obtain**  $p\ q$  **where** *channel i = Some (p, q)* **by** *auto*

**have**  $nr: \sim\ regular\text{-}event\ ev'$

using *assms(4) nonregular-event* **by** *blast*

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

**let**  $?q = occurs\text{-}on\ ev'$

**obtain**  $i'\ r\ u\ u'\ m$  **where** *Send: ev = Send i' ?p r u u' m*

by (*metis assms(3) event.collapse(2)*)

**have** *Snapshot: ev' = Snapshot ?q*

by (*metis assms(4) event.collapse(4)*)

**have**  $cs\ d\ i = cs\ c\ i$

by (*metis Send assms(1) next-send*)

**then have**  $cs\ e\ i = cs\ d'\ i$

**proof** –

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

using *assms(1) assms(3) regular-event-preserves-process-snapshots* **by** *auto*

**then show** *?thesis*

using  $\langle cs\ d\ i = cs\ c\ i \rangle$  *assms(2) assms(5) same-cs-2* **by** *blast*

**qed**

**also have**  $\dots = cs\ e'\ i$

using *assms(3) assms(6) no-cs-change-if-no-event regular-event* **by** *blast*

**finally show** *?thesis* **by** *simp*

**qed**

**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**  
 $cs\ e\ i = cs\ e'\ i$   
**using** *swap-cs-Send-Snapshot* *assms* **by** *auto*

**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** (*cases*  $channel\ i = None$ )  
**case** *True*  
**then show** *?thesis*  
**by** (*metis* *assms*(1) *assms*(2) *assms*(5) *assms*(6) *no-cs-change-if-no-channel*)  
**next**  
**case** *False*  
**then obtain**  $p\ q$  **where** *chan*:  $channel\ i = Some\ (p, q)$  **by** *auto*  
**have** *nr*:  $\sim\ regular\ event\ ev'$   
**using** *assms*(4) *nonregular-event* **by** *blast*  
**let**  $?p = occurs\ on\ ev$   
**let**  $?q = occurs\ on\ ev'$   
**obtain**  $i'\ r\ u\ u'\ m$  **where** *Recv*:  $ev = Recv\ i'\ ?p\ r\ u\ u'\ m$   
**by** (*metis* *assms*(3) *event.collapse*(3))  
**have** *Snapshot*:  $ev' = Snapshot\ ?q$   
**by** (*metis* *assms*(4) *event.collapse*(4))  
**show** *?thesis*  
**proof** (*cases*  $i = i'$ )  
**case** *True*  
**then show** *?thesis*  
**proof** (*cases*  $snd\ (cs\ c\ i) = Recording$ )  
**case** *True*  
**then have**  $cs\ d\ i = (fst\ (cs\ c\ i) @ [m], Recording)$  **using** *Recv* *assms* *True*  $\langle i = i' \rangle\ chan$   
**by** (*metis* *next-recv*)  
**moreover have**  $cs\ e\ i = cs\ d\ i$   
**by** (*metis* *Snapshot* *assms*(2) *calculation* *fst-conv* *next-snapshot*)  
**moreover have**  $cs\ c\ i = cs\ d'\ i$   
**by** (*metis* *Snapshot* *True* *assms*(5) *next-snapshot* *prod.collapse*)  
**moreover have**  $cs\ e'\ i = (fst\ (cs\ d'\ i) @ [m], Recording)$   
**by** (*metis* (*mono-tags*, *lifting*) *Recv* *assms*(1) *assms*(6) *calculation*(1) *calculation*(3) *next-recv*)  
**ultimately show** *?thesis* **by** *simp*  
**next**  
**case** *False*

```

have  $cs\ d\ i = cs\ c\ i$ 
  by (metis False Recv assms(1) next-recv)
have  $cs\ e\ i = cs\ d'\ i$ 
proof –
  have  $\forall p. has-snapshot\ d\ p = has-snapshot\ c\ p$ 
    using assms(1) assms(3) regular-event-preserves-process-snapshots by
auto
  then show ?thesis
    using  $\langle cs\ d\ i = cs\ c\ i \rangle$  assms(2) assms(5) same-cs-2 by blast
qed
moreover have  $cs\ d'\ i = cs\ e'\ i$ 
proof –
  have  $cs\ d'\ i = cs\ c\ i$ 
    by (metis Pair-inject Recv Snapshot True assms(1) assms(5) assms(7)
can-occur-Recv distributed-system.happen-implies-can-occur distributed-system.next-snapshot
distributed-system-axioms option.inject)
  then show ?thesis using chan  $\langle i = i' \rangle$  False Recv assms
    by (metis next-recv)
qed
ultimately show ?thesis by simp
qed
next
case False
have  $cs\ d\ i = cs\ c\ i$ 
  by (metis False Recv assms(1) next-recv)
then have  $cs\ e\ i = cs\ d'\ i$ 
proof –
  have  $\forall p. has-snapshot\ d\ p = has-snapshot\ c\ p$ 
    using assms(1) assms(3) regular-event-preserves-process-snapshots by auto
  then show ?thesis
    using  $\langle cs\ d\ i = cs\ c\ i \rangle$  assms(2) assms(5) same-cs-2 by blast
qed
also have  $\dots = cs\ e'\ i$ 
  by (metis False Recv assms(6) next-recv)
finally show ?thesis by simp
qed
qed

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

```

using *swap-cs-Recv-Snapshot* *assms* by *auto*

**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** (*cases channel i = None*)

**case** *True*

**then show** *?thesis*

by (*metis assms(1) assms(2) assms(5) assms(6) no-cs-change-if-no-channel*)

**next**

**case** *False*

**then obtain**  $p\ q$  **where** *chan: channel i = Some (p, q)* **by** *auto*

**have**  $nr: \sim\ regular\text{-}event\ ev'$

using *assms(4) nonregular-event* **by** *blast*

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

**let**  $?q = occurs\text{-}on\ ev'$

**obtain**  $u''\ u'''$  **where**  $ev = Trans\ ?p\ u''\ u'''$

by (*metis assms(3) event.collapse(1)*)

**obtain**  $i'\ s$  **where**  $ev' = RecvMarker\ i'\ ?q\ s$

by (*metis assms(4) event.collapse(5)*)

**have**  $cs\ d\ i = cs\ c\ i$

by (*metis assms(1) assms(3) event.distinct-disc(4) no-cs-change-if-no-event regular-event*)

**then have**  $cs\ e\ i = cs\ d'\ i$

**proof** –

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

using *assms(1) assms(3) regular-event-preserves-process-snapshots* **by** *auto*

**then show** *?thesis*

using  $\langle cs\ d\ i = cs\ c\ i \rangle$  *assms(2) assms(5) same-cs-2* **by** *blast*

**qed**

**also have**  $\dots = cs\ e'\ i$

using *assms(3) assms(6) no-cs-change-if-no-event regular-event* **by** *blast*

**finally show** *?thesis* **by** *simp*

**qed**

**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
     $cs\ e\ i = cs\ e'\ i$ 
using swap-cs-Trans-RecvMarker assms by auto

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 (cases channel i = None)
  case True
    then show ?thesis
      by (metis assms(1) assms(2) assms(5) assms(6) no-cs-change-if-no-channel)
  next
    case False
      then obtain  $p\ q$  where chan: channel i = Some (p, q) by auto
      have  $nr: \sim\ regular\ event\ ev'$ 
        using assms(4) nonregular-event by blast
      let  $?p = occurs\ on\ ev$ 
      let  $?q = occurs\ on\ ev'$ 
      obtain  $i'\ r\ u\ u'\ m$  where Send: ev = Send i' ?p r u u' m
        by (metis assms(3) event.collapse(2))
      obtain  $i''\ s$  where RecvMarker: ev' = RecvMarker i'' ?q s
        by (metis assms(4) event.collapse(5))
      have  $cs\ d\ i = cs\ c\ i$ 
        by (metis assms(1) assms(3) event.distinct-disc(10,12,14) no-cs-change-if-no-event
nonregular-event)
      then have  $cs\ e\ i = cs\ d'\ i$ 
      proof –
        have  $\forall p. has\ snapshotted\ d\ p = has\ snapshotted\ c\ p$ 
          using assms(1) assms(3) regular-event-preserves-process-snapshots by auto
        then show ?thesis
          using  $\langle cs\ d\ i = cs\ c\ i \rangle$  assms(2) assms(5) same-cs-2 by blast
      qed
      also have  $\dots = cs\ e'\ i$ 
        using assms(3) assms(6) no-cs-change-if-no-event regular-event by blast
      finally show ?thesis by simp
qed

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 ⊢ ev' ↦ d' and
    d' ⊢ ev ↦ e'
  shows
    cs e i = cs e' i
  using swap-cs-Send-RecvMarker assms by auto

lemma swap-cs-Recv-RecvMarker:
  assumes
    c ⊢ ev ↦ d and
    d ⊢ ev' ↦ e and
    isRecv ev and
    isRecvMarker ev' and
    c ⊢ ev' ↦ d' and
    d' ⊢ ev ↦ e' and
    occurs-on ev ≠ occurs-on ev'
  shows
    cs e i = cs e' i
  proof (cases channel i = None)
    case True
      then show ?thesis
        by (metis assms(1) assms(2) assms(5) assms(6) no-cs-change-if-no-channel)
    next
      case False
        then obtain p q where chan: channel i = Some (p, q) by auto
        have nr: ~ regular-event ev'
          using assms(4) nonregular-event by blast
        obtain i' p' r u u' m where Recv: ev = Recv i' p' r u u' m
          by (metis assms(3) event.collapse(3))
        obtain i'' q' s where RecvMarker: ev' = RecvMarker i'' q' s
          by (metis assms(4) event.collapse(5))
        have i' ≠ i''
          proof (rule ccontr)
            assume ~ i' ≠ i''
            then have channel i' = channel i'' by simp
            then have (r, p') = (s, q') using Recv RecvMarker assms can-occur-def by
              simp
            then show False using Recv RecvMarker assms can-occur-def by simp
          qed
        show ?thesis
          proof (cases i = i')
            case True
              then have pqrp: (p, q) = (r, p') using Recv assms can-occur-def chan by simp
              then show ?thesis
                proof (cases snd (cs c i))
                  case NotStarted
                    then have cs d i = cs c i using assms Recv ⟨i = i'⟩ by simp
                    moreover have cs d' i = cs e i
  
```

```

proof –
  have  $\forall p. \text{has-snapshotted } c \ p = \text{has-snapshotted } d \ p$ 
    using assms(1) assms(3) regular-event-preserves-process-snapshots by
auto
with assms(2,5) calculation show ?thesis by (blast intro: same-cs-2[symmetric])
qed
thm same-cs-2
moreover have  $cs \ d' \ i = cs \ e' \ i$ 
proof –
  have  $cs \ d' \ i = cs \ c \ i$ 
  proof –
    have  $\nexists r. \text{channel } i = \text{Some } (r, q')$ 
      using Recv RecvMarker assms(7) chan pgrp by auto
      with RecvMarker assms chan <i = i'> <i' ≠ i''> show ?thesis
      by (cases has-snapshotted c q', auto)
    qed
  then show ?thesis using assms Recv <i = i'> NotStarted by simp
qed
ultimately show ?thesis by simp
next
case Done
then have  $cs \ d \ i = cs \ c \ i$  using assms Recv <i = i'> by simp
moreover have  $cs \ d' \ i = cs \ e \ i$ 
proof –
  have  $\forall p. \text{has-snapshotted } c \ p = \text{has-snapshotted } d \ p$ 
    using assms(1) assms(3) regular-event-preserves-process-snapshots by
auto
then show ?thesis using assms(2,5) calculation by (blast intro: same-cs-2[symmetric])
qed
moreover have  $cs \ d' \ i = cs \ e' \ i$ 
proof –
  have  $cs \ d' \ i = cs \ c \ i$ 
  proof –
    have  $\nexists r. \text{channel } i = \text{Some } (r, q')$ 
      using Recv RecvMarker assms(7) chan pgrp by auto
      with RecvMarker assms chan <i = i'> <i' ≠ i''> show ?thesis
      by (cases has-snapshotted c q', auto)
    qed
  then show ?thesis using assms Recv <i = i'> Done by simp
qed
ultimately show ?thesis by simp
next
case Recording
have  $cs \ d \ i = (\text{fst } (cs \ c \ i) \ @ \ [m], \text{Recording})$ 
  using Recording Recv True assms(1) by auto
moreover have  $cs \ e \ i = cs \ d \ i$ 
proof –
  have  $\nexists r. \text{channel } i = \text{Some } (r, q')$ 
    using Recv RecvMarker assms(7) chan pgrp by auto

```

```

    with RecvMarker assms chan ⟨i = i'⟩ ⟨i' ≠ i''⟩ show ?thesis
      by (cases has-snapshotted d q', auto)
  qed
  moreover have cs c i = cs d' i
  proof -
    have ∃r. channel i = Some (r, q')
      using Recv RecvMarker assms(7) chan pgrp by auto
    with RecvMarker assms chan ⟨i = i'⟩ ⟨i' ≠ i''⟩ show ?thesis
      by (cases has-snapshotted c q', auto)
  qed
  moreover have cs e' i = (fst (cs d' i) @ [m], Recording)
    using Recording Recv True assms(6) calculation(3) by auto
  ultimately show ?thesis by simp
  qed
next
case False
have cs d i = cs c i
  using False Recv assms(1) by auto
then have cs e i = cs d' i
  proof -
    have ∀p. has-snapshotted d p = has-snapshotted c p
      using assms(1) assms(3) regular-event-preserves-process-snapshots by auto
    then show ?thesis
      using ⟨cs d i = cs c i⟩ assms(2) assms(5) same-cs-2 by blast
  qed
  also have ... = cs e' i
    using False Recv assms(6) by auto
  finally show ?thesis by simp
  qed
qed
end

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

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

*no-initial-Marker*:  
 $\forall i$ . ( $\exists p q$ . *channel*  $i$  = *Some* ( $p$ ,  $q$ ))  
 $\longrightarrow$  *Marker*  $\notin$  *set* (*msgs init*  $i$ ) **and**

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

*no-initial-process-snapshot*:  
 $\forall p$ .  $\sim$  *has-snapshot* *init*  $p$  **and**

*no-initial-channel-snapshot*:  
 $\forall i$ . *channel-snapshot* *init*  $i$  = ([], *NotStarted*) **and**

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

**begin**

**definition** *has-channel* **where**

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

**proof** –

**obtain**  $p\ q$  **where**  $p : (\text{UNIV} :: 'a \text{ set}) \wedge q : (\text{UNIV} :: 'a \text{ set}) \wedge p \neq q$

**by** (*metis* (*mono-tags*) *One-nat-def UNIV-eq-I all-not-in-conv at-least-two-processes card-Suc-Diff1 card.empty finite-processes insert-iff iso-tuple-UNIV-I less-numeral-extra*(4) *n-not-Suc-n*)

**then have** (*tranclp has-channel*)  $p\ q$  **using** *strongly-connected* **by** *simp*

**then obtain**  $r\ s$  **where** *has-channel*  $r\ s$

**by** (*meson tranclpD*)

**then show** *?thesis* **using** *has-channel-def* **by** *auto*

**qed**

**abbreviation** *S* **where**

$S \equiv s \text{ init}$

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

**assumes** *trace init t final*

**shows**  $\text{channel } cid = \text{None} \implies \text{msgs } (s \text{ init } t\ i)\ cid = []$

**using** *no-messages-introduced-if-no-channel*[*OF assms no-msgs-if-no-channel*] **by** *blast*

**lemma** *S-induct* [*consumes 3, case-names S-init S-step*]:

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

$\bigwedge i. P\ i\ i;$

$\bigwedge i\ j. i < j \implies j \leq \text{length } t \implies (S\ t\ i) \vdash (t!\ i) \mapsto (S\ t\ (\text{Suc } i)) \implies P\ (\text{Suc}$

$i)\ j \implies P\ i\ j$

$\rrbracket \implies P\ i\ j$

**proof** (*induct j – i arbitrary: i*)

**case**  $0$

**then show** *?case* **by** *simp*

**next**

**case** (*Suc n*)

**then have**  $(S\ t\ i) \vdash (t!\ i) \mapsto (S\ t\ (\text{Suc } i))$  **using** *Suc step-Suc* **by** *simp*

**then show** *?case* **using** *Suc* **by** *simp*

**qed**

**lemma** *exists-index*:

**assumes**

*trace init t final* **and**

$ev \in \text{set } (\text{take } (j - i)\ (\text{drop } i\ t))$

**shows**

$\exists k. i \leq k \wedge k < j \wedge ev = t!\ k$

**proof** –

**have**  $trace (S t i) (take (j - i) (drop i t)) (S t j)$   
**by** (*metis* *assms(1)* *assms(2)* *diff-is-0-eq'* *exists-trace-for-any-i-j* *list.distinct(1)* *list.set-cases* *nat-le-linear* *take-eq-Nil*)  
**obtain**  $l$  **where**  $ev = (take (j - i) (drop i t)) ! l l < length (take (j - i) (drop i t))$   
**by** (*metis* *assms(2)* *in-set-conv-nth*)  
**let**  $?k = l + i$   
**have**  $(take (j - i) (drop i t)) ! l = drop i t ! l$   
**using**  $\langle l < length (take (j - i) (drop i t)) \rangle$  **by** *auto*  
**also have**  $\dots = t ! ?k$   
**by** (*metis* *add commute* *assms(2)* *drop-all empty-iff* *list.set(1)* *nat-le-linear* *nth-drop* *take-Nil*)  
**finally have**  $ev = t ! ?k$   
**using**  $\langle ev = take (j - i) (drop i t) ! l \rangle$  **by** *blast*  
**moreover have**  $i \leq ?k \wedge ?k < j$   
**using**  $\langle l < length (take (j - i) (drop i t)) \rangle$  **by** *auto*  
**ultimately show** *?thesis* **by** *blast*  
**qed**

**lemma** *no-change-if-ge-length-t:*

**assumes**  
 $trace\ init\ t\ final$  **and**  
 $i \geq length\ t$  **and**  
 $j \geq i$   
**shows**  
 $S\ t\ i = S\ t\ j$   
**proof** –  
**have**  $trace (S t i) (take (j - i) (drop i t)) (S t j)$   
**using** *assms(1)* *assms(3)* *exists-trace-for-any-i-j* **by** *blast*  
**moreover have**  $(take (j - i) (drop i t)) = Nil$   
**by** (*simp* *add: assms(2)*)  
**ultimately show** *?thesis*  
**by** (*metis* *tr-init* *trace-and-start-determines-end*)  
**qed**

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

**shows**  
 $\llbracket trace\ init\ t\ final; channel\ cid = Some\ (p, q);$   
 $\sim has\_snapshotted\ (S\ t\ i)\ p \rrbracket$   
 $\implies Marker \notin set\ (msgs\ (S\ t\ i)\ cid)$   
**proof** (*induct i*)  
**case**  $0$   
**then show** *?case*  
**by** (*metis* *exists-trace-for-any-i* *no-initial-Marker* *take-eq-Nil* *tr-init* *trace-and-start-determines-end*)  
**next**  
**case** (*Suc n*)  
**then have** *IH: Marker  $\notin$  set (msgs (S t n) cid)*  
**by** (*meson* *distributed-system.exists-trace-for-any-i-j* *distributed-system.snapshot-stable-2* *distributed-system-axioms* *eq-iff* *le-Suc-eq*)

```

    then obtain tr where decomp: trace (S t n) tr (S t (Suc n)) tr = take (Suc n
- n) (drop n t)
    using Suc exists-trace-for-any-i-j le-Suc-eq by blast
    have Marker ∉ set (msgs (S t (Suc n)) cid)
    proof (cases tr = [])
      case True
      then show ?thesis
        by (metis IH decomp(1) tr-init trace-and-start-determines-end)
    next
      case False
      then obtain ev where step: tr = [ev] (S t n) ⊢ ev ↦ (S t (Suc n))
      by (metis One-nat-def Suc-eq-plus1 Suc-leI ⟨tr = take (Suc n - n) (drop n
t)⟩ ⟨trace (S t n) tr (S t (Suc n))⟩ add-diff-cancel-left' append.simps(1) butlast-take
cancel-comm-monoid-add-class.diff-cancel length-greater-0-conv list.distinct(1) list.sel(3)
snoc-eq-iff-butlast take0 take-Nil trace.cases)
      then show ?thesis
      proof (cases ev)
        case (Snapshot p')
        then show ?thesis
          by (metis IH Suc.prem(2) Suc.prem(3) local.step(2) new-Marker-in-set-implies-nonregular-occurrence
new-msg-in-set-implies-occurrence nonregular-event-induces-snapshot snapshot-state-unchanged)
      next
        case (RecvMarker cid' p' q')
        have p' ≠ p
        proof (rule ccontr)
          assume asm: ~ p' ≠ p
          then have has-snapshotted (S t (Suc n)) p
          proof -
            have ~ regular-event ev using RecvMarker by auto
            moreover have occurs-on ev = p using asm RecvMarker by auto
            ultimately show ?thesis using step(2) Suc.hyps Suc.prem
            by (metis nonregular-event-induces-snapshot snapshot-state-unchanged)
          qed
          then show False using Suc.prem by blast
        qed
      moreover have cid ≠ cid'
      proof (rule ccontr)
        assume ~ cid ≠ cid'
        then have hd (msgs (S t n) cid) = Marker ∧ length (msgs (S t n) cid) >
0
        using step RecvMarker can-occur-def by auto
        then have Marker : set (msgs (S t n) cid)
        using list.set-sel(1) by fastforce
        then show False using IH by simp
      qed
      ultimately have msgs (S t (Suc n)) cid = msgs (S t n) cid
      proof -
        have ∃ r. channel cid = Some (p', r)
        using Suc.prem(2) ⟨p' ≠ p⟩ by auto

```

```

      with ⟨cid ≠ cid'⟩ RecvMarker step show ?thesis by (cases has-snapshotted
(S t n) p', auto)
    qed
    then show ?thesis by (simp add: IH)
  next
    case (Trans p' s s')
    then show ?thesis
      using IH local.step(2) by force
  next
    case (Send cid' p' q' s s' m)
    with step IH show ?thesis by (cases cid' = cid, auto)
  next
    case (Recv cid' p' q' s s' m)
    with step IH show ?thesis by (cases cid' = cid, auto)
  qed
  then show ?case by blast
qed

```

## 5.2 Termination

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

**lemma** *must-exist-snapshot*:

```

  assumes
    trace_init t final
  shows
    ∃ p i. Snapshot p = t ! i
proof (rule ccontr)
  assume  $\nexists p i. \text{Snapshot } p = t ! i$ 
  have  $\forall i p. \sim \text{has-snapshotted } (S t i) p$ 
proof (rule allI)
  fix i
  show  $\forall p. \sim \text{has-snapshotted } (S t i) p$ 
proof (induct i)
  case 0
  then show ?case
  by (metis assms distributed-system.trace-and-start-determines-end distributed-system-axioms
exists-trace-for-any-i computation.no-initial-process-snapshot computation-axioms
take0 tr-init)
  next
  case (Suc n)
  then have IH:  $\forall p. \sim \text{has-snapshotted } (S t n) p$  by auto
  then obtain tr where trace (S t n) tr (S t (Suc n)) tr = take (Suc n - n)
(drop n t)
  using assms exists-trace-for-any-i-j le-Suc-eq by blast
  show  $\forall p. \sim \text{has-snapshotted } (S t (Suc n)) p$ 
proof (cases tr = [])

```

```

    case True
    then show ?thesis
    by (metis IH ‹trace (S t n) tr (S t (Suc n))› tr-init trace-and-start-determines-end)
next
    case False
    then obtain ev where step: tr = [ev] (S t n) † ev † (S t (Suc n))
    by (metis One-nat-def Suc-eq-plus1 Suc-leI ‹tr = take (Suc n - n) (drop n
t)› ‹trace (S t n) tr (S t (Suc n))› add-diff-cancel-left' append.simps(1) butlast-take
cancel-comm-monoid-add-class.diff-cancel length-greater-0-conv list.distinct(1) list.sel(3)
snoc-eq-iff-butlast take0 take-Nil trace.cases)
    then show ?thesis
    using step Suc.hyps proof (cases ev)
    case (Snapshot q)
    then show ?thesis
    by (metis ‹ $\exists p i. \text{Snapshot } p = t ! i$ › ‹tr = [ev]› ‹tr = take (Suc n - n)
(drop n t)› append-Cons append-take-drop-id nth-append-length)
    next
    case (RecvMarker cid' q r)
    then have m: Marker ∈ set (msgs (S t n) cid')
    using RecvMarker-implies-Marker-in-set step by blast
    have ~ has-snapshotted (S t n) q using Suc by auto
    then have Marker ∉ set (msgs (S t n) cid')
    proof -
    have channel cid' = Some (r, q) using step can-occur-def RecvMarker
by auto
    then show ?thesis
    using IH assms no-marker-if-no-snapshot by blast
    qed
    then show ?thesis using m by auto
    qed auto
    qed
    qed
    qed
    obtain j p where has-snapshotted (S t j) p using l2 assms by blast
    then show False
    using ‹ $\forall i p. \neg \text{has-snapshotted (S t i) p}$ › by blast
    qed

```

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

```

assumes
  trace init t final and
  ev = RecvMarker cid p q and
  (S t i) † ev † (S t (Suc i))
shows
  has-snapshotted (S t i) q
proof -
  have Marker = hd (msgs (S t i) cid) ∧ length (msgs (S t i) cid) > 0
  proof -
  have Marker ≠ msgs (S t (Suc i)) cid = msgs (S t i) cid

```

```

    using assms(2) assms(3) next-recv-marker by blast
  then show ?thesis
    by (metis length-greater-0-conv list.discI list.sel(1))
qed
then have Marker ∈ set (msgs (S t i) cid)
  using hd-in-set by fastforce
then show has-snapshotted (S t i) q
proof –
  have channel cid = Some (q, p) using assms can-occur-def by auto
  then show ?thesis
    using ⟨Marker ∈ set (msgs (S t i) cid)⟩ assms(1) no-marker-if-no-snapshot
by blast
qed
qed

```

**lemma** *recv-marker-means-cs-Done*:

```

assumes
  trace init t final and
  t ! i = RecvMarker cid p q and
  i < length t
shows
  snd (cs (S t (i+1)) cid) = Done
proof –
  have (S t i) ⊢ (t ! i) ⇨ (S t (i+1))
    using assms(1) assms(3) step-Suc by auto
  then show ?thesis
    by (simp add: assms(2))
qed

```

**lemma** *snapshot-produces-marker*:

```

assumes
  trace init t final and
  ~ has-snapshotted (S t i) p and
  has-snapshotted (S t (Suc i)) p and
  channel cid = Some (p, q)
shows
  Marker : set (msgs (S t (Suc i)) cid) ∨ has-snapshotted (S t i) q
proof –
  obtain ev where ex-ev: (S t i) ⊢ ev ⇨ (S t (Suc i))
    by (metis append-Nil2 append-take-drop-id assms(1) assms(2) assms(3) distributed-system.step-Suc distributed-system-axioms drop-eq-Nil less-Suc-eq-le nat-le-linear not-less-eq s-def)
  then have occurs-on ev = p
    using assms(2) assms(3) no-state-change-if-no-event by force
  then show ?thesis
    using assms ex-ev proof (cases ev)
      case (Snapshot r)
        then have Marker ∈ set (msgs (S t (Suc i)) cid)
          using ex-ev assms(2) assms(3) assms(4) by fastforce

```

```

    then show ?thesis by simp
  next
    case (RecvMarker cid' r s)
    have r = p using ⟨occurs-on ev = p⟩
      by (simp add: RecvMarker)
    then show ?thesis
    proof (cases cid = cid')
      case True
      then have has-snapshotted (S t i) q
        using RecvMarker RecvMarker-implies-Marker-in-set assms(1) assms(2)
        assms(4) ex-ev no-marker-if-no-snapshot by blast
      then show ?thesis by simp
    next
      case False
      then have ∃ s. channel cid = Some (r, s) using RecvMarker assms can-occur-def
        ⟨r = p⟩ by simp
      then have msgs (S t (Suc i)) cid = msgs (S t i) cid @ [Marker]
        using RecvMarker assms ex-ev ⟨r = p⟩ False by simp
      then show ?thesis by simp
    qed
  qed auto
qed

```

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

```

  assumes
    trace init t final
  shows
    ∃ i. ~ has-snapshotted (S t i) p ∧ has-snapshotted (S t (Suc i)) p (is ?Q)
  proof -
    obtain i where has-snapshotted (S t i) p using l2 assms by blast
    let ?j = LEAST j. has-snapshotted (S t j) p
    have ?j ≠ 0
    proof -
      have ~ has-snapshotted (S t 0) p
        by (metis exists-trace-for-any-i list.discI no-initial-process-snapshot s-def
        take-eq-Nil trace.simps)
      then show ?thesis
        by (metis (mono-tags, lifting) ⟨has-snapshotted (S t i) p⟩ wellorder-Least-lemma(1))
    qed
    have ?j ≤ i
      by (meson Least-le ⟨has-snapshotted (S t i) p⟩)
    have ¬ has-snapshotted (S t (?j - 1)) p (is ?P)
    proof (rule ccontr)
      assume ¬ ?P
      then have has-snapshotted (S t (?j - 1)) p by simp
      then have ∃ j. j < ?j ∧ has-snapshotted (S t j) p
        by (metis One-nat-def ⟨(LEAST j. ps (S t j) p ≠ None) ≠ 0⟩ diff-less lessI
        not-gr-zero)
      then show False
    qed
  qed

```

```

    using not-less-Least by blast
  qed
  show ?thesis
  proof (rule ccontr)
    assume  $\neg ?Q$ 
    have  $\forall i. \neg \text{has-snapshotted } (S\ t\ i)\ p$ 
    proof (rule allI)
      fix  $i'$ 
      show  $\neg \text{has-snapshotted } (S\ t\ i')\ p$ 
      proof (induct  $i'$ )
        case 0
        then show ?case
          using  $\langle \text{LEAST } j. \text{ps } (S\ t\ j)\ p \neq \text{None} \neq 0 \rangle$  by force
      next
        case (Suc  $i''$ )
        then show ?case
          using  $\langle \exists i. \neg \text{ps } (S\ t\ i)\ p \neq \text{None} \wedge \text{ps } (S\ t\ (\text{Suc } i))\ p \neq \text{None} \rangle$  by blast
      qed
    qed
  then show False
    using  $\langle \text{ps } (S\ t\ i)\ p \neq \text{None} \rangle$  by blast
  qed
qed

```

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

```

  assumes
    trace init t final
  shows
    has-snapshotted final p
  proof -
    obtain  $i$  where  $\text{has-snapshotted } (S\ t\ i)\ p \wedge i \leq \text{length } t$ 
    using assms l2 by blast
    moreover have  $\text{final} = (S\ t\ (\text{length } t))$ 
    by (metis (no-types, lifting) assms exists-trace-for-any-i le-Suc-eq length-Cons take-Nil take-all trace.simps trace-and-start-determines-end)
    ultimately show ?thesis
      using assms exists-trace-for-any-i-j snapshot-stable by blast
  qed

```

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

```

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

```

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

```

  assumes
    channel cid = Some (p, q)
    trace init t final
  shows
     $\exists !j. \text{next-marker-free-state } t\ i\ cid = j \wedge j \geq i \wedge \text{Marker} \notin \text{set } (\text{msgs } (S\ t\ j))$ 

```

```

cid)
proof (cases Marker ∈ set (msgs (S t i) cid))
  case False
  then have next-marker-free-state t i cid = i unfolding next-marker-free-state-def
    by (metis (no-types, lifting) Least-equality order-refl)
  then show ?thesis using False assms by blast
next
  case True
  then obtain j where j ≥ i Marker ∉ set (msgs (S t j) cid) using l1 assms by
blast
  then show ?thesis
    by (metis (no-types, lifting) LeastI-ex next-marker-free-state-def)
qed

theorem snapshot-algorithm-must-terminate:
  assumes
    trace init t final
  shows
    ∃ phi. ((∀ p. has-snapshotted (S t phi) p)
      ∧ (∀ cid. Marker ∉ set (msgs (S t phi) cid)))
proof -
  let ?i = {i. i ≤ length t ∧ (∀ p. has-snapshotted (S t i) p)}
  have fin-i: finite ?i by auto
  moreover have ?i ≠ empty
  proof -
    have ∀ p. has-snapshotted (S t (length t)) p
      by (meson assms exists-trace-for-any-i-j l2 snapshot-stable-2)
    then show ?thesis by blast
  qed
  then obtain i where asm: ∀ p. has-snapshotted (S t i) p by blast
  have f: ∀ j. j ≥ i → (∀ p. has-snapshotted (S t j) p)
    using snapshot-stable asm exists-trace-for-any-i-j valid assms by blast
  let ?s = (λcid. (next-marker-free-state t i cid)) ‘ { cid. channel cid ≠ None }
  have ?s ≠ empty using exists-some-channel by auto
  have fin-s: finite ?s using finite-channels by simp
  let ?phi = Max ?s
  have ?phi ≥ i
  proof (rule ccontr)
    assume asm: ¬ ?phi ≥ i
    obtain cid p q where g: channel cid = Some (p, q) using exists-some-channel
  by auto
  then have next-marker-free-state t i cid ≥ i using exists-next-marker-free-state
assms by blast
  then have Max ?s ≥ i using Max-ge-iff g fin-s by fast
  then show False using asm by simp
  qed
  then have ∧cid. Marker ∉ set (msgs (S t ?phi) cid)
  proof -
    fix cid

```

```

show Marker  $\notin$  set (msgs (S t ?phi) cid)
proof (cases Marker : set (msgs (S t i) cid))
  case False
  then show ?thesis
  using  $\langle i \leq \text{Max } ?s \rangle$  asm assms exists-trace-for-any-i-j no-markers-if-all-snapshotted
by blast
next
  case True
  then have cpq: channel cid  $\neq$  None using no-messages-if-no-channel assms
by fastforce
  then obtain p q where chan: channel cid = Some (p, q) by auto
  then obtain j where i: j = next-marker-free-state t i cid Marker  $\notin$  set (msgs
(S t j) cid)
    using exists-next-marker-free-state assms by fast
    have j  $\leq$  ?phi using cpq fin-s i(1) pair-imageI by simp
    then show Marker  $\notin$  set (msgs (S t ?phi) cid)
    proof -
      have trace (S t j) (take (?phi - j) (drop j t)) (S t ?phi)
        using  $\langle j \leq ?phi \rangle$  assms exists-trace-for-any-i-j by blast
      moreover have  $\forall p$ . has-snapshotted (S t j) p
        by (metis assms chan f computation.exists-next-marker-free-state compu-
tation-axioms i(1))
      ultimately show ?thesis
      using i(2) no-markers-if-all-snapshotted by blast
    qed
  qed
qed
thus ?thesis using f  $\langle ?phi \geq i \rangle$  by blast
qed

```

### 5.3 Correctness

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

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

```

shows trace init t final  $\implies$  has-snapshotted (S t i) p  $\implies$  j  $\geq$  i  $\implies$  has-snapshotted
(S t j) p
using exists-trace-for-any-i-j snapshot-stable by blast

```

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

```

shows trace init t final  $\implies$   $\sim$  has-snapshotted (S t i) p  $\implies$  i  $\geq$  j  $\implies$   $\sim$ 
has-snapshotted (S t j) p
using snapshot-stable-ver-2 by blast

```

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

**assumes**  
*trace init t final and*  
*has-snapshotted (S t i) p and*  
*~ has-snapshotted (S t i) q and*  
*channel cid = Some (p, q)*  
**shows**  
*Marker : set (msgs (S t i) cid)*  
**proof** –  
**obtain j where** *~ has-snapshotted (S t j) p ∧ has-snapshotted (S t (Suc j)) p*  
**using** *exists-snapshot-for-all-p assms by blast*  
**have** *j ≤ i*  
**proof** (*rule ccontr*)  
**assume** *asm: ~ j ≤ i*  
**then have** *~ has-snapshotted (S t i) p*  
**using** *⟨¬ has-snapshotted (S t j) p ∧ has-snapshotted (S t (Suc j)) p⟩ assms(1)*  
*less-imp-le-nat snapshot-stable-ver-3*  
**by** (*meson nat-le-linear*)  
**then show** *False using assms(2) by simp*  
**qed**  
**have** *i ≤ length t*  
**proof** (*rule ccontr*)  
**assume** *~ i ≤ length t*  
**then have** *i > length t*  
**using** *not-less by blast*  
**obtain i' where** *a: ∀ p. has-snapshotted (S t i') p using assms snapshot-algorithm-must-terminate*  
**by** *blast*  
**have** *i' ≥ i*  
**using** *⟨∀ p. has-snapshotted (S t i') p⟩ assms(1) assms(3) nat-le-linear snapshot-stable-ver-3 by blast*  
**have** *(S t i') ≠ (S t i) using assms a by force*  
**then have** *i ≤ length t*  
**using** *⟨i ≤ i'⟩ assms(1) computation.no-change-if-ge-length-t computation-axioms nat-le-linear by fastforce*  
**then show** *False using ⟨~ i ≤ length t⟩ by simp*  
**qed**  
**have** *marker-in-set: Marker : set (msgs (S t (Suc j)) cid)*  
**using** *⟨¬ has-snapshotted (S t j) p ∧ has-snapshotted (S t (Suc j)) p⟩ ⟨j ≤ i⟩*  
*assms(1) assms(3) assms(4) snapshot-produces-marker snapshot-stable-ver-3 by blast*  
**show** *?thesis*  
**proof** (*rule ccontr*)  
**assume** *asm: Marker ∉ set (msgs (S t i) cid)*  
**then have** *range: (Suc j) < i*  
**by** (*metis Suc-lessI ⟨¬ ps (S t j) p ≠ None ∧ ps (S t (Suc j)) p ≠ None⟩ ⟨j ≤ i⟩*  
*assms(2) marker-in-set order.order-iff-strict*)  
**let** *?k = LEAST k. k ≥ (Suc j) ∧ Marker ∉ set (msgs (S t k) cid)*  
**have** *range-k: (Suc j) < ?k ∧ ?k ≤ i*  
**proof** –  
**have** *j < (LEAST n. Suc j ≤ n ∧ Marker ∉ set (msgs (S t n) cid))*

**by** (*metis* (*full-types*) *Suc-le-eq* *assms(1)* *assms(4)* *exists-next-marker-free-state* *next-marker-free-state-def*)  
**then show** *?thesis*  
**proof** –  
**assume** *a1*:  $j < (\text{LEAST } n. \text{Suc } j \leq n \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ n) \ \text{cid}))$   
**have**  $j < i$   
**using** *local.range* **by** *linarith*  
**then have**  $(\text{Suc } j \leq i \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ i) \ \text{cid})) \wedge (\text{LEAST } n. \text{Suc } j \leq n \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ n) \ \text{cid})) \neq \text{Suc } j$   
**by** (*metis* (*lifting*) *Suc-leI* *asm* *marker-in-set* *wellorder-Least-lemma(1)*)  
**then show** *?thesis*  
**using** *a1* **by** (*simp* *add*: *wellorder-Least-lemma(2)*)  
**qed**  
**qed**  
**have** *a*:  $\text{Marker} : \text{set } (\text{msgs } (S \ t \ (?k-1)) \ \text{cid})$   
**proof** –  
**obtain** *nn* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat* **where**  
 $\forall x0 \ x1. (\exists v2. x0 = \text{Suc } (x1 + v2)) = (x0 = \text{Suc } (x1 + nn \ x0 \ x1))$   
**by** *moura*  
**then have** *f1*:  $(\text{LEAST } n. \text{Suc } j \leq n \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ n) \ \text{cid})) = \text{Suc } (\text{Suc } j + nn \ (\text{LEAST } n. \text{Suc } j \leq n \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ n) \ \text{cid}))) \ (\text{Suc } j)$   
**using**  $\langle \text{Suc } j < (\text{LEAST } k. \text{Suc } j \leq k \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ k) \ \text{cid})) \wedge (\text{LEAST } k. \text{Suc } j \leq k \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ k) \ \text{cid})) \leq i \rangle$  *less-iff-Suc-add*  
**by** *fastforce*  
**have** *f2*:  $\text{Suc } j \leq \text{Suc } j + nn \ (\text{LEAST } n. \text{Suc } j \leq n \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ n) \ \text{cid}))) \ (\text{Suc } j)$   
**by** *simp*  
**have** *f3*:  $\forall p \ n. \neg p \ (n::\text{nat}) \vee \text{Least } p \leq n$   
**by** (*meson* *wellorder-Least-lemma(2)*)  
**have**  $\neg (\text{LEAST } n. \text{Suc } j \leq n \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ n) \ \text{cid})) \leq \text{Suc } j + nn \ (\text{LEAST } n. \text{Suc } j \leq n \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ n) \ \text{cid}))) \ (\text{Suc } j)$   
**using** *f1* **by** *linarith*  
**then have** *f4*:  $\neg (\text{Suc } j \leq \text{Suc } j + nn \ (\text{LEAST } n. \text{Suc } j \leq n \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ n) \ \text{cid}))) \ (\text{Suc } j) \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ (\text{Suc } j + nn \ (\text{LEAST } n. \text{Suc } j \leq n \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ n) \ \text{cid}))) \ (\text{Suc } j))) \ \text{cid})$   
**using** *f3* **by** *force*  
**have**  $\text{Suc } j + nn \ (\text{LEAST } n. \text{Suc } j \leq n \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ n) \ \text{cid}))) \ (\text{Suc } j) = (\text{LEAST } n. \text{Suc } j \leq n \wedge \text{Marker} \notin \text{set } (\text{msgs } (S \ t \ n) \ \text{cid})) - 1$   
**using** *f1* **by** *linarith*  
**then show** *?thesis*  
**using** *f4* *f2* **by** *presburger*  
**qed**  
**have** *b*:  $\text{Marker} \notin \text{set } (\text{msgs } (S \ t \ ?k) \ \text{cid})$   
**using** *assms(1)* *assms(4)* *exists-next-marker-free-state* *next-marker-free-state-def*  
**by** *fastforce*  
**have**  $?k - 1 < i$  **using** *range-k* **by** *auto*  
**then obtain** *ev* **where** *step*:  $(S \ t \ (?k-1)) \vdash \text{ev} \mapsto (S \ t \ (\text{Suc } (?k-1)))$   
**by** (*meson* *Suc-le-eq*  $\langle i \leq \text{length } t \rangle$  *assms(1)* *le-trans* *step-Suc*)

**then show** *False*  
**using** *a* *assms(1)* *assms(3)* *assms(4)* *b* *computation.snapshot-stable-ver-3*  
*computation-axioms less-iff-Suc-add range-k recv-marker-means-snapshotted-2* **by**  
*fastforce*  
**qed**  
**qed**

### 5.3.1 Pre- and postrecording events

**definition** *prerecording-event*:

*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*:

*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*

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

**using** *assms computation.postrecording-event computation-axioms prerecording-event*  
**by** *metis*

**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**  
 $\text{postrecording-event } t i$

**using** *assms computation.postrecording-event computation-axioms prerecording-event*  
**by** *metis*

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

**assumes**  
 $i < j$  **and**  
 $j < \text{length } t$  **and**  
 $\text{neighboring: } \forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t ! k)$  **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) ≠ occurs-on (t ! j)*  
**proof** –  
**let** ?p = *occurs-on (t ! i)*  
**let** ?q = *occurs-on (t ! j)*  
**have** sp: *has-snapshotted (S t i) ?p*  
**using** *assms postrecording-event prerecording-event by blast*  
**have** nsq: *~ has-snapshotted (S t j) ?q*  
**using** *assms postrecording-event prerecording-event by blast*  
**show** ?p ≠ ?q  
**proof** –  
**have** *~ has-snapshotted (S t i) ?q*  
**proof** (*rule ccontr*)  
**assume** sq: *~ ~ has-snapshotted (S t i) ?q*  
**from** *⟨i < j⟩* **have** *i ≤ j* **using** *less-imp-le by blast*  
**then obtain** tr **where** *ex-trace: trace (S t i) tr (S t j)*  
**using** *exists-trace-for-any-i-j valid by blast*  
**then have** *has-snapshotted (S t j) ?q* **using** *ex-trace snapshot-stable sq by blast*  
*blast*  
**then show** *False* **using** *nsq by simp*  
**qed**  
**then show** *?thesis* **using** *sp by auto*  
**qed**  
**qed**

**lemma** *post-before-pre-neighbors:*  
**assumes**  
*i < j and*  
*j < length t and*  
*neighboring: ∀ k. (i < k ∧ k < j) ⟶ ~ regular-event (t ! k) and*  
*post-ei: postrecording-event t i and*  
*pre-ej: prerecording-event t j and*  
*valid: trace init t final*  
**shows**  
*Ball (set (take (j - (i+1)) (drop (i+1) t))) (%ev. ~ regular-event ev ∧ ~ occurs-on ev = occurs-on (t ! j))*  
**proof** –  
**let** ?p = *occurs-on (t ! i)*  
**let** ?q = *occurs-on (t ! j)*  
**let** ?between = *take (j - (i+1)) (drop (i+1) t)*  
**show** *?thesis*  
**proof** (*unfold Ball-def, rule allI, rule impI*)  
**fix** ev  
**assume** *ev : set ?between*  
**have** *len-nr: length ?between = (j - (i+1))* **using** *assms(2) by auto*  
**then obtain** l **where** *?between ! l = ev* **and** *range-l: 0 ≤ l ∧ l < (j - (i+1))*

```

    by (metis ⟨ev ∈ set (take (j - (i + 1)) (drop (i + 1) t))⟩ gr-zeroI in-set-conv-nth
le-numeral-extra(3) less-le)
  let ?k = l + (i+1)
  have ?between ! l = (t ! ?k)
  proof -
    have j < length t
      by (metis assms(2))
    then show ?thesis
      by (metis (no-types) Suc-eq-plus1 Suc-leI add commute assms(1) drop-take
length-take less-diff-conv less-imp-le-nat min.absorb2 nth-drop nth-take range-l)
    qed
  have ~ regular-event ev
    by (metis (no-types, lifting) assms(3) range-l One-nat-def Suc-eq-plus1 ⟨take
(j - (i + 1)) (drop (i + 1) t) ! l = ev⟩ ⟨take (j - (i + 1)) (drop (i + 1) t)
! l = t ! (l + (i + 1))⟩ add.left-commute add-lessD1 lessI less-add-same-cancel2
less-diff-conv order-le-less)
  have step-ev: (S t ?k) ⊢ ev ↦ (S t (?k+1))
  proof -
    have j ≤ length t
      by (metis assms(2) less-or-eq-imp-le)
    then have l + (i + 1) < length t
      by (meson less-diff-conv less-le-trans range-l)
    then show ?thesis
      by (metis (no-types) Suc-eq-plus1 ⟨take (j - (i + 1)) (drop (i + 1) t)
! l = ev⟩ ⟨take (j - (i + 1)) (drop (i + 1) t) ! l = t ! (l + (i + 1))⟩ dis-
tributed-system.step-Suc distributed-system-axioms valid)
    qed
  obtain cid s r where f: ev = RecvMarker cid s r ∨ ev = Snapshot r using
⟨~ regular-event ev⟩
  by (meson isRecvMarker-def isSnapshot-def nonregular-event)
  from f have occurs-on ev ≠ ?q
  proof (elim disjE)
    assume snapshot: ev = Snapshot r
    show ?thesis
    proof (rule ccontr)
      assume occurs-on-q: ~ occurs-on ev ≠ ?q
      then have ?q = r using snapshot by auto
      then have q-snapshotted: has-snapshotted (S t (?k+1)) ?q
        using snapshot step-ev by auto
      then show False
    proof -
      have l + (i + 1) < j
        by (meson less-diff-conv range-l)
      then show ?thesis
    by (metis (no-types) Suc-eq-plus1 Suc-le-eq computation.snapshot-stable-ver-2
computation-axioms pre-ej prerecording-event q-snapshotted valid)
    qed
  qed
  next

```

```

assume RecvMarker:  $ev = \text{RecvMarker } cid \ s \ r$ 
show ?thesis
proof (rule ccontr)
  assume occurs-on-q:  $\sim \text{occurs-on } ev \neq ?q$ 
  then have  $s = ?q$  using RecvMarker by auto
  then have q-snapshotted:  $\text{has-snapshotted } (S \ t \ (?k+1)) \ ?q$ 
  proof (cases has-snapshotted  $(S \ t \ ?k) \ ?q$ )
    case True
      then show ?thesis using snapshot-stable-ver-2 step-Suc step-ev valid by
auto
    next
      case False
        then show  $\text{has-snapshotted } (S \ t \ (?k+1)) \ ?q$ 
          using  $\langle s = ?q \rangle$  next-recv-marker RecvMarker step-ev by auto
        qed
      then show False
    proof –
      have  $l + (i + 1) < j$ 
        using less-diff-conv range-l by blast
      then show ?thesis
        by (metis (no-types) Suc-eq-plus1 Suc-le-eq computation.snapshot-stable-ver-2
computation-axioms pre-ej prerecording-event q-snapshotted valid)
      qed
    qed
  then show  $\neg \text{regular-event } ev \wedge \text{occurs-on } ev \neq ?q$ 
    using  $\langle \sim \text{regular-event } ev \rangle$  by simp
  qed
qed

lemma can-swap-neighboring-pre-and-postrecording-events:
assumes
   $i < j$  and
   $j < \text{length } t$  and
   $\text{occurs-on } (t \ ! \ i) = p$  and
   $\text{occurs-on } (t \ ! \ j) = q$  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
   $\text{can-occur } (t \ ! \ j) \ (S \ t \ i)$ 
proof –
  have  $p \neq q$  using post-before-pre-different-processes assms by auto
  have sp:  $\text{has-snapshotted } (S \ t \ i) \ p$ 
    using assms(3) post-ei postrecording-event prerecording-event by blast
  have nsq:  $\sim \text{has-snapshotted } (S \ t \ j) \ q$ 
    using assms(4) pre-ej prerecording-event by auto

```

```

let ?nr = take (j - (Suc i)) (drop (Suc i) t)
have valid-subtrace: trace (S t (Suc i)) ?nr (S t j)
  using assms(1) exists-trace-for-any-i-j valid by fastforce
have Ball (set ?nr) (%ev. ~ occurs-on ev = q ∧ ~ regular-event ev)
proof -
  have ?nr = take (j - (i+1)) (drop (i+1) t) by auto
  then show ?thesis
    by (metis assms(1) assms(2) assms(4) neighboring post-ei pre-ej valid
post-before-pre-neighbors)
qed
then have la: list-all (%ev. ~ occurs-on ev = q) ?nr
  by (meson list-all-length nth-mem)
have tj-to-tSi: can-occur (t ! j) (S t (Suc i))
proof -
  have list-all (%ev. ~ isSend ev) ?nr
  proof -
    have list-all (%ev. ~ regular-event ev) ?nr
      using ⟨∀ ev∈set (take (j - (Suc i)) (drop (Suc i) t)). occurs-on ev ≠ q ∧
¬ regular-event ev⟩ ⟨list-all (λev. occurs-on ev ≠ q) (take (j - (Suc i)) (drop (Suc
i) t))⟩ list.pred-mono-strong by fastforce
    then show ?thesis
      by (simp add: list.pred-mono-strong)
  qed
  moreover have ~ isRecvMarker (t ! j) using prerecording-event assms by
auto
  moreover have can-occur (t ! j) (S t j)
  proof -
    have (S t j) ⊢ (t ! j) ↦ (S t (Suc j))
      using assms(2) step-Suc valid by auto
    then show ?thesis
      using happen-implies-can-occur by blast
  qed
  ultimately show can-occur (t ! j) (S t (Suc i))
    using assms(4) event-can-go-back-if-no-sender-trace valid-subtrace la by blast
qed
show can-occur (t ! j) (S t i)
proof (cases isSend (t ! i))
  case False
  have ~ isRecvMarker (t ! j) using assms prerecording-event by auto
  moreover have ~ isSend (t ! i) using False by simp
  ultimately show ?thesis
    by (metis ⟨p ≠ q⟩ assms(3) assms(4) event-can-go-back-if-no-sender post-ei
postrecording-event step-Suc tj-to-tSi valid)
  next
  case True
  obtain cid s u u' m where Send: t ! i = Send cid p s u u' m
    by (metis True isSend-def assms(3) event.sel(2))
  have chan: channel cid = Some (p, s)
  proof -

```

```

have can-occur (t ! i) (S t i)
by (meson computation.postrecording-event computation-axioms happen-implies-can-occur
post-ei step-Suc valid)
then show ?thesis using can-occur-def Send by simp
qed
have n: (S t i) ⊢ (t ! i) ⇔ (S t (Suc i))
using assms(1) assms(2) step-Suc valid True by auto
have st: states (S t i) q = states (S t (Suc i)) q
using Send ⟨p ≠ q⟩ n by auto
have isTrans (t ! j) ∨ isSend (t ! j) ∨ isRecv (t ! j)
using assms(7) computation.prerecording-event computation-axioms regu-
lar-event by blast
then show ?thesis
proof (elim disjE)
assume isTrans (t ! j)
then show ?thesis
by (metis (no-types, lifting) tj-to-tSi st can-occur-def assms(4) event.case(1)
event.collapse(1))
next
assume isSend (t ! j)
then obtain cid' s' u'' u''' m' where Send: t ! j = Send cid' q s' u'' u''' m'
by (metis (no-types, lifting) assms(4) event.sel(2) isSend-def)
have co-tSi: can-occur (Send cid' q s' u'' u''' m') (S t (Suc i))
using Send tj-to-tSi by auto
then have channel cid' = Some (q, s') ∧ send cid' q s' u'' u''' m'
using Send can-occur-def by simp
then show ?thesis using can-occur-def st Send assms co-tSi by auto
next
assume isRecv (t ! j)
then obtain cid' s' u'' u''' m' where Recv: t ! j = Recv cid' q s' u'' u''' m'
by (metis assms(4) event.sel(3) isRecv-def)
have co-tSi: can-occur (Recv cid' q s' u'' u''' m') (S t (Suc i))
using Recv tj-to-tSi by auto
then have a: channel cid' = Some (s', q) ∧ length (msgs (S t (Suc i)) cid')
> 0
    ∧ hd (msgs (S t (Suc i)) cid') = Msg m'
using can-occur-def co-tSi by fastforce
show can-occur (t ! j) (S t i)
proof (cases cid = cid')
case False
with Send n have msgs (S t (Suc i)) cid' = msgs (S t i) cid' by auto
then have b: length (msgs (S t i) cid') > 0 ∧ hd (msgs (S t i) cid') = Msg
m'
using a by simp
with can-occur-Recv co-tSi st a Recv show ?thesis
unfolding can-occur-def by auto
next
case True
have stu: states (S t i) q = u''

```

```

    using can-occur-Recv co-tSi st by blast
show ?thesis
proof (rule ccontr)
  have marker-in-set: Marker ∈ set (msgs (S t i) cid)
  proof -
    have (s', q) = (p, q)
    using True a chan by auto
    then show ?thesis
  by (metis (no-types, lifting) True ⟨p ≠ q⟩ a assms(3) marker-must-stay-if-no-snapshot
n no-state-change-if-no-event nsq snapshot-stable-2 sp valid valid-subtrace)
  qed
  assume asm: ~ can-occur (t ! j) (S t i)
  then show False
  proof (unfold can-occur-def, (auto simp add: marker-in-set True Recv
stu))
    assume msgs (S t i) cid' = []
    then show False using marker-in-set
    by (simp add: True)
  next
    assume hd (msgs (S t i) cid') ≠ Msg m'
    have msgs (S t i) cid' ≠ [] using marker-in-set by auto
    then have msgs (S t (Suc i)) cid = msgs (S t i) cid @ [Msg m]
    using Send True n chan by auto
    then have hd (msgs (S t (Suc i)) cid) ≠ Msg m'
    using True ⟨hd (msgs (S t i) cid') ≠ Msg m'⟩ ⟨msgs (S t i) cid' ≠ []⟩
  by auto
    then have ~ can-occur (t ! j) (S t (Suc i))
    using True a by blast
    then show False
    using tj-to-tSi by blast
  next
    assume ~ recv cid' q s' u'' u''' m'
    then show False
    using can-occur-Recv co-tSi by blast
  next
    assume channel cid' ≠ Some (s', q)
    then show False using can-occur-def tj-to-tSi Recv by simp
  qed
qed
qed
qed
qed
qed
qed

```

### 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;
    trace init t final  $\mathbb{I}$ 
 $\implies$  trace init (swap-events i j t) final
     $\wedge (\forall k. k \geq j + 1 \longrightarrow S (\text{swap-events } i \ j \ t) \ k = S \ t \ k)$ 
     $\wedge (\forall k. k \leq i \longrightarrow S (\text{swap-events } i \ j \ t) \ k = S \ t \ k)$ 
     $\wedge$  prerecording-event (swap-events i j t) i
     $\wedge$  postrecording-event (swap-events i j t) (i+1)
     $\wedge (\forall k. k > i+1 \wedge k < j+1$ 
       $\longrightarrow \sim \text{regular-event } ((\text{swap-events } i \ j \ t) \ ! \ k))$ 
proof (induct j - (i+1) arbitrary: j t)
  case 0
  let ?p = occurs-on (t ! i)
  let ?q = occurs-on (t ! j)
  have j = (i+1)
    using 0.premis 0.hyps by linarith
  let ?subt = take (j - (i+1)) (drop (i+1) t)
  have t = take i t @ [t ! i] @ ?subt @ [t ! j] @ drop (j+1) t
  proof -
    have take (Suc i) t = take i t @ [t ! i]
      using 0.premis(2) <j = i + 1> add-lessD1 take-Suc-conv-app-nth by blast
    then show ?thesis
      by (metis (no-types) 0.hyps 0.premis(2) Suc-eq-plus1 <j = i + 1> append-assoc
        append-take-drop-id self-append-conv2 take-Suc-conv-app-nth take-eq-Nil)
  qed
  have sp: has-snapshotted (S t i) ?p
    using 0.premis postrecording-event prerecording-event by blast
  have nsq:  $\sim$  has-snapshotted (S t j) ?q
    using 0.premis postrecording-event prerecording-event by blast
  have ?p  $\neq$  ?q
    using 0.premis computation.post-before-pre-different-processes computation-axioms
  by blast
  have ?subt = Nil
    by (simp add: <j = i + 1>)
  have reg-step-1: (S t i)  $\vdash$  (t ! i)  $\mapsto$  (S t j)
    by (metis 0.premis(2) 0.premis(6) Suc-eq-plus1 <j = i + 1> add-lessD1 step-Suc)
  have reg-step-2: (S t j)  $\vdash$  (t ! j)  $\mapsto$  (S t (j+1))
    using 0.premis(2) 0.premis(6) step-Suc by auto
  have can-occur (t ! j) (S t i)
    using 0.premis can-swap-neighboring-pre-and-postrecording-events by blast
  then obtain d' where new-step1: (S t i)  $\vdash$  (t ! j)  $\mapsto$  d'
    using exists-next-if-can-occur by blast

  have st: states d' ?p = states (S t i) ?p
    using <(S t i)  $\vdash$  t ! j  $\mapsto$  d'> <occurs-on (t ! i)  $\neq$  occurs-on (t ! j)> no-state-change-if-no-event
  by auto
  then have can-occur (t ! i) d'
    using <occurs-on (t ! i)  $\neq$  occurs-on (t ! j)> event-stays-valid-if-no-occurrence
  happen-implies-can-occur new-step1 reg-step-1 by auto
  then obtain e where new-step2: d'  $\vdash$  (t ! i)  $\mapsto$  e

```

```

using exists-next-if-can-occur by blast

have states e = states (S t (j+1))
proof (rule ext)
  fix p
  show states e p = states (S t (j+1)) p
  proof (cases p = ?p ∨ p = ?q)
    case True
      then show ?thesis
      proof (elim disjE)
        assume p = ?p
        then have states d' p = states (S t i) p
          by (simp add: st)
        thm same-state-implies-same-result-state
        then have states e p = states (S t j) p
        using 0.premis(2) 0.premis(6) new-step2 reg-step-1 by (blast intro:same-state-implies-same-result-state)
        moreover have states (S t j) p = states (S t (j+1)) p
          using ⟨occurs-on (t ! i) ≠ occurs-on (t ! j)⟩ ⟨p = occurs-on (t ! i)⟩
no-state-change-if-no-event reg-step-2 by auto
          ultimately show ?thesis by simp
        next
          assume p = ?q
          then have states (S t j) p = states (S t i) p
          using reg-step-1 ⟨occurs-on (t ! i) ≠ occurs-on (t ! j)⟩ no-state-change-if-no-event
by auto
          then have states d' p = states (S t (j+1)) p
            using 0.premis(5) prerecording-event computation-axioms new-step1
reg-step-2 same-state-implies-same-result-state by blast
          moreover have states e p = states (S t (j+1)) p
            using ⟨occurs-on (t ! i) ≠ occurs-on (t ! j)⟩ ⟨p = occurs-on (t ! j)⟩
calculation new-step2 no-state-change-if-no-event by auto
            ultimately show ?thesis by simp
          qed
        next
          case False
          then have states (S t i) p = states (S t j) p
            using no-state-change-if-no-event reg-step-1 by auto
          moreover have ... = states (S t (j+1)) p
            using False no-state-change-if-no-event reg-step-2 by auto
          moreover have ... = states d' p
            using False calculation new-step1 no-state-change-if-no-event by auto
          moreover have ... = states e p
            using False new-step2 no-state-change-if-no-event by auto
          ultimately show ?thesis by simp
          qed
        qed
      qed
    qed
  moreover have msgs e = msgs (S t (j+1))
  proof (rule ext)

```

```

fix cid
have isTrans (t ! i) ∨ isSend (t ! i) ∨ isRecv (t ! i)
  using 0.premis(4) computation.postrecording-event computation-axioms regular-event by blast
moreover have isTrans (t ! j) ∨ isSend (t ! j) ∨ isRecv (t ! j)
  using 0.premis(5) computation.prerecording-event computation-axioms regular-event by blast
ultimately show msgs e cid = msgs (S t (j+1)) cid
proof (elim disjE, goal-cases)
  case 1
  then have msgs d' cid = msgs (S t j) cid
    by (metis Trans-msg new-step1 reg-step-1)
  then show ?thesis
    using Trans-msg ⟨isTrans (t ! i)⟩ ⟨isTrans (t ! j)⟩ new-step2 reg-step-2 by
auto
  next
  case 2
  then show ?thesis
    using ⟨occurs-on (t ! i) ≠ occurs-on (t ! j)⟩ new-step1 new-step2 reg-step-1
reg-step-2 swap-msgs-Trans-Send by auto
  next
  case 3
  then show ?thesis
    using ⟨occurs-on (t ! i) ≠ occurs-on (t ! j)⟩ new-step1 new-step2 reg-step-1
reg-step-2 swap-msgs-Trans-Recv by auto
  next
  case 4
  then show ?thesis
    using ⟨occurs-on (t ! i) ≠ occurs-on (t ! j)⟩ new-step1 new-step2 reg-step-1
reg-step-2 swap-msgs-Send-Trans by auto
  next
  case 5
  then show ?thesis
    using ⟨occurs-on (t ! i) ≠ occurs-on (t ! j)⟩ new-step1 new-step2 reg-step-1
reg-step-2 swap-msgs-Recv-Trans by auto
  next
  case 6
  then show ?thesis
    using ⟨occurs-on (t ! i) ≠ occurs-on (t ! j)⟩ new-step1 new-step2 reg-step-1
reg-step-2 by (blast intro:swap-msgs-Send-Send[symmetric])
  next
  case 7
  then show ?thesis
    using ⟨occurs-on (t ! i) ≠ occurs-on (t ! j)⟩ new-step1 new-step2 reg-step-1
reg-step-2 swap-msgs-Send-Recv by auto
  next
  case 8
  then show ?thesis
    using ⟨occurs-on (t ! i) ≠ occurs-on (t ! j)⟩ new-step1 new-step2 reg-step-1

```

```

reg-step-2 swap-msgs-Send-Recv by simp
  next
  case 9
  then show ?thesis
    using ⟨occurs-on (t ! i) ≠ occurs-on (t ! j)⟩ new-step1 new-step2 reg-step-1
reg-step-2 by (blast intro:swap-msgs-Recv-Recv[symmetric])
  qed
  qed

  moreover have process-snapshot e = process-snapshot (S t (j+1))
  proof (rule ext)
    fix p
    have process-snapshot d' p = process-snapshot (S t j) p
      by (metis 0.prem5(4) 0.prem5(5) computation.postrecording-event computation.prerecording-event computation-axioms new-step1 reg-step-1 regular-event-preserves-process-snapshots)
    then show process-snapshot e p = process-snapshot (S t (j+1)) p
      by (metis 0.prem5(4) 0.prem5(5) computation.postrecording-event computation.prerecording-event computation-axioms new-step2 reg-step-2 regular-event-preserves-process-snapshots)
    qed

  moreover have channel-snapshot e = channel-snapshot (S t (j+1))
  proof (rule ext)
    fix cid
    show cs e cid = cs (S t (j+1)) cid
    proof (cases isRecv (t ! i); cases isRecv (t ! j), goal-cases)
      case 1
      then show ?thesis
        using ⟨?p ≠ ?q⟩ new-step1 new-step2 reg-step-1 reg-step-2
      by (blast intro:regular-event-implies-same-channel-snapshot-Recv-Recv[symmetric])
    next
      case 2
      moreover have regular-event (t ! j) using prerecording-event 0 by simp
      ultimately show ?thesis
      using ⟨?p ≠ ?q⟩ new-step1 new-step2 reg-step-1 reg-step-2 regular-event-implies-same-channel-snapshot-Recv-Recv[symmetric]
    next
      case 3
      assume 3: ~ isRecv (t ! i) isRecv (t ! j)
      moreover have regular-event (t ! i) using postrecording-event 0 by simp
      ultimately show ?thesis
      using ⟨?p ≠ ?q⟩ new-step1 new-step2 reg-step-1 reg-step-2 regular-event-implies-same-channel-snapshot-Recv-Recv[symmetric]
    next
      case 4
      assume 4: ~ isRecv (t ! i) ~ isRecv (t ! j)
      moreover have regular-event (t ! j) using prerecording-event 0 by simp
      moreover have regular-event (t ! i) using postrecording-event 0 by simp
      ultimately show ?thesis
      using ⟨?p ≠ ?q⟩ new-step1 new-step2 reg-step-1 reg-step-2
      by (metis no-cs-change-if-no-event)
  qed

```

```

qed
qed
ultimately have  $e = S t (j+1)$  by simp
then have  $(S t i) \vdash (t ! j) \mapsto d' \wedge d' \vdash (t ! i) \mapsto (S t (j+1))$ 
  using new-step1 new-step2 by blast
then have swap: trace  $(S t i) [t ! j, t ! i] (S t (j+1))$ 
  by (meson trace.simps)
have take  $(j-1) t @ [t ! j, t ! i] = ((take (j+1) t)[i := t ! j])[j := t ! i]$ 
proof -
  have  $i = j - 1$ 
  by (simp add:  $\langle j = i + 1 \rangle$ )
  show ?thesis
  proof (subst (1 2 3)  $\langle i = j - 1 \rangle$ )
    have  $j < \text{length } t$  using 0.prem1 by auto
    then have take  $(j - 1) t @ [t ! j, t ! (j - 1)] @ \text{drop } (j + 1) t = t[j - 1 := t ! j, j := t ! (j - 1)]$ 
      by (metis Suc-eq-plus1  $\langle i = j - 1 \rangle \langle j = i + 1 \rangle$  add-Suc-right arith-special(3) swap-neighbors)
    then show take  $(j - 1) t @ [t ! j, t ! (j - 1)] = (take (j+1) t)[j - 1 := t ! j, j := t ! (j - 1)]$ 
      proof -
        assume a1: take  $(j - 1) t @ [t ! j, t ! (j - 1)] @ \text{drop } (j + 1) t = t[j - 1 := t ! j, j := t ! (j - 1)]$ 
        have f2:  $t[j - 1 := t ! j, j := t ! (j - 1)] = take j (t[j - 1 := t ! j]) @ t ! (j - 1) \# \text{drop } (Suc j) (t[j - 1 := t ! j])$ 
          by (metis (no-types) 0.prem2 length-list-update upd-conv-take-nth-drop)
        have f3:  $\forall n na. \neg n < na \vee Suc n \leq na$ 
          using Suc-leI by blast
        then have min  $(\text{length } t) (j + 1) = j + 1$ 
          by (metis (no-types) 0.prem2 Suc-eq-plus1 min.absorb2)
        then have f4:  $\text{length } ((take (j + 1) t)[j - 1 := t ! j]) = j + 1$ 
          by simp
        have f5:  $j + 1 \leq \text{length } (t[j - 1 := t ! j])$ 
          using f3 by (metis (no-types) 0.prem2 Suc-eq-plus1 length-list-update)
        have  $Suc j \leq j + 1$ 
          by linarith
        then have  $(take (j + 1) (t[j - 1 := t ! j]))[j := t ! (j - 1)] = take j (t[j - 1 := t ! j]) @ t ! (j - 1) \# [] @ []$ 
          using f5 f4 by (metis (no-types) Suc-eq-plus1 add-diff-cancel-right' butlast-conv-take butlast-take drop-eq-Nil lessI self-append-conv2 take-update-swap upd-conv-take-nth-drop)
        then show ?thesis
          using f2 a1 by (simp add: take-update-swap)
      qed
    qed
  qed
  have s: trace init  $(take i t) (S t i)$ 
    using 0.prem6 exists-trace-for-any-i by blast

```

```

have e: trace (S t (j+1)) (take (length t - (j+1)) (drop (j+1) t)) final
proof -
  have trace init (take (length t) t) final
  by (simp add: 0.premis(6))
  then show ?thesis
  by (metis 0.premis(2) Suc-eq-plus1 Suc-leI exists-trace-for-any-i exists-trace-for-any-i-j
nat-le-linear take-all trace-and-start-determines-end)
qed
have trace init (take i t @ [t ! j] @ [t ! i] @ drop (j+1) t) final
proof -
  from s swap have trace init (take i t @ [t ! j, t ! i]) (S t (j+1)) using trace-trans
by blast
  then have trace init (take i t @ [t ! j, t ! i] @ (take (length t - (j+1)) (drop
(j+1) t))) final
  using e trace-trans by fastforce
  moreover have take (length t - (j+1)) (drop (j+1) t) = drop (j+1) t by
simp
  ultimately show ?thesis by simp
qed
moreover have take i t @ [t ! j] @ [t ! i] @ drop (j+1) t = (t[i := t ! j])[j := t
! i]
proof -
  have length (take i t @ [t ! j] @ [t ! i] @ drop (j+1) t) = length ((t[i := t !
j])[j := t ! i])
  by (metis (mono-tags, lifting) ‹t = take i t @ [t ! i] @ take (j - (i + 1))
(drop (i + 1) t) @ [t ! j] @ drop (j + 1) t› ‹take (j - (i + 1)) (drop (i + 1) t)
= []› length-append length-list-update list.size(4) self-append-conv2)
  moreover have  $\bigwedge k. k < \text{length} ((t[i := t ! j])[j := t ! i]) \implies (take i t @ [t !
j] @ [t ! i] @ drop (j+1) t) ! k = ((t[i := t ! j])[j := t ! i]) ! k$ 
proof -
  fix k
  assume  $k < \text{length} ((t[i := t ! j])[j := t ! i])$ 
  show  $(take i t @ [t ! j] @ [t ! i] @ drop (j+1) t) ! k = ((t[i := t ! j])[j := t !
i]) ! k$ 
proof (cases  $k = i \vee k = j$ )
  case True
  then show ?thesis
  proof (elim disjE)
    assume  $k = i$ 
    then show ?thesis
    by (metis (no-types, lifting) ‹ $k < \text{length} (t[i := t ! j, j := t ! i])$ › append-Cons
le-eq-less-or-eq length-list-update length-take min.absorb2 nth-append-length nth-list-update-eq
nth-list-update-neq)
  next
  assume  $k = j$ 
  then show ?thesis
  by (metis (no-types, lifting) 0.premis(4) Suc-eq-plus1 ‹ $j = i + 1$ ›
‹ $k < \text{length} (t[i := t ! j, j := t ! i])$ › append.assoc append-Cons le-eq-less-or-eq
length-append-singleton length-list-update length-take min.absorb2 nth-append-length

```

```

nth-list-update postrecording-event)
  qed
next
case knij: False
then show ?thesis
proof (cases k < i)
  case True
  then show ?thesis
  by (metis (no-types, lifting) 0.prem1(2) ⟨j = i + 1⟩ add-lessD1 length-take
less-imp-le-nat min.absorb2 not-less nth-append nth-list-update-neq nth-take)
  next
  case False
  then have k > j
  using ⟨j = i + 1⟩ knij by linarith
  then have (take i t @ [t ! j] @ [t ! i] @ drop (j+1) t) ! k = drop (j+1) t
! (k-(j+1))
  proof -
  assume a1: j < k
  have f2: ∀ n na. ((n::nat) < na) = (n ≤ na ∧ n ≠ na)
  using nat-less-le by blast
  have f3: i + 0 = min (length t) i + (0 + 0)
  using 0.prem1(2) ⟨j = i + 1⟩ by linarith
  have f4: min (length t) i + Suc (0 + 0) = length (take i t) + length [t
! j]
  by force
  have f5: take i t @ [t ! j] @ [] = take i t @ [t ! j]
  by auto
  have j = length (take i t @ [t ! j] @ [])
  using f3 by (simp add: ⟨j = i + 1⟩)
  then have j + 1 = length (take i t @ [t ! j] @ [t ! i])
  by fastforce
  then show ?thesis
  using f5 f4 f3 f2 a1 by (metis (no-types) One-nat-def ⟨j = i
+ 1⟩ add-Suc-right append.assoc length-append less-antisym list.size(3) not-less
nth-append)
  qed
  moreover have (t[i := t ! j])[j := t ! i] ! k = drop (j+1) ((t[i := t ! j])[j
:= t ! i]) ! (k-(j+1))
  using 0.prem1(2) ⟨j < k⟩ by auto
  moreover have drop (j+1) ((t[i := t ! j])[j := t ! i]) = drop (j+1) t
  using 0.prem1(1) by auto
  ultimately show ?thesis by simp
  qed
qed
qed
ultimately show ?thesis by (simp add: list-eq-iff-nth-eq)
qed
moreover have ∀ k. k ≥ j + 1 ⟶ S t k = S ((t[i := t ! j])[j := t ! i]) k
proof (rule allI, rule impI)

```

**fix**  $k$   
**assume**  $k \geq j + 1$   
**let**  $?newt = ((t[i := t! j])[j := t! i])$   
**have**  $trace\ init\ (take\ k\ ?newt)\ (S\ ?newt\ k)$   
**using**  $calculation(1)\ calculation(2)\ exists\ trace\ for\ any\ i\ \mathbf{by}\ auto$   
**have**  $take\ k\ ?newt = take\ (j+1)\ ?newt\ @\ take\ (k - (j+1))\ (drop\ (j+1)\ ?newt)$   
**by**  $(metis\ \langle j + 1 \leq k \rangle\ le\ add\ diff\ inverse\ take\ add)$   
**have**  $same\ traces:\ drop\ (j+1)\ t = drop\ (j+1)\ ?newt$   
**by**  $(metis\ 0.premis(1)\ Suc\ eq\ plus1\ \langle j = i + 1 \rangle\ drop\ update\ cancel\ less\ SucI\ less\ add\ same\ cancel1)$   
**have**  $trace\ init\ (take\ (j+1)\ ((t[i := t! j])[j := t! i]))\ (S\ t\ (j+1))$   
**by**  $(metis\ (no\ types,\ lifting)\ \langle j = i + 1 \rangle\ \langle take\ (j - 1)\ t\ @\ [t! j,\ t! i] = (take\ (j + 1)\ t)[i := t! j,\ j := t! i] \rangle\ add\ diff\ cancel\ right'\ local.swap\ s\ take\ update\ swap\ trace\ trans)$   
**moreover** **have**  $trace\ init\ (take\ (j+1)\ ?newt)\ (S\ ?newt\ (j+1))$   
**using**  $\langle take\ i\ t\ @\ [t! j]\ @\ [t! i]\ @\ drop\ (j + 1)\ t = t[i := t! j,\ j := t! i] \rangle$   
 $\langle trace\ init\ (take\ i\ t\ @\ [t! j]\ @\ [t! i]\ @\ drop\ (j + 1)\ t)\ final \rangle\ exists\ trace\ for\ any\ i$   
**by**  $auto$   
**ultimately** **have**  $S\ ?newt\ (j+1) = S\ t\ (j+1)$   
**using**  $trace\ and\ start\ determines\ end\ \mathbf{by}\ blast$   
**have**  $trace\ (S\ t\ (j+1))\ (take\ (k - (j+1))\ (drop\ (j+1)\ t))\ (S\ t\ k)$   
**using**  $0.premis(6)\ \langle j + 1 \leq k \rangle\ exists\ trace\ for\ any\ i\ j\ \mathbf{by}\ blast$   
**moreover** **have**  $trace\ (S\ ?newt\ (j+1))\ (take\ (k - (j+1))\ (drop\ (j+1)\ ?newt))$   
 $(S\ ?newt\ k)$   
**using**  $\langle j + 1 \leq k \rangle\ \langle take\ i\ t\ @\ [t! j]\ @\ [t! i]\ @\ drop\ (j + 1)\ t = t[i := t! j,\ j := t! i] \rangle$   
 $\langle trace\ init\ (take\ i\ t\ @\ [t! j]\ @\ [t! i]\ @\ drop\ (j + 1)\ t)\ final \rangle\ exists\ trace\ for\ any\ i\ j\ \mathbf{by}\ fastforce$   
**ultimately** **show**  $S\ t\ k = S\ ?newt\ k$   
**using**  $\langle S\ (t[i := t! j,\ j := t! i])\ (j + 1) = S\ t\ (j + 1) \rangle\ same\ traces$   
 $trace\ and\ start\ determines\ end\ \mathbf{by}\ auto$   
**qed**  
**moreover** **have**  $\forall k. k \leq i \longrightarrow S\ t\ k = S\ ((t[i := t! j])[j := t! i])\ k$   
**proof**  $(rule\ allI,\ rule\ impI)$   
**fix**  $k$   
**assume**  $k \leq i$   
**let**  $?newt = ((t[i := t! j])[j := t! i])$   
**have**  $trace\ init\ (take\ k\ t)\ (S\ t\ k)$   
**using**  $0.premis(6)\ exists\ trace\ for\ any\ i\ \mathbf{by}\ blast$   
**moreover** **have**  $trace\ init\ (take\ k\ ?newt)\ (S\ ?newt\ k)$   
**using**  $\langle take\ i\ t\ @\ [t! j]\ @\ [t! i]\ @\ drop\ (j + 1)\ t = t[i := t! j,\ j := t! i] \rangle$   
 $\langle trace\ init\ (take\ i\ t\ @\ [t! j]\ @\ [t! i]\ @\ drop\ (j + 1)\ t)\ final \rangle\ exists\ trace\ for\ any\ i$   
**by**  $auto$   
**moreover** **have**  $take\ k\ t = take\ k\ ?newt$   
**using**  $0.premis(1)\ \langle k \leq i \rangle\ \mathbf{by}\ auto$   
**ultimately** **show**  $S\ t\ k = S\ ?newt\ k$   
**by**  $(simp\ add:\ trace\ and\ start\ determines\ end)$   
**qed**  
**moreover** **have**  $prerecording\ event\ (swap\ events\ i\ j\ t)\ i$   
**proof**  $-$

**have**  $\sim$  *has-snapshotted* ( $S ((t[i := t! j])[j := t! i]) i$ )  $?q$   
**by** (*metis*  $0.premis(6)$   $\langle j = i + 1 \rangle$  *add.right-neutral calculation(4) le-add1 nsq snapshot-stable-ver-3*)  
**moreover have** *regular-event* ( $((t[i := t! j])[j := t! i]) ! i$ )  
**by** (*metis*  $0.premis(4)$   $0.premis(5)$   $\langle \text{occurs-on } (t! i) \neq \text{occurs-on } (t! j) \rangle$  *nth-list-update-eq nth-list-update-neq postrecording-event prerecording-event*)  
**moreover have**  $i < \text{length } ((t[i := t! j])[j := t! i])$   
**using**  $0.premis(1)$   $0.premis(2)$  **by** *auto*  
**ultimately show** *?thesis unfolding prerecording-event*  
**by** (*metis* (*no-types, opaque-lifting*)  $0.premis(1)$   $\langle \text{take } (j - (i + 1)) (\text{drop } (i + 1) t) = [] \rangle$   $\langle \text{take } i t @ [t! j] @ [t! i] @ \text{drop } (j + 1) t = t[i := t! j, j := t! i] \rangle$  *append-Cons length-list-update nat-less-le nth-list-update-eq nth-list-update-neq self-append-conv2*)  
**qed**  
**moreover have** *postrecording-event* (*swap-events*  $i j t$ )  $(i+1)$   
**proof** –  
**have** *has-snapshotted* ( $S ((t[i := t! j])[j := t! i]) (i+1)$ )  $?p$   
**by** (*metis*  $0.premis(4)$  *add.right-neutral calculation(1) calculation(2) calculation(4) le-add1 postrecording-event snapshot-stable-ver-3*)  
**moreover have** *regular-event* ( $((t[i := t! j])[j := t! i]) ! j$ )  
**using**  $0.premis(2)$   $0.premis(4)$  *length-list-update postrecording-event* **by** *auto*  
**moreover have**  $j < \text{length } t$  **using**  $0.premis$  **by** *auto*  
**ultimately show** *?thesis unfolding postrecording-event*  
**by** (*metis*  $\langle j = i + 1 \rangle$  *length-list-update nth-list-update-eq swap-neighbors-2*)  
**qed**  
**moreover have**  $\forall k. k > i+1 \wedge k < j+1 \longrightarrow \sim$  *regular-event* ( $(\text{swap-events } i j t) ! k$ ) **using**  $0$  **by** *force*  
**ultimately show** *?case using*  $\langle j = i + 1 \rangle$  **by** *force*  
**next**  
**case** (*Suc*  $n$ )  
**let**  $?p = \text{occurs-on } (t! i)$   
**let**  $?q = \text{occurs-on } (t! j)$   
**let**  $?t = \text{take } ((j+1) - i) (\text{drop } i t)$   
**let**  $?subt = \text{take } (j - (i+1)) (\text{drop } (i+1) t)$   
**let**  $?subt' = \text{take } ((j-1) - (i+1)) (\text{drop } (i+1) t)$   
**have**  $sp$ : *has-snapshotted* ( $S t i$ )  $?p$   
**using**  $Suc.premis$  *postrecording-event prerecording-event* **by** *blast*  
**have**  $nsq$ :  $\sim$  *has-snapshotted* ( $S t j$ )  $?q$   
**using**  $Suc.premis$  *postrecording-event prerecording-event* **by** *blast*  
**have**  $?p \neq ?q$   
**using**  $Suc.premis$  *computation.post-before-pre-different-processes computation-axioms*  
**by** *blast*  
**have**  $?subt \neq Nil$   
**using**  $Suc.hyps(2)$   $Suc.premis(1)$   $Suc.premis(2)$  **by** *auto*  
**have**  $?subt' = \text{butlast } ?subt$   
**by** (*metis*  $Suc.premis(2)$  *Suc-eq-plus1 butlast-drop butlast-take drop-take less-imp-le-nat*)  
**have**  $i < \text{length } t$   
**using**  $\langle \text{postrecording-event } t i \rangle$  *postrecording-event*  $[\text{of } t i]$  **by** *simp*  
**then have** *step*:  $\langle S t i \vdash t! i \mapsto S t (Suc i) \rangle$

```

    using ‹trace init t final› by (rule step-Suc)
  have ?t = t ! i # ?subt @ [t ! j]
proof -
  have f1: Suc j - i = Suc (j - i)
    using Suc.prem1 Suc-diff-le le-simps(1) by presburger
  have f2: t ! i # drop (Suc i) t = drop i t
    by (meson Cons-nth-drop-Suc Suc.prem1 Suc.prem2 less-trans)
  have f3: t ! j # drop (Suc j) t = drop j t
    using Cons-nth-drop-Suc Suc.prem2 by blast
  have f4: j - (i + 1) + (i + 1) = j
    using Suc.prem1 by force
  have j - (i + 1) + Suc 0 = j - i
    using Suc.prem1 Suc-diff-Suc by presburger
  then show ?thesis
    using f4 f3 f2 f1 by (metis One-nat-def Suc.hyps(2) Suc-eq-plus1 drop-drop
take-Suc-Cons take-add take-eq-Nil)
qed
  then have trace (S t i) ?t (S t (j+1))
    by (metis Suc.prem1 Suc.prem6 Suc-eq-plus1 exists-trace-for-any-i-j less-SucI
nat-less-le)
  then have reg-tr-1: trace (S t i) (t ! i # ?subt) (S t j)
    using ‹i < j› ‹i < length t› ‹trace init t final› step
    by simp (meson exists-trace-for-any-i-j less-eq-Suc-le trace.simps)
  have reg-st-2: (S t j) ⊢ (t ! j) ↔ (S t (j+1))
    using Suc.prem2 Suc.prem6 step-Suc by auto
  have ?subt = ?subt' @ [t ! (j-1)]
proof -
  have f1: ∀ n es. ¬ n < length es ∨ take n es @ [hd (drop n es)::('a, 'b, 'c) event]
= take (Suc n) es
    by (meson take-hd-drop)
  have f2: j - 1 - (i + 1) = n
    by (metis (no-types) Suc.hyps(2) Suc-eq-plus1 diff-Suc-1 diff-diff-left plus-1-eq-Suc)

  have f3: ∀ n na. ¬ n < na ∨ Suc n ≤ na
    using Suc-leI by blast
  then have f4: Suc i ≤ j - 1
    by (metis (no-types) Suc.hyps(2) Suc-eq-plus1 diff-diff-left plus-1-eq-Suc
zero-less-Suc zero-less-diff)
  have f5: i + 1 < j
    by (metis Suc.hyps(2) zero-less-Suc zero-less-diff)
  then have f6: t ! (j - 1) = hd (drop n (drop (i + 1) t))
    using f4 f3 by (metis (no-types) Suc.hyps(2) Suc.prem2 Suc-eq-plus1
Suc-lessD add-Suc-right diff-Suc-1 drop-drop hd-drop-conv-nth le-add-diff-inverse2
plus-1-eq-Suc)
  have n < length (drop (i + 1) t)
    using f5 f3 by (metis (no-types) Suc.hyps(2) Suc.prem2 Suc-eq-plus1
Suc-lessD drop-drop le-add-diff-inverse2 length-drop zero-less-diff)
  then show ?thesis
    using f6 f2 f1 Suc.hyps(2) by presburger

```

**qed**  
**then have** *reg-tr*:  $\text{trace } (S \ t \ i) \ (t ! \ i \ \# \ ?\text{subt}') \ (S \ t \ (j-1))$   
**proof** –  
**have** *f1*:  $j - \text{Suc } i = \text{Suc } n$   
**using** *Suc.hyps(2)* **by** *presburger*  
**have** *f2*:  $\text{length } (\text{take } j \ t) = j$   
**by** (*metis* (*no-types*) *Suc.prem(2)* *length-take min.absorb2 nat-le-linear not-less*)  
**have** *f3*:  $(t ! \ i \ \# \ \text{drop } (\text{Suc } i) \ (\text{take } j \ t)) \ @ \ [t ! \ j] = \text{drop } i \ (\text{take } (\text{Suc } j) \ t)$   
**by** (*metis* (*no-types*) *Suc-eq-plus1*  $\langle \text{take } (j + 1 - i) \ (\text{drop } i \ t) = t ! \ i \ \# \ \text{take } (j - (i + 1)) \ (\text{drop } (i + 1) \ t) \ @ \ [t ! \ j] \rangle$  *append-Cons drop-take*)  
**have** *f4*:  $\text{Suc } (i + n) = j - 1$   
**using** *f1* **by** (*metis* (*no-types*) *Suc.prem(1)* *Suc-diff-Suc add-Suc-right diff-Suc-1 le-add-diff-inverse nat-le-linear not-less*)  
**have**  $\text{Suc } (j - 1) = j$   
**using** *f1* **by** *simp*  
**then have** *f5*:  $\text{butlast } (\text{take } (\text{Suc } j) \ t) = \text{take } j \ t$   
**using** *f4 f3 f2 f1* **by** (*metis* (*no-types*) *Groups.add-ac(2)* *One-nat-def append-eq-conv-conj append-take-drop-id butlast-take diff-Suc-1 drop-drop length-append length-drop list.size(3) list.size(4) order-refl plus-1-eq-Suc plus-nat.simps(2) take-add take-all*)  
**have** *f6*:  $\text{butlast } (\text{take } j \ t) = \text{take } (j - 1) \ t$   
**by** (*meson* *Suc.prem(2)* *butlast-take nat-le-linear not-less*)  
**have**  $\text{drop } (\text{Suc } i) \ (\text{take } j \ t) \neq []$   
**by** (*metis* (*no-types*) *Nil-is-append-conv Suc-eq-plus1*  $\langle \text{take } (j - (i + 1)) \ (\text{drop } (i + 1) \ t) = \text{take } (j - 1 - (i + 1)) \ (\text{drop } (i + 1) \ t) \ @ \ [t ! \ (j - 1)] \rangle$  *drop-take list.distinct(1)*)  
**then show** *?thesis*  
**using** *f6 f5 f4 f3* **by** (*metis* (*no-types*) *Suc.prem(6)* *Suc-eq-plus1 butlast.simps(2) butlast-drop butlast-snoc drop-take exists-trace-for-any-i-j less-add-Suc1 nat-le-linear not-less*)  
**qed**  
  
**have** *reg-st-1*:  $(S \ t \ (j-1)) \vdash (t ! \ (j-1)) \mapsto (S \ t \ j)$   
**by** (*metis* *Suc.prem(1)* *Suc.prem(2)* *Suc.prem(6)* *Suc-lessD diff-Suc-1 less-imp-Suc-add step-Suc*)  
**have**  $\sim \text{regular-event } (t ! \ (j-1))$   
**using** *Suc.prem(3)*  $\langle \text{take } (j - (i + 1)) \ (\text{drop } (i + 1) \ t) \neq [] \rangle$  *less-diff-conv*  
**by** *auto*  
**moreover have** *regular-event*  $(t ! \ j)$   
**using** *Suc.prem(5)* *computation.prerecording-event computation-axioms* **by** *blast*  
**moreover have** *can-occur*  $(t ! \ j) \ (S \ t \ j)$   
**using** *happen-implies-can-occur reg-tr-1 reg-st-2* **by** *blast*  
**moreover have** *njmiq*:  $\text{occurs-on } (t ! \ (j-1)) \neq ?q$   
**proof** (*rule ccontr*)  
**assume**  $\sim \text{occurs-on } (t ! \ (j-1)) \neq ?q$   
**then have**  $\text{occurs-on } (t ! \ (j-1)) = ?q$  **by** *simp*  
**then have** *has-snapshotted*  $(S \ t \ j) \ ?q$

```

    using Suc.premis(6) calculation(1) diff-le-self nonregular-event-induces-snapshot
reg-st-1 snapshot-stable-ver-2 by blast
    then show False using nsq by simp
qed
ultimately have can-occur (t ! j) (S t (j-1))
    using reg-tr reg-st-1 event-can-go-back-if-no-sender by auto
then obtain d where new-st-1: (S t (j-1)) ⊢ (t ! j) ↦ d
    using exists-next-if-can-occur by blast
then have trace (S t i) (t ! i # ?subt' @ [t ! j]) d using reg-tr trace-snoc by
fastforce
moreover have can-occur (t ! (j-1)) d
    using ⟨(S t (j-1)) ⊢ t ! j ↦ d⟩ ⟨occurs-on (t ! (j - 1)) ≠ occurs-on (t ! j)⟩
event-stays-valid-if-no-occurrence happen-implies-can-occur reg-st-1 by auto
moreover obtain e where new-st-2: d ⊢ (t ! (j-1)) ↦ e
    using calculation(2) exists-next-if-can-occur by blast

have pre-swap: e = (S t (j+1))
proof -
    have states e = states (S t (j+1))
    proof (rule ext)
        fix p
        have states (S t (j-1)) p = states (S t j) p
        using no-state-change-if-nonregular-event ⟨~ regular-event (t ! (j-1))⟩ reg-st-1
by auto
        moreover have states d p = states e p
            using no-state-change-if-nonregular-event ⟨~ regular-event (t ! (j-1))⟩
new-st-2 by auto
        moreover have states d p = states (S t (j+1)) p
        proof -
            have ∀ a. states (S t (j + 1)) a = states d a
            by (meson ⟨~ regular-event (t ! (j - 1))⟩ new-st-1 no-state-change-if-nonregular-event
reg-st-1 reg-st-2 same-state-implies-same-result-state)
            then show ?thesis
            by presburger
        qed
    ultimately show states e p = states (S t (j+1)) p by simp
qed

moreover have msgs e = msgs (S t (j+1))
proof (rule ext)
    fix cid
    have isTrans (t ! j) ∨ isSend (t ! j) ∨ isRecv (t ! j)
        using ⟨regular-event (t ! j)⟩ by auto
    moreover have isSnapshot (t ! (j-1)) ∨ isRecvMarker (t ! (j-1))
        using nonregular-event ⟨~ regular-event (t ! (j-1))⟩ by auto
    ultimately show msgs e cid = msgs (S t (j+1)) cid
    proof (elim disjE, goal-cases)
        case 1
        then show ?case

```

```

        using new-st-1 new-st-2 njmiq reg-st-1 reg-st-2 swap-Trans-Snapshot by
    auto
    next
    case 2
    then show ?case
    using new-st-1 new-st-2 njmiq reg-st-1 reg-st-2 swap-msgs-Trans-RecvMarker
by auto
    next
    case 3
    then show ?case
    using new-st-1 new-st-2 njmiq reg-st-1 reg-st-2 swap-Send-Snapshot by
    auto
    next
    case 4
    then show ?case
    using new-st-1 new-st-2 njmiq reg-st-1 reg-st-2 swap-Recv-Snapshot by
    auto
    next
    case 5
    then show ?case
    using new-st-1 new-st-2 njmiq reg-st-1 reg-st-2 swap-msgs-Send-RecvMarker
by auto
    next
    case 6
    then show ?case
    using new-st-1 new-st-2 njmiq reg-st-1 reg-st-2 swap-msgs-Recv-RecvMarker
by auto
    qed
    qed

    moreover have process-snapshot e = process-snapshot (S t (j+1))
    proof (rule ext)
    fix p
    have process-snapshot (S t j) p = process-snapshot (S t (j+1)) p
    using ‹regular-event (t ! j)› reg-st-2 regular-event-preserves-process-snapshots
by blast
    moreover have process-snapshot (S t (j-1)) p = process-snapshot d p
    using ‹regular-event (t ! j)› new-st-1 regular-event-preserves-process-snapshots
by blast
    moreover have process-snapshot e p = process-snapshot (S t j) p
    proof -
    have occurs-on (t ! j) = p  $\longrightarrow$  ps e p = ps (S t j) p
    using calculation(2) new-st-2 njmiq no-state-change-if-no-event reg-st-1
by force
    then show ?thesis
    by (meson new-st-1 new-st-2 no-state-change-if-no-event reg-st-1 same-snapshot-state-implies-same-resul
    qed
    ultimately show process-snapshot e p = process-snapshot (S t (j+1)) p by
    simp

```

```

qed

moreover have cs e = cs (S t (j+1))
proof (rule ext)
  fix cid
  have isTrans (t ! j) ∨ isSend (t ! j) ∨ isRecv (t ! j)
    using ⟨regular-event (t ! j)⟩ by auto
  moreover have isSnapshot (t ! (j-1)) ∨ isRecvMarker (t ! (j-1))
    using nonregular-event ⟨ $\sim$  regular-event (t ! (j-1))⟩ by auto
  ultimately show cs e cid = cs (S t (j+1)) cid
  proof (elim disjE, goal-cases)
    case 1
    then show ?case
      using new-st-1 new-st-2 reg-st-1 reg-st-2 swap-cs-Trans-Snapshot by auto
    next
    case 2
    then show ?case
      using new-st-1 new-st-2 reg-st-1 reg-st-2 swap-cs-Trans-RecvMarker by
  auto
    next
    case 3
    then show ?case
      using new-st-1 new-st-2 reg-st-1 reg-st-2 swap-cs-Send-Snapshot by auto
    next
    case 4
    then show ?case
      using new-st-1 new-st-2 reg-st-1 reg-st-2 swap-cs-Recv-Snapshot njmiq by
  auto
    next
    case 5
    then show ?case
      using new-st-1 new-st-2 reg-st-1 reg-st-2 swap-cs-Send-RecvMarker by auto
    next
    case 6
    then show ?case
      using new-st-1 new-st-2 reg-st-1 reg-st-2 swap-cs-Recv-RecvMarker njmiq
  by auto
  qed
  qed
  ultimately show ?thesis by auto
qed

let ?it = (t[j-1 := t ! j])[j := t ! (j-1)]
have same-prefix: take (j-1) ?it = take (j-1) t by simp
have same-suffix: drop (j+1) ?it = drop (j+1) t by simp
have trace-prefix: trace init (take (j-1) ?it) (S t (j-1))
  using Suc.premis(6) exists-trace-for-any-i by auto
have ?it = take (j-1) t @ [t ! j, t ! (j-1)] @ drop (j+1) t
proof -

```

**have**  $1 < j$   
**by** (*metis* (*no-types*) *Suc.hyps*(2) *Suc-eq-plus1* *add-lessD1* *plus-1-eq-Suc* *zero-less-Suc* *zero-less-diff*)  
**then have**  $j - 1 + 1 = j$   
**by** (*metis* (*no-types*) *le-add-diff-inverse2* *nat-less-le*)  
**then show** *?thesis*  
**by** (*metis* (*no-types*) *Suc.prem*(2) *Suc-eq-plus1* *add-Suc-right* *one-add-one* *swap-neighbors*)  
**qed**  
**have** *trace* ( $S\ t\ (j-1)$ ) [ $t!\ j, t!\ (j-1)$ ] ( $S\ t\ (j+1)$ )  
**by** (*metis* *new-st-1* *new-st-2* *pre-swap* *trace.simps*)  
**have** *trace* *init* ( $\text{take}\ (j+1)\ t\ @\ \text{drop}\ (j+1)\ t$ ) *final*  
**by** (*simp* *add: Suc.prem*(6))  
**then have** *trace* *init* ( $\text{take}\ (j+1)\ t$ ) ( $S\ t\ (j+1)$ )  $\wedge$  *trace* ( $S\ t\ (j+1)$ ) ( $\text{drop}\ (j+1)\ t$ ) *final*  
**using** *Suc.prem*(6) *exists-trace-for-any-i* *split-trace* *trace-and-start-determines-end*  
**by** *blast*  
**then have** *trace-suffix: trace* ( $S\ t\ (j+1)$ ) ( $\text{drop}\ (j+1)\ ?it$ ) *final* **using** *same-suffix*  
**by** *simp*  
**have** *trace* *init* *?it* *final*  
**by** (*metis* (*no-types*, *lifting*)  $\langle t[j - 1 := t!\ j, j := t!\ (j - 1)] = \text{take}\ (j - 1)\ t\ @\ [t!\ j, t!\ (j - 1)]\ @\ \text{drop}\ (j + 1)\ t \rangle$   $\langle \text{trace}\ (S\ t\ (j + 1))\ (\text{drop}\ (j + 1)\ (t[j - 1 := t!\ j, j := t!\ (j - 1)]))\ \text{final} \rangle$   $\langle \text{trace}\ (S\ t\ (j - 1))\ [t!\ j, t!\ (j - 1)]\ (S\ t\ (j + 1)) \rangle$   $\langle \text{trace}\ \text{init}\ (\text{take}\ (j - 1)\ (t[j - 1 := t!\ j, j := t!\ (j - 1)]))\ (S\ t\ (j - 1)) \rangle$  *same-prefix* *same-suffix* *trace-trans*)  
**have** *suffix-same-states:  $\forall k. k > j \longrightarrow S\ t\ k = S\ ?it\ k$*   
**proof** (*rule* *allI*, *rule* *impI*)  
**fix**  $k$   
**assume**  $k > j$   
**have** *eq-trace: drop* ( $j+1$ )  $t = \text{drop}\ (j+1)\ ?it$  **by** *simp*  
**have** *trace* *init* ( $\text{take}\ (j+1)\ ?it$ ) ( $S\ ?it\ (j+1)$ )  
**using**  $\langle \text{trace}\ \text{init}\ (t[j - 1 := t!\ j, j := t!\ (j - 1)])\ \text{final} \rangle$  *exists-trace-for-any-i*  
**by** *blast*  
**moreover have** *trace* *init* ( $\text{take}\ (j+1)\ ?it$ ) ( $S\ t\ (j+1)$ )  
**proof** –  
**have** *f1:  $\forall es\ esa\ esb\ esc. (esb::('a, 'b, 'c)\ \text{event}\ \text{list})\ @\ es \neq esa\ @\ esc\ @\ es \vee esa\ @\ esc = esb$*   
**by** *auto*  
**have** *f2: take* ( $j + 1$ ) ( $t[j - 1 := t!\ j, j := t!\ (j - 1)]$ )  $@\ \text{drop}\ (j + 1)\ t = t[j - 1 := t!\ j, j := t!\ (j - 1)]$   
**by** (*metis* *append-take-drop-id* *same-suffix*)  
**have** *trace* *init* ( $\text{take}\ (j - 1)\ t\ @\ [t!\ j, t!\ (j - 1)]$ ) ( $S\ t\ (j + 1)$ )  
**using**  $\langle \text{trace}\ (S\ t\ (j - 1))\ [t!\ j, t!\ (j - 1)]\ (S\ t\ (j + 1)) \rangle$  *same-prefix* *trace-prefix* *trace-trans* **by** *presburger*  
**then show** *?thesis*  
**using** *f2* *f1* **by** (*metis* (*no-types*)  $\langle t[j - 1 := t!\ j, j := t!\ (j - 1)] = \text{take}\ (j - 1)\ t\ @\ [t!\ j, t!\ (j - 1)]\ @\ \text{drop}\ (j + 1)\ t \rangle$ )  
**qed**  
**ultimately have** *eq-start: S* *?it* ( $j+1$ ) =  $S\ t\ (j+1)$

```

    using trace-and-start-determines-end by blast
  then have take k ?it = take (j+1) ?it @ take (k - (j+1)) (drop (j+1) ?it)
    by (metis Suc-eq-plus1 Suc-leI <j < k> le-add-diff-inverse take-add)
  have trace (S ?it (j+1)) (take (k - (j+1)) (drop (j+1) ?it)) (S ?it k)
    by (metis Suc-eq-plus1 Suc-leI <j < k> <trace init (t[j - 1 := t ! j, j := t ! (j
- 1)]) final> exists-trace-for-any-i-j)
  moreover have trace (S t (j+1)) (take (k - (j+1)) (drop (j+1) t)) (S t k)
    using Suc.prem6(6) <j < k> exists-trace-for-any-i-j by fastforce
  ultimately show S t k = S ?it k
    using eq-start trace-and-start-determines-end by auto
qed
have prefix-same-states:  $\forall k. k < j \longrightarrow S t k = S ?it k$ 
proof (rule allI, rule impI)
  fix k
  assume k < j
  have trace init (take k t) (S t k)
    using Suc.prem6(6) exists-trace-for-any-i by blast
  moreover have trace init (take k ?it) (S ?it k)
    by (meson <trace init (t[j - 1 := t ! j, j := t ! (j - 1)]) final> ex-
ists-trace-for-any-i)
  ultimately show S t k = S ?it k
    using <k < j> s-def by auto
qed
moreover have j - 1 < length ?it
  using Suc.prem2(2) by auto
moreover have prerecording-event ?it (j-1)
proof -
  have f1:  $t[j - 1 := t ! j, j := t ! (j - 1)] ! (j - 1) = t[j - 1 := t ! j] ! (j - 1)$ 
  by (metis (no-types) njmiq nth-list-update-neq)
  have j  $\neq$  0
  by (metis (no-types) Suc.prem1 not-less-zero)
  then have  $\neg j < 1$ 
  by blast
  then have S t (j - 1) = S (t[j - 1 := t ! j, j := t ! (j - 1)]) (j - 1)
  by (simp add: prefix-same-states)
  then show ?thesis
  using f1 by (metis <regular-event (t ! j)> calculation(4) computation.prerecording-event
computation-axioms length-list-update njmiq no-state-change-if-no-event nsq nth-list-update-eq
reg-st-1)
qed
moreover have postrecording-event ?it i
proof -
  have i < length ?it
  using Suc.prem4(4) postrecording-event by auto
  then show ?thesis
proof -
  assume i < length (t[j - 1 := t ! j, j := t ! (j - 1)])
  have i < j - 1

```

```

    by (metis (no-types) Suc.hyps(2) cancel-ab-semigroup-add-class.diff-right-commute
diff-diff-left zero-less-Suc zero-less-diff)
    then show ?thesis
      using Suc.premis(1) Suc.premis(4) postrecording-event prefix-same-states by
auto
    qed
  qed
  moreover have  $i < j - 1$ 
    using Suc.hyps(2) by auto
  moreover have  $\forall k. i < k \wedge k < (j-1) \longrightarrow \sim \text{regular-event } (?it ! k)$ 
  proof (rule allI, rule impI)
    fix k
    assume  $i < k \wedge k < (j-1)$ 
    show  $\sim \text{regular-event } (?it ! k)$ 
      using Suc.premis(3)  $\langle i < k \wedge k < j - 1 \rangle$  by force
  qed
  moreover have  $(j-1) - (i+1) = n$  using Suc.premis Suc.hyps by auto
  ultimately have ind: trace init (swap-events i (j-1) ?it) final
     $\wedge (\forall k. k \geq (j-1)+1 \longrightarrow S \text{ (swap-events } i \text{ (j-1) ?it) } k = S$ 
? $it \ k)$ 
     $\wedge (\forall k. k \leq i \longrightarrow S \text{ (swap-events } i \text{ (j-1) ?it) } k = S \text{ ?it } k)$ 
     $\wedge \text{prerecording-event (swap-events } i \text{ (j-1) ?it) } i$ 
     $\wedge \text{postrecording-event (swap-events } i \text{ (j-1) ?it) } (i+1)$ 
     $\wedge (\forall k. k > i+1 \wedge k < (j-1)+1 \longrightarrow \sim \text{regular-event } ((\text{swap-events}$ 
 $i \text{ (j-1) ?it) ! } k))$ 
    using Suc.hyps  $\langle \text{trace init ?it final} \rangle$  by blast
  then have new-trace: trace init (swap-events i (j-1) ?it) final by blast
  have equal-suffix-states:  $\forall k. k \geq j \longrightarrow S \text{ (swap-events } i \text{ (j-1) ?it) } k = S \text{ ?it } k$ 
    using Suc.premis(1) ind by simp
  have equal-prefix-states:  $\forall k. k \leq i \longrightarrow S \text{ (swap-events } i \text{ (j-1) ?it) } k = S \text{ ?it } k$ 
    using ind by blast
  have neighboring-events-shifted:  $\forall k. k > i+1 \wedge k < j \longrightarrow \sim \text{regular-event}$ 
 $((\text{swap-events } i \text{ (j-1) ?it) ! } k)$ 
    using ind by force

  let ?itn = swap-events i (j-1) ?it
  have ?itn = swap-events i j t
  proof -
    have f1:  $i \leq j - 1$ 
      using  $\langle i < j - 1 \rangle$  less-imp-le-nat by blast
    have  $t ! j \# [t ! (j - 1)] @ \text{drop } (j + 1) t = \text{drop } (j - 1) (\text{take } (j - 1) t @$ 
 $[t ! j, t ! (j - 1)] @ \text{drop } (j + 1) t)$ 
      using  $\langle t[j - 1 := t ! j, j := t ! (j - 1)] = \text{take } (j - 1) t @ [t ! j, t ! (j -$ 
 $1)] @ \text{drop } (j + 1) t \rangle$  same-prefix by force
    then have f2:  $t[j - 1 := t ! j, j := t ! (j - 1)] ! (j - 1) = t ! j \wedge \text{drop } (j -$ 
 $1 + 1) (t[j - 1 := t ! j, j := t ! (j - 1)]) = t ! (j - 1) \# [] @ \text{drop } (j + 1) t$ 
      by (metis (no-types) Cons-nth-drop-Suc Suc-eq-plus1  $\langle j - 1 < \text{length } (t[j -$ 
 $1 := t ! j, j := t ! (j - 1)] \rangle \langle t[j - 1 := t ! j, j := t ! (j - 1)] = \text{take } (j - 1) t$ 
 $@ [t ! j, t ! (j - 1)] @ \text{drop } (j + 1) t \rangle$  append-Cons list.inject)

```

```

have  $t ! i = t[j - 1 := t ! j, j := t ! (j - 1)] ! i$ 
by (metis (no-types) Suc.prems(1)  $\langle i < j - 1 \rangle$  nat-neq-iff-nth-list-update-neq)
then show ?thesis
using f2 f1 by (metis (no-types) Suc.prems(1)  $\langle \text{take } (j - (i + 1)) (\text{drop } (i + 1) t) = \text{take } (j - 1 - (i + 1)) (\text{drop } (i + 1) t) @ [t ! (j - 1)] \rangle$  append.assoc
append-Cons drop-take-less-imp-le-nat same-prefix-take-update-cancel)
qed

moreover have  $\forall k. k \leq i \longrightarrow S t k = S ?itn k$ 
using Suc.prems(1) equal-prefix-states prefix-same-states by auto
moreover have  $\forall k. k \geq j + 1 \longrightarrow S t k = S ?itn k$ 
by (metis (no-types, lifting) Suc-eq-plus1 add-lessD1 equal-suffix-states lessI
nat-less-le suffix-same-states)
moreover have  $\forall k. k > i + 1 \wedge k < j + 1 \longrightarrow \sim \text{regular-event } (?itn ! k)$ 
proof -
have  $\sim \text{regular-event } (?itn ! j)$ 
proof -
have f1:  $j - 1 < \text{length } t$ 
using  $\langle j - 1 < \text{length } (t[j - 1 := t ! j, j := t ! (j - 1)]) \rangle$  by force
have f2:  $\bigwedge n \text{ na es. } \neg n < \text{na} \vee \neg \text{na} < \text{length es} \vee \text{drop } (S \text{uc } \text{na}) (\text{take } n \text{ es}) @ [\text{hd } (\text{drop } \text{na } \text{es}), \text{es} ! n :: ('a, 'b, 'c) \text{event}] @ \text{take } (\text{na} - S \text{uc } n) (\text{drop } (S \text{uc } n) \text{es}) @ \text{drop } (S \text{uc } \text{na}) \text{es} = \text{drop } (S \text{uc } \text{na}) \text{es}$ 
by (metis Suc-eq-plus1 hd-drop-conv-nth swap-identical-tails)
have f3:  $t ! j = \text{hd } (\text{drop } j t)$ 
by (simp add: Suc.prems(2) hd-drop-conv-nth)
have  $\neg j < 1$ 
using Suc.prems(1) by blast
then have  $\neg \text{regular-event } (\text{hd } (\text{drop } j (\text{take } i (t[j - 1 := \text{hd } (\text{drop } j t), j := \text{hd } (\text{drop } (j - 1) t)])) @ [\text{hd } (\text{drop } (j - 1) (t[j - 1 := \text{hd } (\text{drop } j t), j := \text{hd } (\text{drop } (j - 1) t)]))], t[j - 1 := \text{hd } (\text{drop } j t), j := \text{hd } (\text{drop } (j - 1) t)] ! i] @ \text{take } (j - 1 - S \text{uc } i) (\text{drop } (S \text{uc } i) (t[j - 1 := \text{hd } (\text{drop } j t), j := \text{hd } (\text{drop } (j - 1) t)])) @ \text{drop } (S \text{uc } (j - 1)) (t[j - 1 := \text{hd } (\text{drop } j t), j := \text{hd } (\text{drop } (j - 1) t)]))$ 
using f2 f1 by (metis (no-types) Suc.prems(2)  $\langle \neg \text{regular-event } (t ! (j - 1)) \rangle$ 
 $\langle i < j - 1 \rangle$  add-diff-inverse-nat hd-drop-conv-nth length-list-update nth-list-update-eq
plus-1-eq-Suc)
then show ?thesis
using f3 f1 by (metis Suc.prems(2) Suc-eq-plus1  $\langle i < j - 1 \rangle$  hd-drop-conv-nth
length-list-update swap-identical-length)
qed
then show ?thesis
by (metis Suc-eq-plus1 less-Suc-eq neighboring-events-shifted)
qed

ultimately show ?case using ind by presburger
qed

```

### 5.3.3 Relating configurations and the computed snapshot

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

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

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

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

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

*state-equal-to-snapshot*  $c\ c' \equiv$   
 $\text{ps-equal-to-snapshot } c\ c' \wedge \text{cs-equal-to-snapshot } c\ c'$

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

**assumes**

*trace init t final*

**shows**

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

**by** (*metis* *assms exists-trace-for-any-i take-eq-Nil tr-init trace-and-start-determines-end*)

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

**assumes**

*trace init t final*

**shows**

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

**by** (*metis* *assms exists-trace-for-any-i order-refl take-all trace-and-start-determines-end*)

**lemma** *snapshot-event*:

**assumes**

*trace init t final and*

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

*has-snapshotted*  $(S\ t\ (i+1))\ p$

**shows**

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

**proof** –

**have**  $(S\ t\ i) \vdash (t\ !\ i) \mapsto (S\ t\ (i+1))$

**by** (*metis* *Suc-eq-plus1 assms(1) assms(2) assms(3) distributed-system.step-Suc computation-axioms computation-def nat-less-le not-less not-less-eq s-def take-all*)

**then show** *?thesis*

**using** *assms(2) assms(3) nonregular-event regular-event-cannot-induce-snapshot*

**by** *blast*

**qed**

**lemma** *snapshot-state*:

**assumes**

*trace init t final and*

*states*  $(S\ t\ i)\ p = u$  **and**

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

*has-snapshotted*  $(S\ t\ (i+1))\ p$

```

shows
  ps (S t (i+1)) p = Some u
proof -
  have step: (S t i) ⊢ (t ! i) ⇔ (S t (i+1))
    by (metis add commute assms(1) assms(3) assms(4) le-SucI le-eq-less-or-eq
le-refl nat-neq-iff no-change-if-ge-length-t plus-1-eq-Suc step-Suc)
  let ?q = occurs-on (t ! i)
  have qp: ?q = p
  proof (rule ccontr)
    assume ?q ≠ p
    then have has-snapshotted (S t (i+1)) p = has-snapshotted (S t i) p
      using local.step no-state-change-if-no-event by auto
    then show False using assms by simp
  qed
  have isSnapshot (t ! i) ∨ isRecvMarker (t ! i) using assms snapshot-event by
auto
  then show ?thesis
  proof (elim disjE, goal-cases)
    case 1
    then have t ! i = Snapshot p
      by (metis event.collapse(4) qp)
    then show ?thesis
      using assms(2) local.step by auto
  next
    case 2
    then obtain cid' q where t ! i = RecvMarker cid' p q
      by (metis event.collapse(5) qp)
    then show ?thesis using assms step by auto
  qed
qed

lemma snapshot-state-unchanged-trace-2:
shows
  ⌊ trace init t final; i ≤ j; j ≤ length t;
  ps (S t i) p = Some u
  ⌋ ⇒ ps (S t j) p = Some u
proof (induct i j rule:S-induct)
  case S-init
  then show ?case by simp
next
  case S-step
  then show ?case using snapshot-state-unchanged by auto
qed

lemma no-recording-cs-if-not-snapshotted:
shows
  ⌊ trace init t final; ~ has-snapshotted (S t i) p;
  channel cid = Some (q, p) ⌋ ⇒ cs (S t i) cid = cs init cid
proof (induct i)

```

```

case 0
then show ?case
  by (metis exists-trace-for-any-i list.discI take-eq-Nil trace.simps)
next
case (Suc i)
have Suc i < length t
proof -
  have has-snapshotted final p
  using all-processes-snapshotted-in-final-state valid by blast
show ?thesis
proof (rule ccontr)
  assume ~ Suc i < length t
  then have Suc i ≥ length t by simp
  then have has-snapshotted (S t (Suc i)) p
  using Suc.prem(1) ⟨ps final p ≠ None⟩ final-is-s-t-len-t snapshot-stable-ver-3
by blast
  then show False using Suc by simp
qed
qed

then have t-dec: trace init (take i t) (S t i) ∧ (S t i) ⊢ (t ! i) ↔ (S t (Suc i))
  using Suc.prem(1) exists-trace-for-any-i step-Suc by auto
moreover have step: (S t i) ⊢ (t ! i) ↔ (S t (Suc i)) using calculation by simp

ultimately have IH: cs (S t i) cid = cs init cid
  using Suc.hyps Suc.prem(1) Suc.prem(2) Suc.prem(3) snapshot-state-unchanged
by fastforce

then show ?case
proof (cases t ! i)
case (Snapshot r)
  have r ≠ p
  proof (rule ccontr)
  assume ~ r ≠ p
  then have r = p by simp
  then have has-snapshotted (S t (Suc i)) p
  using Snapshot step by auto
  then show False using Suc by simp
qed
then have cs (S t i) cid = cs (S t (Suc i)) cid
  using Snapshot Suc.prem(3) local.step by auto
then show ?thesis using IH by simp
next
case (RecvMarker cid' r s)
  have r ≠ p
  proof (rule ccontr)
  assume ~ r ≠ p
  then have r = p by simp
  then have has-snapshotted (S t (Suc i)) p

```

```

    using RecvMarker t-dec recv-marker-means-snapshotted-1 by blast
  then show False using Suc by simp
qed
have cid' ≠ cid
proof (rule ccontr)
  assume ~ cid' ≠ cid
  then have channel cid' = Some (s, r) using t-dec can-occur-def RecvMarker
by simp
  then show False
    using Suc.premis(3) ⟨¬ cid' ≠ cid⟩ ⟨r ≠ p⟩ by auto
qed
then have cs (S t i) cid = cs (S t (Suc i)) cid
proof -
  have ‡s. channel cid = Some (s, r) using ⟨r ≠ p⟩ Suc by simp
  with RecvMarker t-dec ⟨cid' ≠ cid⟩ ⟨r ≠ p⟩ Suc.premis(3) show ?thesis
  by (cases has-snapshotted (S t i) r, auto)
qed
then show ?thesis using IH by simp
next
case (Trans r u u')
then show ?thesis
  using IH t-dec by auto
next
case (Send cid' r s u u' m)
then show ?thesis
  using IH local.step by auto
next
case (Recv cid' r s u u' m)
then have snd (cs (S t i) cid) = NotStarted
  by (simp add: IH no-initial-channel-snapshot)
  with Recv step Suc show ?thesis by (cases cid' = cid, auto)
qed
qed

```

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

```

  assumes
    trace init t final and
    snd (cs (S t i) cid) = Done and
    channel cid = Some (p, q)
  shows
    has-snapshotted (S t i) q
proof -
  show ?thesis
    using assms no-initial-channel-snapshot no-recording-cs-if-not-snapshotted by
fastforce
qed

```

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

```

  assumes

```

*trace init t final*  
**shows**  
 $\exists! i. \sim \text{has-snapshotted } (S\ t\ i)\ p \wedge \text{has-snapshotted } (S\ t\ (i+1))\ p$  (**is**  $?P$ )  
**proof** –  
**have**  $\sim \text{has-snapshotted init } p$   
**using** *no-initial-process-snapshot* **by** *auto*  
**moreover have** *has-snapshotted final*  $p$   
**using** *all-processes-snapshotted-in-final-state valid* **by** *blast*  
**moreover have** *trace*  $(S\ t\ 0)\ t\ (S\ t\ (\text{length } t))$   
**using** *assms final-is-s-t-len-t init-is-s-t-0* **by** *auto*  
**ultimately have** *ex-snap*:  $\exists i. \sim \text{has-snapshotted } (S\ t\ i)\ p \wedge \text{has-snapshotted } (S\ t\ (i+1))\ p$   
**using** *assms exists-snapshot-for-all-p* **by** *auto*  
**show** *?thesis*  
**proof** (*rule ccontr*)  
**assume**  $\sim ?P$   
**then have**  $\exists i\ j. (i \neq j) \wedge \sim \text{has-snapshotted } (S\ t\ i)\ p \wedge \text{has-snapshotted } (S\ t\ (i+1))\ p \wedge$   
 $\sim \text{has-snapshotted } (S\ t\ j)\ p \wedge \text{has-snapshotted } (S\ t\ (j+1))\ p$   
 $p$   
**using** *ex-snap* **by** *blast*  
**then have**  $\exists i\ j. (i < j) \wedge \sim \text{has-snapshotted } (S\ t\ i)\ p \wedge \text{has-snapshotted } (S\ t\ (i+1))\ p \wedge$   
 $\sim \text{has-snapshotted } (S\ t\ j)\ p \wedge \text{has-snapshotted } (S\ t\ (j+1))\ p$   
 $p$   
**by** (*meson linorder-neqE-nat*)  
**then obtain**  $i\ j$  **where**  $i < j \wedge \sim \text{has-snapshotted } (S\ t\ i)\ p \wedge \text{has-snapshotted } (S\ t\ (i+1))\ p$   
 $\wedge \sim \text{has-snapshotted } (S\ t\ j)\ p \wedge \text{has-snapshotted } (S\ t\ (j+1))\ p$   
**by** *blast*  
**have** *trace*  $(S\ t\ (i+1))\ (\text{take } (j - (i+1))\ (\text{drop } (i+1)\ t))\ (S\ t\ j)$   
**using**  $\langle i < j \rangle$  *assms exists-trace-for-any-i-j* **by** *fastforce*  
**then have** *has-snapshotted*  $(S\ t\ j)\ p$   
**using**  $\langle ps\ (S\ t\ (i+1))\ p \neq \text{None} \rangle$  *snapshot-stable* **by** *blast*  
**then show** *False* **using**  $\langle \sim \text{has-snapshotted } (S\ t\ j)\ p \rangle$  **by** *simp*  
**qed**  
**qed**

**lemma** *initial-cs-changes-implies-nonregular-event*:

**assumes**

*trace init t final* **and**

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

*snd*  $(cs\ (S\ t\ (i+1))\ cid) \neq \text{NotStarted}$  **and**

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

**shows**

$\sim \text{regular-event } (t\ !\ i)$

**proof** –

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

**proof** (*rule ccontr*)

```

    assume  $\sim i < \text{length } t$ 
    then have  $S t i = S t (i+1)$ 
      using assms(1) no-change-if-ge-length-t by auto
    then show False using assms by presburger
    qed
  then have step:  $(S t i) \vdash (t ! i) \mapsto (S t (i+1))$ 
    using assms(1) step-Suc by auto
  show ?thesis
  proof (rule ccontr)
    assume  $\sim \sim \text{regular-event } (t ! i)$ 
    then have regular-event  $(t ! i)$  by simp
    then have  $cs (S t i) \text{ cid} = cs (S t (i+1)) \text{ cid}$ 
    proof (cases isRecv  $(t ! i)$ )
      case False
      then show ?thesis
        using  $\langle \text{regular-event } (t ! i) \rangle \text{ local.step no-cs-change-if-no-event}$  by blast
    next
      case True
      then obtain  $\text{cid}' r s u u' m$  where Recv:  $t ! i = \text{Recv cid}' r s u u' m$  by
        (meson isRecv-def)
      with assms step show ?thesis
      proof (cases  $\text{cid} = \text{cid}'$ )
        case True
        then show ?thesis using assms step Recv by simp
      next
        case False
        then show ?thesis using assms step Recv by simp
      qed
    qed
  then show False using assms by simp
  qed
  qed
  qed

```

**lemma** *cs-in-initial-state-implies-not-snapshotted*:

```

  assumes
    trace init t final and
     $\text{snd } (cs (S t i) \text{ cid}) = \text{NotStarted}$  and
     $\text{channel cid} = \text{Some } (p, q)$ 
  shows
     $\sim \text{has-snapshotted } (S t i) q$ 
  proof (rule ccontr)
    assume  $\sim \sim \text{has-snapshotted } (S t i) q$ 
    then obtain  $j$  where  $j < i$   $\sim \text{has-snapshotted } (S t j) q \text{ has-snapshotted } (S t$ 
       $(j+1)) q$ 
    by (metis Suc-eq-plus1 assms(1) exists-snapshot-for-all-p computation.snapshot-stable-ver-3
      computation-axioms nat-le-linear order-le-less)
    have step-j:  $(S t j) \vdash (t ! j) \mapsto (S t (j+1))$ 
      by (metis  $\langle \neg \neg ps (S t i) q \neq \text{None} \rangle \langle \neg ps (S t j) q \neq \text{None} \rangle \langle j < i \rangle$ 
      add commute assms(1) linorder-neqE-nat no-change-if-ge-length-t order-le-less or-

```

```

der-refl plus-1-eq-Suc step-Suc
  have tr-j-i: trace (S t (j+1)) (take (i - (j+1)) (drop (j+1) t)) (S t i)
    using ⟨j < i⟩ assms(1) exists-trace-for-any-i-j by fastforce
  have ~ regular-event (t ! j)
    using step-j ⟨¬ ps (S t j) q ≠ None⟩ ⟨ps (S t (j + 1)) q ≠ None⟩ regu-
lar-event-cannot-induce-snapshot by blast
  then have isSnapshot (t ! j) ∨ isRecvMarker (t ! j)
    using nonregular-event by auto
  then have snd (cs (S t (j+1)) cid) ≠ NotStarted
  proof (elim disjE, goal-cases)
    case 1
    have occurs-on (t ! j) = q
    using ⟨¬ ps (S t j) q ≠ None⟩ ⟨ps (S t (j + 1)) q ≠ None⟩ distributed-system.no-state-change-if-no-event
distributed-system-axioms step-j by fastforce
    with 1 have t ! j = Snapshot q using isSnapshot-def by auto
    then show ?thesis using step-j assms by simp
  next
    case 2
    have occurs-on (t ! j) = q
    using ⟨¬ ps (S t j) q ≠ None⟩ ⟨ps (S t (j + 1)) q ≠ None⟩ distributed-system.no-state-change-if-no-event
distributed-system-axioms step-j by fastforce
    with 2 obtain cid' s where RecvMarker: t ! j = RecvMarker cid' q s
    by (metis event.collapse(5))
    then show ?thesis
  proof (cases cid' = cid)
    case True
    then show ?thesis using RecvMarker step-j assms by simp
  next
    case False
    have ~ has-snapshot (S t j) q
    using ⟨¬ ps (S t j) q ≠ None⟩ by auto
    moreover have ∃ r. channel cid = Some (r, q)
    by (simp add: assms(3))
    ultimately show ?thesis using RecvMarker step-j assms False by simp
  qed
qed
then have snd (cs (S t i) cid) ≠ NotStarted
  using tr-j-i cs-not-not-started-stable-trace assms by blast
then show False using assms by simp
qed

```

**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 -
  have step: (S t i) ⊢ (t ! i) ⇔ (S t (i+1)) using step-Suc assms by auto
  have isSnapshot (t ! i) ∨ isRecvMarker (t ! i)
    using assms(3) nonregular-event by blast
  then show ?thesis
  proof (elim disjE, goal-cases)
    case 1
    then show ?thesis
      using assms cs-in-initial-state-implies-not-snapshotted local.step nonregular-event-induces-snapshot by blast
    next
    case 2
    then show ?thesis
      by (metis assms(1) assms(2) assms(3) assms(4) assms(5) cs-in-initial-state-implies-not-snapshotted local.step nonregular-event-induces-snapshot)
  qed
qed

```

**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 (rule ccontr)
  assume ~ has-snapshotted (S t i) q
  have  $\llbracket$  trace init t final; ~ has-snapshotted (S t i) p; channel cid = Some (p, q)
 $\rrbracket$ 
     $\implies$  snd (cs (S t i) cid) = NotStarted
  proof (induct i)
    case 0
    then show ?case
      using init-is-s-t-0 no-initial-channel-snapshot by auto
    next
    case (Suc n)
    have step: (S t n) ⊢ (t ! n) ⇔ (S t (n+1))
      by (metis Suc.prem(2) Suc-eq-plus1 all-processes-snapshotted-in-final-state
assms(1) distributed-system.step-Suc distributed-system-axioms final-is-s-t-len-t le-add1
not-less snapshot-stable-ver-3)
    have snd (cs (S t n) cid) = NotStarted
      using Suc.hyps Suc.prem(2) assms snapshot-state-unchanged computation-axioms
local.step by fastforce
    then show ?case
      by (metis Suc.prem(1) <¬ ps (S t i) q ≠ None> assms(2) assms(3) cs-not-not-started-stable-trace
exists-trace-for-any-i no-recording-cs-if-not-snapshotted recording-state.simps(2))
  qed

```

**then show** *False*  
**using**  $\langle \neg ps (S t i) q \neq None \rangle$  *assms computation.no-initial-channel-snapshot*  
*computation-axioms no-recording-cs-if-not-snapshotted* **by** *fastforce*  
**qed**

**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** –  
**have** *trace init (take i t) (S t i)*  
**using** *assms(1) exists-trace-for-any-i* **by** *blast*  
**then have** *RecvMarker cid q p : set (take i t)*  
**by** (*metis assms(1,2,4) cs-done-implies-has-snapshotted done-only-from-recv-marker-trace*  
*computation.no-initial-process-snapshot computation-axioms init-is-s-t-0 list.discI*  
*trace.simps*)  
**then obtain** *k where t ! k = RecvMarker cid q p 0 ≤ k k < i*  
**by** (*metis add.right-neutral add-diff-cancel-right' append-Nil append-take-drop-id*  
*assms(1) exists-index take0*)  
**then have** *has-snapshotted (S t (k+1)) q*  
**by** (*metis (no-types, lifting) Suc-eq-plus1 Suc-leI assms(1,2,4) computation.cs-done-implies-has-snapshotted*  
*computation.no-change-if-ge-length-t computation-axioms less-le not-less-eq recv-marker-means-cs-Done*)  
**then show** *has-snapshotted (S t i) q*  
**using** *assms cs-done-implies-has-snapshotted* **by** *blast*  
**have** *step-k: (S t k) ⊢ (t ! k) ↦ (S t (k+1))*  
**by** (*metis Suc-eq-plus1 <k < i> add-lessD1 assms(1) assms(3) distributed-system.step-Suc*  
*distributed-system-axioms less-imp-add-positive*)  
**then have** *Marker : set (msgs (S t k) cid)*  
**proof** –  
**have** *can-occur (t ! k) (S t k)* **using** *happen-implies-can-occur step-k* **by** *blast*  
**then show** *?thesis unfolding can-occur-def <t ! k = RecvMarker cid q p>*  
**using** *hd-in-set* **by** *fastforce*  
**qed**  
**then have** *has-snapshotted (S t k) p*  
**using** *assms(1,4) no-marker-if-no-snapshot* **by** *blast*  
**then show** *has-snapshotted (S t i) p*  
**using**  $\langle k < i \rangle$  *assms(1) less-imp-le-nat snapshot-stable-ver-3* **by** *blast*  
**qed**

**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*  
**using** *assms* **proof** (*induct i j rule: S-induct*)

```

    case (S-init i)
  then show ?case by auto
next
  case (S-step i j)
  have snap-p: has-snapshotted (S t i) p
  using S-step.hyps(1) S-step.hyps(2) S-step.prems(1,2) assms(1) cs-done-implies-both-snapshotted(1)
  by auto
  have snap-q: has-snapshotted (S t i) q
  using S-step.prems(1,2) assms(1) cs-done-implies-has-snapshotted by blast
  from S-step have cs (S t i) cid = cs (S t (Suc i)) cid
  proof (cases t ! i)
    case (Snapshot r)
    from Snapshot S-step.hyps(3) snap-p have False if r = p using that by (auto
simp: can-occur-def)
    moreover
    from Snapshot S-step.hyps(3) snap-q have False if r = q using that by (auto
simp: can-occur-def)
    ultimately show ?thesis using Snapshot S-step by force
  next
  case (RecvMarker cid' r s)
  then show ?thesis
  proof (cases has-snapshotted (S t i) r)
    case True
    with RecvMarker S-step show ?thesis
    proof (cases cid = cid')
      case True
      then have cs (S t (Suc i)) cid = (fst (cs (S t i) cid), Done)
      using RecvMarker S-step by simp
      then show ?thesis
      by (metis S-step.prems(1) prod.collapse)
    qed auto
  next
  case no-snap: False
  then show ?thesis
  proof (cases cid = cid')
    case True
    then have cs (S t (Suc i)) cid = (fst (cs (S t i) cid), Done)
    using RecvMarker S-step by simp
    then show ?thesis
    by (metis S-step.prems(1) prod.collapse)
  next
  case False
  then have r ≠ p using no-snap snap-p by auto
  moreover have  $\nexists s. \text{channel } cid = \text{Some } (s, r)$ 
  using S-step(5) assms(1) cs-done-implies-has-snapshotted no-snap by blast
  ultimately show ?thesis using RecvMarker S-step False no-snap by simp
  qed
  qed
next

```

```

    case (Recv cid' r s u u' m)
    with S-step show ?thesis by (cases cid = cid', auto)
  qed auto
  with S-step show ?case by auto
qed

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 -
  obtain j where jj: j < i ~ has-snapshotted (S t j) p has-snapshotted (S t (j+1))
  p
  by (metis Suc-eq-plus1 assms(1) assms(2) exists-snapshot-for-all-p computation.snapshot-stable-ver-2 computation-axioms le-eq-less-or-eq nat-neq-iff)
  have step: (S t j) ⊢ (t ! j) ⇔ (S t (j+1))
  by (metis <¬ ps (S t j) p ≠ None> <j < i> add commute assms(1) assms(2) linorder-neqE-nat no-change-if-ge-length-t order-le-less order-refl plus-1-eq-Suc step-Suc)
  then have Marker : set (msgs (S t (j+1)) cid)
  proof -
    have ~ regular-event (t ! j)
    by (meson <¬ ps (S t j) p ≠ None> <ps (S t (j + 1)) p ≠ None> distributed-system.regular-event-cannot-induce-snapshot distributed-system-axioms local.step)
    then have isSnapshot (t ! j) ∨ isRecvMarker (t ! j) using nonregular-event
  by blast
  then show ?thesis
  proof (elim disjE, goal-cases)
    case 1
    then obtain r where Snapshot: t ! j = Snapshot r by (meson isSnapshot-def)
    then have r = p
    using jj(2) jj(3) local.step by auto
    then show ?thesis using Snapshot assms step by simp
  next
    case 2
    then obtain cid' s where RecvMarker: t ! j = RecvMarker cid' p s
    by (metis jj(2,3) distributed-system.no-state-change-if-no-event distributed-system-axioms event.sel(5) isRecvMarker-def local.step)
    moreover have cid ≠ cid'
    proof (rule ccontr)
      assume ~ cid ≠ cid'
      then have snd (cs (S t (j+1)) cid) = Done using RecvMarker step by
    simp
    then have snd (cs (S t i) cid) = Done

```

```

proof –
  assume  $a1: \text{snd } (cs \ (S \ t \ (j + 1)) \ cid) = \text{Done}$ 
  have  $f2: ps \ (S \ t \ j) \ p = \text{None}$ 
    using  $jj(2)$  by blast
  have  $j < \text{length } t$ 
    using  $assms(4)$   $jj(1)$  by linarith
  then have  $t ! j = \text{RecvMarker } cid \ q \ p$ 
    using  $f2 \ a1 \ assms(1) \ assms(5) \ cs\text{-done}\text{-implies}\text{-both}\text{-snapshotted}(1)$ 
done-only-from-recv-marker local.step by blast
    then show  $?thesis$ 
      using  $f2$  by  $(metis \ (no\text{-types}) \ \text{Suc}\text{-eq}\text{-plus1} \ assms(1) \ local.\text{step} \ \text{recv}\text{-marker}\text{-means}\text{-snapshotted})$ 
    qed
  then show False using  $assms$  by simp
qed
ultimately show  $?thesis$  using  $jj \ assms \ step$  by auto
qed
show  $?thesis$ 
proof (rule ccontr)
  let  $?t = \text{take } (i - (j+1)) \ (\text{drop } (j+1) \ t)$ 
  have  $tr\text{-}j: \text{trace } (S \ t \ (j+1)) \ ?t \ (S \ t \ i)$ 
    using  $assms(1)$  exists-trace-for-any-i-j  $jj(1)$  by fastforce
  assume  $\sim \text{Marker} : \text{set } (msgs \ (S \ t \ i) \ cid)$ 
  then obtain  $ev$  where  $ev \in \text{set } ?t \ \exists p \ q. \ ev = \text{RecvMarker } cid \ p \ q$ 
    using  $\langle \text{Marker} \in \text{set } (msgs \ (S \ t \ (j + 1)) \ cid) \rangle$  marker-must-be-delivered-2-trace
tr-j assms by blast
  obtain  $k$  where  $t ! k = ev \ j < k \ k < i$ 
    using  $\langle ev \in \text{set } (\text{take } (i - (j + 1)) \ (\text{drop } (j + 1) \ t)) \rangle$   $assms(1)$  exists-index
by fastforce
  have  $step\text{-}k: (S \ t \ k) \vdash (t ! k) \mapsto (S \ t \ (k+1))$ 
proof –
  have  $k < \text{length } t$ 
    using  $\langle k < i \rangle$   $assms(4)$  by auto
  then show  $?thesis$  using step-Suc  $assms$  by simp
qed
  have  $ev = \text{RecvMarker } cid \ q \ p$  using  $assms \ step\text{-}k$  can-occur-def
    using  $\langle \exists p \ q. \ ev = \text{RecvMarker } cid \ p \ q \rangle$   $\langle t ! k = ev \rangle$  by auto
  then have  $\text{snd } (cs \ (S \ t \ (k+1)) \ cid) = \text{Done}$ 
    using  $\langle k < i \rangle \ \langle t ! k = ev \rangle$   $assms(1) \ assms(4)$  recv-marker-means-cs-Done by
auto
  moreover have  $\text{trace } (S \ t \ (k+1)) \ (\text{take } (i - (k+1)) \ (\text{drop } (k+1) \ t)) \ (S \ t \ i)$ 
    using  $\langle k < i \rangle \ \langle \text{trace } \text{init } t \ \text{final} \rangle$  exists-trace-for-any-i-j by fastforce
  ultimately have  $\langle \text{snd } (cs \ (S \ t \ i) \ cid) = \text{Done} \rangle$ 
    using cs-done-implies-same-snapshots  $[of \ t \ \langle \text{Suc } k \rangle \ i \ cid \ p \ q]$   $\langle k < i \rangle$   $assms(1)$ 
 $assms(4) \ assms(5)$ 
    by simp
  then show False
    using  $assms$  by simp
qed

```

qed

**lemma** *no-marker-left-in-final-state*:

**assumes**

*trace init t final*

**shows**

*Marker ∉ set (msgs final cid) (is ?P)*

**proof** (*rule ccontr*)

**assume**  $\sim ?P$

**then obtain** *i* **where**  $i > \text{length } t$  *Marker ∉ set (msgs (S t i) cid)* **using** *assms*  
*l1*

**by** (*metis final-is-s-t-len-t le-neq-implies-less*)

**then have**  $S\ t\ (\text{length } t) \neq S\ t\ i$

**proof** –

**have**  $\text{msgs } (S\ t\ i)\ \text{cid} \neq \text{msgs } \text{final}\ \text{cid}$

**using**  $\langle \text{Marker} \notin \text{set } (\text{msgs } (S\ t\ i)\ \text{cid}) \rangle \langle \sim ?P \rangle$  **by** *auto*

**then show** *?thesis* **using** *final-is-s-t-len-t assms* **by** *auto*

qed

**moreover have**  $S\ t\ (\text{length } t) = S\ t\ i$

**using** *assms*  $\langle i > \text{length } t \rangle$  *less-imp-le no-change-if-ge-length-t* **by** *simp*

**ultimately show** *False* **by** *simp*

qed

**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** (*rule ccontr*)

**assume** *cs-not-done*:  $\sim \text{snd } (cs\ \text{final}\ \text{cid}) = \text{Done}$

**obtain** *i* **where** *snap-p*:  $\sim \text{has-snapshotted } (S\ t\ i)\ p\ \text{has-snapshotted } (S\ t\ (i+1))$   
*p*

**by** (*metis Suc-eq-plus1 assms(1) exists-snapshot-for-all-p*)

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

**proof** –

**have**  $S\ t\ i \neq S\ t\ (i+1)$  **using** *snap-p* **by** *auto*

**then show** *?thesis*

**by** (*meson assms(1) computation.no-change-if-ge-length-t computation-axioms*  
*le-add1 not-less*)

qed

**let**  $?t = \text{take } (\text{length } t - (i+1))\ (\text{drop } (i+1)\ t)$

**have** *tr*:  $\text{trace } (S\ t\ (i+1))\ ?t\ (S\ t\ (\text{length } t))$

**using**  $\langle i < \text{length } t \rangle$  *assms(1) exists-trace-for-any-i-j less-eq-Suc-le* **by** *fastforce*

**have**  $\text{Marker} \in \text{set } (\text{msgs } (S\ t\ (i+1))\ \text{cid})$

**proof** –

**have** *n-done*:  $\text{snd } (cs\ (S\ t\ (i+1))\ \text{cid}) \neq \text{Done}$

**proof** (*rule ccontr*)

**assume**  $\sim \text{snd} (cs (S t (i+1)) cid) \neq \text{Done}$   
**then have**  $\text{snd} (cs \text{ final } cid) = \text{Done}$   
**by** (*metis Suc-eq-plus1 Suc-leI*  $\langle i < \text{length } t \rangle$  *assms final-is-s-t-len-t computation.cs-done-implies-same-snapshots computation-axioms order-refl*)  
**then show** *False* **using** *cs-not-done* **by** *simp*  
**qed**  
**then show** *?thesis* **using** *snapshotted-and-not-done-implies-marker-in-channel snap-p assms*  
**proof** –  
**have**  $i+1 \leq \text{length } t$  **using**  $\langle i < \text{length } t \rangle$  **by** *auto*  
**then show** *?thesis*  
**using** *snapshotted-and-not-done-implies-marker-in-channel snap-p assms n-done* **by** *simp*  
**qed**  
**qed**  
**moreover have**  $\text{Marker} \notin \text{set} (msgs (S t (\text{length } t)) cid)$  **using** *final-is-s-t-len-t no-marker-left-in-final-state assms* **by** *blast*  
**ultimately have** *rm-prov*:  $\exists ev \in \text{set } ?t. (\exists q p. ev = \text{RecvMarker } cid \ q \ p)$  **using** *tr message-must-be-delivered-2-trace assms*  
**by** (*simp add: marker-must-be-delivered-2-trace*)  
**then obtain**  $k$  **where**  $\exists q p. t ! k = \text{RecvMarker } cid \ q \ p$   $i+1 \leq k$   $k < \text{length } t$   
**by** (*metis assms(1) exists-index*)  
**then have** *step*:  $(S t k) \vdash (t ! k) \mapsto (S t (k+1))$   
**by** (*metis Suc-eq-plus1-left add commute assms(1) step-Suc*)  
**then have** *RecvMarker*:  $t ! k = \text{RecvMarker } cid \ q \ p$   
**by** (*metis RecvMarker-given-channel*  $\langle \exists q p. t ! k = \text{RecvMarker } cid \ q \ p \rangle$  *assms(2) event.disc(25) event.sel(10) happen-implies-can-occur*)  
**then have**  $\text{snd} (cs (S t (k+1)) cid) = \text{Done}$   
**using**  $\langle k < \text{length } t \rangle$  *assms(1) recv-marker-means-cs-Done* **by** *blast*  
**then have**  $\text{snd} (cs \text{ final } cid) = \text{Done}$   
**using**  $\langle \text{Marker} \notin \text{set} (msgs (S t (\text{length } t)) cid) \rangle$  *all-processes-snapshotted-in-final-state assms(1) assms(2) final-is-s-t-len-t snapshotted-and-not-done-implies-marker-in-channel* **by** *fastforce*  
**then show** *False* **using** *cs-not-done* **by** *simp*  
**qed**

**lemma** *cs-NotStarted-implies-empty-cs*:

**shows**

$\llbracket \text{trace } \text{init } t \text{ final}; \text{ channel } cid = \text{Some } (p, q); i < \text{length } t; \sim \text{has-snapshotted } (S t i) \ q \rrbracket$

$\implies cs (S t i) \ cid = (\llbracket, \text{NotStarted} \rrbracket)$

**by** (*simp add: no-initial-channel-snapshot no-recording-cs-if-not-snapshotted*)

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

**assumes**

$i < j$  **and**

$j \leq \text{length } t$  **and**

*trace init t final* **and**

$\text{fst} (cs (S t i) \ cid) \neq \text{fst} (cs (S t j) \ cid)$  **and**

$channel\ cid = Some\ (p, q)$   
**shows**  
 $\exists k. i \leq k \wedge k < j \wedge (\exists p\ q\ u\ u'\ m. t ! k = Recv\ cid\ q\ p\ u\ u'\ m) \wedge (snd\ (cs\ (S\ t\ k)\ cid) = Recording)\ (is\ ?P)$   
**proof** (rule *ccontr*)  
**assume**  $\sim ?P$   
**have**  $\llbracket i < j; j \leq length\ t; \sim ?P; trace\ init\ t\ final; channel\ cid = Some\ (p, q) \rrbracket$   
 $\implies fst\ (cs\ (S\ t\ i)\ cid) = fst\ (cs\ (S\ t\ j)\ cid)$   
**proof** (induct  $j - i$  arbitrary:  $i$ )  
**case**  $0$   
**then show**  $?case$  by *linarith*  
**next**  
**case** (*Suc*  $n$ )  
**then have** *step*:  $(S\ t\ i) \vdash t ! i \mapsto (S\ t\ (Suc\ i))$   
**using** *step-Suc* by *auto*  
**then have**  $fst\ (cs\ (S\ t\ (Suc\ i))\ cid) = fst\ (cs\ (S\ t\ i)\ cid)$   
**by** (*metis* *Suc.prem*s(1) *Suc.prem*s(3) *assms*(5) *fst-cs-changed-by-recv-recording-le-eq-less-or-eq*)  
**also have**  $fst\ (cs\ (S\ t\ (Suc\ i))\ cid) = fst\ (cs\ (S\ t\ j)\ cid)$   
**proof** –  
**have**  $j - Suc\ i = n$  **using** *Suc* by *simp*  
**moreover have**  $\sim (\exists k. (Suc\ i) \leq k \wedge k < j \wedge (\exists p\ q\ u\ u'\ m. t ! k = Recv\ cid\ q\ p\ u\ u'\ m) \wedge (snd\ (cs\ (S\ t\ k)\ cid) = Recording))$   
**using**  $\langle \sim ?P \rangle$  *Suc.prem*s(3) *Suc-leD* by *blast*  
**ultimately show**  $?thesis$  **using** *Suc* by (*metis* *Suc-lessI*)  
**qed**  
**finally show**  $?case$  by *simp*  
**qed**  
**then show** *False* **using** *assms*  $\langle \sim ?P \rangle$  by *blast*  
**qed**

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

**assumes**  
 $trace\ init\ t\ final$  **and**  
 $fst\ (cs\ (S\ t\ i)\ cid) \neq []$  **and**  
 $i \leq length\ t$  **and**  
 $channel\ cid = Some\ (p, q)$   
**shows**  
 $\exists j. j < i \wedge postrecording-event\ t\ j$   
**proof** –  
**have**  $fst\ (cs\ init\ cid) = []$  **using** *no-initial-channel-snapshot* by *auto*  
**then have** *diff-cs*:  $fst\ (cs\ (S\ t\ 0)\ cid) \neq fst\ (cs\ (S\ t\ i)\ cid)$   
**using** *assms*(1) *assms*(2) *init-is-s-t-0* by *auto*  
**moreover have**  $0 < i$   
**proof** (rule *ccontr*)  
**assume**  $\sim 0 < i$   
**then have**  $0 = i$  by *auto*  
**then have**  $fst\ (cs\ (S\ t\ 0)\ cid) = fst\ (cs\ (S\ t\ i)\ cid)$   
**by** *blast*

then show *False* using *diff-cs* by *simp*  
**qed**  
 ultimately obtain  $j$  where  $j < i$  and *Recv*:  $\exists p q u u' m. t ! j = \text{Recv } cid \ q \ p$   
 $u \ u' \ m \ \text{snd} \ (cs \ (S \ t \ j) \ cid) = \text{Recording}$   
 using *assms(1) assms(3) assms(4) fst-changed-by-recv-recording-trace* by *blast*  
 then have *has-snapshotted*  $(S \ t \ j) \ q$   
 using *assms(1) assms(4) cs-recording-implies-snapshot* by *blast*  
 moreover have *regular-event*  $(t ! j)$  using *Recv* by *auto*  
 moreover have *occurs-on*  $(t ! j) = q$   
**proof** –  
 have *can-occur*  $(t ! j) \ (S \ t \ j)$   
 by (*meson Suc-le-eq*  $\langle j < i \rangle$  *assms(1) assms(3) happen-implies-can-occur*  
*le-trans step-Suc*)  
 then show *?thesis* using *Recv Recv-given-channel assms(4)* by *force*  
**qed**  
 ultimately have *postrecording-event*  $t \ j$  **unfolding** *postrecording-event* using  $\langle j$   
 $< i \rangle$  *assms(3)* by *simp*  
 then show *?thesis* using  $\langle j < i \rangle$  by *auto*  
**qed**

### 5.3.4 Relating process states

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

**assumes**  
*trace* *init*  $t$  *final* **and**  
*ps* *final*  $p = \text{Some } u$  **and**  
 $\sim$  *has-snapshotted*  $(S \ t \ i) \ p$  **and**  
*has-snapshotted*  $(S \ t \ (i+1)) \ p$  **and**  
 $i < \text{length } t$   
**shows**  
*states*  $(S \ t \ i) \ p = u$   
**proof** (*rule ccontr*)  
**assume** *states*  $(S \ t \ i) \ p \neq u$   
**then have** *ps*  $(S \ t \ (i+1)) \ p \neq \text{Some } u$   
 using *assms(1) assms(3) snapshot-state* by *force*  
**then have** *ps* *final*  $p \neq \text{Some } u$   
 by (*metis One-nat-def Suc-leI add.right-neutral add-Suc-right assms(1) assms(3)*  
*assms(4) assms(5) final-is-s-t-len-t order-refl snapshot-state snapshot-state-unchanged-trace-2*)  
**then show** *False* using *assms* by *simp*  
**qed**

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

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

```

fix p
show Some (states (S t i) p) = ps final p
proof (rule ccontr)
  obtain s where ps final p = Some s  $\vee$  ps final p = None by auto
  moreover assume Some (states (S t i) p)  $\neq$  ps final p
  ultimately have ps final p = None  $\vee$  states (S t i) p  $\neq$  s by auto
  then show False
  proof (elim disjE)
    assume ps final p = None
    then show False
      using assms all-processes-snapshotted-in-final-state by blast
  next
    assume st: states (S t i) p  $\neq$  s
    then obtain j where  $\sim$  has-snapshotted (S t j) p  $\wedge$  has-snapshotted (S t
(j+1)) p
      using Suc-eq-plus1 assms(1) exists-snapshot-for-all-p by presburger
    then show False
    proof (cases has-snapshotted (S t i) p)
      case False
        then have j  $\geq$  i
          by (metis Suc-eq-plus1  $\langle \neg$  ps (S t j) p  $\neq$  None  $\wedge$  ps (S t (j + 1)) p  $\neq$ 
None  $\rangle$  assms(1) not-less-eq-eq snapshot-stable-ver-3)

      let ?t = take (j-i) (drop i t)
      have  $\exists$  ev. ev  $\in$  set ?t  $\wedge$  regular-event ev  $\wedge$  occurs-on ev = p
      proof (rule ccontr)
        assume  $\sim$  ( $\exists$  ev. ev  $\in$  set ?t  $\wedge$  regular-event ev  $\wedge$  occurs-on ev = p)
        moreover have trace (S t i) ?t (S t j)
          using  $\langle i \leq j \rangle$  assms(1) exists-trace-for-any-i-j by blast
        ultimately have states (S t j) p = states (S t i) p
          using no-state-change-if-only-nonregular-events st by blast
        then show False
          by (metis  $\langle \neg$  ps (S t j) p  $\neq$  None  $\wedge$  ps (S t (j + 1)) p  $\neq$  None  $\rangle$ 
 $\langle$ ps final p = Some s  $\vee$  ps final p = None  $\rangle$  assms(1) final-is-s-t-len-t computation.all-processes-snapshotted-in-final-state computation.snapshot-stable-ver-3 computation-axioms linorder-not-le snapshot-state-must-have-been-reached st)
      qed

    then obtain ev where ev  $\in$  set ?t  $\wedge$  regular-event ev  $\wedge$  occurs-on ev = p
      by blast
    then obtain k where t-ind: 0  $\leq$  k  $\wedge$  k < length ?t  $\wedge$  ev = ?t ! k
      by (metis in-set-conv-nth not-le not-less-zero)
    moreover have length ?t  $\leq$  j - i by simp
    ultimately have ?t ! k = (drop i t) ! k
      using less-le-trans nth-take by blast
    also have ... = t ! (k+i)
      by (metis  $\langle$ ev  $\in$  set (take (j - i) (drop i t))  $\wedge$  regular-event ev  $\wedge$  occurs-on ev = p  $\rangle$  add commute drop-eq-Nil length-greater-0-conv length-pos-if-in-set nat-le-linear nth-drop take-eq-Nil)

```

```

finally have  $?t ! k = t ! (k+i)$  by simp
have prerecording-event  $t (k+i)$ 
proof -
  have regular-event  $(?t ! k)$ 
    using  $\langle ev \in \text{set } (take (j - i) (drop i t)) \wedge \text{regular-event } ev \wedge \text{occurs-on } ev = p \rangle$  t-ind by blast
  moreover have occurs-on  $(?t ! k) = p$ 
    using  $\langle ev \in \text{set } (take (j - i) (drop i t)) \wedge \text{regular-event } ev \wedge \text{occurs-on } ev = p \rangle$  t-ind by blast
  moreover have  $\sim \text{has-snapshotted } (S t (k+i)) p$ 
    proof -
    have  $k+i \leq j$ 
      using  $\langle \text{length } (take (j - i) (drop i t)) \leq j - i \rangle$  t-ind by linarith
      show ?thesis
      using  $\langle \neg ps (S t j) p \neq \text{None} \wedge ps (S t (j + 1)) p \neq \text{None} \rangle$   $k+i \leq j$ 
assms(1) snapshot-stable-ver-3 by blast
    qed
  ultimately show ?thesis
    using  $\langle take (j - i) (drop i t) ! k = t ! (k + i) \rangle$  prerecording-event t-ind
by auto
  qed

then show False using assms by auto
next
case True

have  $j < i$ 
proof (rule ccontr)
  assume  $\sim j < i$ 
  then have  $j \geq i$  by simp
  moreover have  $\sim \text{has-snapshotted } (S t j) p$ 
    using  $\langle \neg ps (S t j) p \neq \text{None} \wedge ps (S t (j + 1)) p \neq \text{None} \rangle$  by blast
  moreover have trace  $(S t i) (take (j - i) (drop i t)) (S t j)$ 
    using assms(1) calculation(1) exists-trace-for-any-i-j by blast
  ultimately have  $\sim \text{has-snapshotted } (S t i) p$ 
    using snapshot-stable by blast
  then show False using True by simp
qed

let  $?t = take (i-j) (drop j t)$ 
have  $\exists ev. ev \in \text{set } ?t \wedge \text{regular-event } ev \wedge \text{occurs-on } ev = p$ 
proof (rule ccontr)
  assume  $\sim (\exists ev. ev \in \text{set } ?t \wedge \text{regular-event } ev \wedge \text{occurs-on } ev = p)$ 
  moreover have trace  $(S t j) ?t (S t i)$ 
    using  $\langle j < i \rangle$  assms(1) exists-trace-for-any-i-j less-imp-le by blast
  ultimately have states  $(S t j) p = \text{states } (S t i) p$ 
    using no-state-change-if-only-nonregular-events by auto
  moreover have states  $(S t j) p = s$ 
    by (metis  $\langle \neg ps (S t j) p \neq \text{None} \wedge ps (S t (j + 1)) p \neq \text{None} \rangle$ )

```

$\langle ps \text{ final } p = \text{Some } s \vee ps \text{ final } p = \text{None} \rangle \text{ assms}(1) \text{ final-is-s-t-len-t computation.all-processes-snapshotted-in-final-state computation.snapshot-stable-ver-3 computation-axioms linorder-not-le snapshot-state-must-have-been-reached}$   
**ultimately show** *False* **using**  $\langle \text{states } (S \ t \ i) \ p \neq s \rangle$  **by** *simp*  
**qed**

**then obtain** *ev* **where**  $ev: ev \in \text{set } ?t \wedge \text{regular-event } ev \wedge \text{occurs-on } ev = p$  **by** *blast*  
**then obtain** *k* **where**  $t\text{-ind}: 0 \leq k \wedge k < \text{length } ?t \wedge ev = ?t \ ! \ k$   
**by** *(metis in-set-conv-nth le0)*  
**have**  $\text{length } ?t \leq i - j$  **by** *simp*  
**have**  $?t \ ! \ k = (\text{drop } j \ t) \ ! \ k$   
**using** *t-ind* **by** *auto*  
**also have**  $\dots = t \ ! \ (k + j)$   
**by** *(metis <ev \in set (take (i - j) (drop j t)) \wedge regular-event ev \wedge occurs-on ev = p> add.commute drop-eq-Nil length-greater-0-conv length-pos-if-in-set nat-le-linear nth-drop take-eq-Nil)*  
**finally have**  $?t \ ! \ k = t \ ! \ (k + j)$  **by** *simp*  
**have** *postrecording-event*  $t \ (k + j)$   
**proof** –  
**have** *trace*  $(S \ t \ j) \ (\text{take } k \ (\text{drop } j \ t)) \ (S \ t \ (k + j))$   
**by** *(metis add-diff-cancel-right' assms(1) exists-trace-for-any-i-j le-add-same-cancel2 t-ind)*  
**then have** *has-snapshotted*  $(S \ t \ (k + j)) \ p$   
**by** *(metis Suc-eq-plus1 Suc-leI <\neg ps (S t j) p \neq None \wedge ps (S t (j + 1)) p \neq None> <take (i - j) (drop j t) ! k = t ! (k + j)> assms(1) drop-eq-Nil ev computation.snapshot-stable-ver-3 computation-axioms le-add-same-cancel2 length-greater-0-conv length-pos-if-in-set linorder-not-le order-le-less regular-event-preserves-process-snapshots step-Suc t-ind take-eq-Nil)*  
**then show** *?thesis*  
**using**  $\langle \text{take } (i - j) \ (\text{drop } j \ t) \ ! \ k = t \ ! \ (k + j) \rangle$  *ev postrecording-event t-ind* **by** *auto*  
**qed**  
**moreover have**  $k + j < i$   
**using**  $\langle \text{length } (\text{take } (i - j) \ (\text{drop } j \ t)) \leq i - j \rangle$  *t-ind* **by** *linarith*  
**ultimately show** *False* **using** *assms(3)* **by** *simp*  
**qed**  
**qed**  
**qed**  
**qed**

### 5.3.5 Relating channel states

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

**shows**

- $\llbracket i < j; j \leq \text{length } t; \text{trace init } t \text{ final};$
- $\text{has-snapshotted } (S \ t \ i) \ p;$
- $\text{snd } (cs \ (S \ t \ i) \ cid) = \text{Recording};$
- $\text{snd } (cs \ (S \ t \ j) \ cid) = \text{Done};$

```

    channel cid = Some (p, q) ]
  => map Msg (fst (cs (S t j) cid))
    = map Msg (fst (cs (S t i) cid)) @ takeWhile ((≠) Marker) (msgs (S t i)
cid)
proof (induct j - (i+1) arbitrary: i)
  case 0
  then have j = i+1 by simp
  then have step: (S t i) ⊢ (t ! i) ⇔ (S t j) using 0.prem1 step-Suc by simp
  then have rm: ∃ q p. t ! i = RecvMarker cid q p using done-only-from-recv-marker
0.prem1 by force
  then have RecvMarker: t ! i = RecvMarker cid q p
  by (metis 0.prem1(7) RecvMarker-given-channel event.collapse(5) event.disc(25)
event.inject(5) happen-implies-can-occur local.step)
  then have takeWhile ((≠) Marker) (msgs (S t i) cid) = []
  proof -
    have can-occur (t ! i) (S t i) using happen-implies-can-occur step by simp
    then show ?thesis
  proof -
    have ∧p ms. takeWhile p ms = [] ∨ p (hd ms::'c message)
    by (metis (no-types) hd-append2 hd-in-set set-take WhileD take While-drop While-id)
    then show ?thesis
    using ⟨can-occur (t ! i) (S t i)⟩ can-occur-def rm by fastforce
  qed
qed
then show ?case
  using local.step rm by auto
next
  case (Suc n)
  then have step: (S t i) ⊢ (t ! i) ⇔ (S t (i+1))
  by (metis Suc-eq-plus1 less-SucI nat-induct-at-least step-Suc)
  have ib: i+1 < j ∧ j ≤ length t ∧ has-snapshotted (S t (i+1)) p ∧ snd (cs (S t
j) cid) = Done
  using Suc.hyps(2) Suc.prem1(2) Suc.prem1(4) Suc.prem1(6) local.step snap-
shot-state-unchanged by auto
  have snap-q: has-snapshotted (S t i) q
  using Suc(7) Suc.prem1(3) Suc cs-recording-implies-snapshot by blast
  then show ?case
  proof (cases t ! i)
  case (Snapshot r)
  then have r ≠ p
  using Suc.prem1(4) can-occur-def local.step by auto
  then have msgs (S t (i+1)) cid = msgs (S t i) cid
  using Snapshot local.step Suc.prem1(7) by auto
  moreover have cs (S t (i+1)) cid = cs (S t i) cid
  proof -
    have r ≠ q using Snapshot can-occur-def snap-q step by auto
    then show ?thesis using Snapshot local.step Suc.prem1(7) by auto
  qed
  ultimately show ?thesis using Suc ib by force

```

```

next
  case (RecvMarker cid' r s)
  then show ?thesis
  proof (cases cid = cid')
    case True
    then have takeWhile ((≠) Marker) (msgs (S t i) cid) = []
    proof -
      have can-occur (t ! i) (S t i) using happen-implies-can-occur step by simp
      then show ?thesis
      proof -
        have  $\bigwedge p$  ms. takeWhile p ms = []  $\vee$  p (hd ms::'c message)
        by (metis (no-types) hd-append2 hd-in-set set-takeWhileD takeWhile-dropWhile-id)
        then show ?thesis
          using RecvMarker True  $\langle$ can-occur (t ! i) (S t i) $\rangle$  can-occur-def by
fastforce
      qed
    qed
    moreover have snd (cs (S t (i+1)) cid) = Done
    using RecvMarker Suc.prem1 Suc.prem2 Suc.prem3 True recv-marker-means-cs-Done
  by auto
  moreover have fst (cs (S t i) cid) = fst (cs (S t (i+1)) cid)
  using RecvMarker True local.step by auto
  ultimately show ?thesis
  by (metis Suc.prem1 Suc.prem2 Suc.prem3 Suc.prem7 Suc-eq-plus1
  Suc-leI append-Nil2 cs-done-implies-same-snapshots)
next
  case False
  then have msgs (S t i) cid = msgs (S t (i+1)) cid
  proof (cases has-snapshotted (S t i) r)
    case True
    then show ?thesis using RecvMarker step Suc False by auto
  next
  case False
  with RecvMarker step Suc  $\langle$ cid  $\neq$  cid' $\rangle$  show ?thesis by (cases s = p, auto)
  qed
  moreover have cs (S t i) cid = cs (S t (i+1)) cid
  proof (cases has-snapshotted (S t i) r)
    case True
    then show ?thesis using RecvMarker step Suc False by auto
  next
  case no-snap: False
  then show ?thesis
  proof (cases r = q)
    case True
    then show ?thesis using snap-q no-snap  $\langle$ r = q $\rangle$  by simp
  next
  case False
  then show ?thesis using RecvMarker step Suc no-snap False  $\langle$ cid  $\neq$  cid' $\rangle$ 
  by simp

```

```

      qed
    qed
    ultimately show ?thesis using Suc ib by simp
  qed
next
case (Trans r u u')
then have msgs (S t i) cid = msgs (S t (i+1)) cid using step by auto
moreover have cs (S t i) cid = cs (S t (i+1)) cid using step Trans by auto
ultimately show ?thesis using Suc ib by simp
next
case (Send cid' r s u u' m)
then show ?thesis
proof (cases cid = cid')
  case True
  have marker-in-msgs: Marker ∈ set (msgs (S t i) cid)
  proof -
    have has-snapshotted (S t i) p using Suc by simp
    moreover have i < length t
      using Suc.premis(1) Suc.premis(2) less-le-trans by blast
    moreover have snd (cs (S t i) cid) ≠ Done using Suc by simp
    ultimately show ?thesis using snapshotted-and-not-done-implies-marker-in-channel
      less-imp-le using Suc by blast
  qed
  then have takeWhile ((≠) Marker) (msgs (S t i) cid) = takeWhile ((≠)
Marker) (msgs (S t (i+1)) cid)
  proof -
    have butlast (msgs (S t (i+1)) cid) = msgs (S t i) cid using step True
Send by auto
    then show ?thesis
    proof -
      have takeWhile ((≠) Marker) (msgs (S t i) cid @ [last (msgs (S t (i +
1)) cid)]) = takeWhile ((≠) Marker) (msgs (S t i) cid)
      using marker-in-msgs by force
      then show ?thesis
      by (metis (no-types) ⟨butlast (msgs (S t (i + 1)) cid) = msgs (S t i)
cid⟩ append-butlast-last-id in-set-butlastD length-greater-0-conv length-pos-if-in-set
marker-in-msgs)
    qed
  qed
  moreover have cs (S t i) cid = cs (S t (i+1)) cid using step Send by auto
  ultimately show ?thesis using ib Suc by simp
next
case False
then have msgs (S t i) cid = msgs (S t (i+1)) cid using step Send by auto
moreover have cs (S t i) cid = cs (S t (i+1)) cid using step Send by auto
ultimately show ?thesis using Suc ib by simp
qed
next
case (Recv cid' r s u u' m)

```

```

then show ?thesis
proof (cases cid = cid')
  case True
    then have msgs-ip1: Msg m # msgs (S t (i+1)) cid = msgs (S t i) cid
      using Suc Recv step by auto
    moreover have cs-ip1: cs (S t (i+1)) cid = (fst (cs (S t i) cid) @ [m],
Recording)
      using True Suc Recv step by auto
    ultimately show ?thesis
    proof -
      have map Msg (fst (cs (S t j) cid))
        = map Msg (fst (cs (S t (i+1)) cid)) @ takeWhile ((≠) Marker) (msgs
(S t (i+1)) cid)
        using Suc ib cs-ip1 by force
      moreover have map Msg (fst (cs (S t i) cid)) @ takeWhile ((≠) Marker)
(msgs (S t i) cid)
        = map Msg (fst (cs (S t (i+1)) cid)) @ takeWhile ((≠) Marker)
(msgs (S t (i+1)) cid)
      proof -
        have takeWhile ((≠) Marker) (Msg m # msgs (S t (i+1)) cid) = Msg m
# takeWhile ((≠) Marker) (msgs (S t (i + 1)) cid)
        by auto
        then have takeWhile ((≠) Marker) (msgs (S t i) cid) = Msg m # takeWhile
((≠) Marker) (msgs (S t (i + 1)) cid)
        by (metis msgs-ip1)
        then show ?thesis
          using cs-ip1 by auto
        qed
      ultimately show ?thesis by simp
    qed
  next
    case False
      then have msgs (S t i) cid = msgs (S t (i+1)) cid using step Recv by auto
      moreover have cs (S t i) cid = cs (S t (i+1)) cid using step Recv False by
auto
      ultimately show ?thesis using Suc ib by simp
    qed
  qed
qed

```

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

```

shows
  [ i ≤ j; trace init t final;
    ~ has-snapshot (S t i) p;
    ∀ k. i ≤ k ∧ k < j ⟶ ~ occurs-on (t ! k) = p;
    snd (cs (S t i) cid) = Recording;
    channel cid = Some (p, q) ]
  ⟹ map Msg (fst (cs (S t j) cid)) @ takeWhile ((≠) Marker) (msgs (S t j)
cid)

```

```

      = map Msg (fst (cs (S t i) cid)) @ takeWhile ((≠) Marker) (msgs (S t i)
cid)
      ∧ snd (cs (S t j) cid) = Recording
proof (induct j - i arbitrary: i)
  case 0
  then show ?case by auto
next
  case (Suc n)
  then have step: (S t i) ⊢ (t ! i) ⇔ (S t (i+1))
  by (metis Suc-eq-plus1 all-processes-snapshotted-in-final-state distributed-system.step-Suc
distributed-system-axioms computation.final-is-s-t-len-t computation-axioms linorder-not-le
snapshot-stable-ver-3)
  have ib: i+1 ≤ j ∧ ~ has-snapshotted (S t (i+1)) p
      ∧ (∀ k. (i+1) ≤ k ∧ k < j → ~ occurs-on (t ! k) = p) ∧ j - (i+1) = n
  by (metis Suc.hyps(2) Suc.prem(1) Suc.prem(3) Suc.prem(4) diff-Suc-1
diff-diff-left Suc-eq-plus1 Suc-leD Suc-le-eq Suc-neq-Zero cancel-comm-monoid-add-class.diff-cancel
le-neq-implies-less le-refl local.step no-state-change-if-no-event)
  have snap-q: has-snapshotted (S t i) q
  using Suc.prem(5,6) Suc.prem(2) cs-recording-implies-snapshot by blast
  then show ?case
proof (cases t ! i)
  case (Snapshot r)
  then have r ≠ p using Suc by auto
  then have msgs (S t (i+1)) cid = msgs (S t i) cid
  using Snapshot local.step Suc.prem(6) by auto
  moreover have cs (S t (i+1)) cid = cs (S t i) cid
proof -
  have r ≠ q using step can-occur-def Snapshot snap-q by auto
  then show ?thesis using Snapshot step Suc by simp
qed
  ultimately show ?thesis using Suc ib by auto
next
  case (RecvMarker cid' r s)
  then show ?thesis
proof (cases cid = cid')
  case True
  then have Marker ∈ set (msgs (S t i) cid)
  using RecvMarker RecvMarker-implies-Marker-in-set local.step by blast
  then have has-snapshotted (S t i) p
  using Suc.prem(2) no-marker-if-no-snapshot Suc by blast
  then show ?thesis using Suc.prem by simp
next
  case False
  then have msgs (S t i) cid = msgs (S t (i+1)) cid
proof (cases has-snapshotted (S t i) r)
  case True
  then show ?thesis using RecvMarker step Suc False by auto
next
  case False

```

```

    with RecvMarker step Suc ⟨cid ≠ cid'⟩ show ?thesis by (cases s = p, auto)
  qed
  moreover have cs (S t i) cid = cs (S t (i+1)) cid
  proof (cases has-snapshotted (S t i) r)
    case True
    then show ?thesis using RecvMarker step Suc False by auto
  next
    case no-snap: False
    then show ?thesis
    proof (cases r = q)
      case True
      then show ?thesis using snap-q no-snap ⟨r = q⟩ by simp
    next
      case False
      then show ?thesis using RecvMarker step Suc no-snap False ⟨cid ≠ cid'⟩
  by simp
  qed
  qed
  ultimately show ?thesis using Suc ib by auto
  qed
next
  case (Trans r u u')
  then have msgs (S t i) cid = msgs (S t (i+1)) cid using step by auto
  moreover have cs (S t i) cid = cs (S t (i+1)) cid using step Trans by auto
  ultimately show ?thesis using Suc ib by auto
next
  case (Send cid' r s u u' m)
  then have r ≠ p
  using Suc.hyps(2) Suc.prem(4) Suc by auto
  have cid ≠ cid'
  proof (rule ccontr)
    assume ~ cid ≠ cid'
    then have channel cid = channel cid' by auto
    then have (p, q) = (r, s) using can-occur-def step Send Suc by auto
    then show False using ⟨r ≠ p⟩ by simp
  qed
  then have msgs (S t i) cid = msgs (S t (i+1)) cid using step Send by simp
  moreover have cs (S t i) cid = cs (S t (i+1)) cid using step Send by auto
  ultimately show ?thesis using Suc ib by auto
next
  case (Recv cid' r s u u' m)
  then show ?thesis
  proof (cases cid = cid')
    case True
    then have msgs-ip1: Msg m # msgs (S t (i+1)) cid = msgs (S t i) cid
    using Suc Recv step by auto
    moreover have cs-ip1: cs (S t (i+1)) cid = (fst (cs (S t i) cid) @ [m],
Recording)
    using True Suc Recv step by auto

```

**ultimately show** *?thesis*  
**proof** –  
    **have**  $\text{map } \text{Msg } (\text{fst } (\text{cs } (\text{S } t \ j) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (\text{S } t \ j) \ \text{cid})$   
     $= \text{map } \text{Msg } (\text{fst } (\text{cs } (\text{S } t \ (i+1)) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (\text{S } t \ (i+1)) \ \text{cid})$   
     $\wedge \ \text{snd } (\text{cs } (\text{S } t \ j) \ \text{cid}) = \text{Recording}$   
    **using** *Suc ib cs-ip1* **by** *auto*  
    **moreover have**  $\text{map } \text{Msg } (\text{fst } (\text{cs } (\text{S } t \ i) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (\text{S } t \ i) \ \text{cid})$   
     $= \text{map } \text{Msg } (\text{fst } (\text{cs } (\text{S } t \ (i+1)) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (\text{S } t \ (i+1)) \ \text{cid})$   
    **proof** –  
    **have**  $\text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{Msg } m \ \# \ \text{msgs } (\text{S } t \ (i + 1)) \ \text{cid}) = \text{Msg } m \ \# \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (\text{S } t \ (i + 1)) \ \text{cid})$   
    **by** *fastforce*  
    **then have**  $\text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (\text{S } t \ i) \ \text{cid}) = \text{Msg } m \ \# \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (\text{S } t \ (i + 1)) \ \text{cid})$   
    **by** *(metis msgs-ip1)*  
    **then show** *?thesis*  
    **using** *cs-ip1* **by** *force*  
    **qed**  
    **ultimately show** *?thesis using cs-ip1 by simp*  
**qed**  
**next**  
    **case** *False*  
    **then have**  $\text{msgs } (\text{S } t \ i) \ \text{cid} = \text{msgs } (\text{S } t \ (i+1)) \ \text{cid}$  **using** *step Recv* **by** *auto*  
    **moreover have**  $\text{cs } (\text{S } t \ i) \ \text{cid} = \text{cs } (\text{S } t \ (i+1)) \ \text{cid}$  **using** *step Recv False* **by** *auto*  
    **ultimately show** *?thesis using Suc ib by auto*  
    **qed**  
**qed**  
**qed**

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

**shows**

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

**proof** (*induct j – i arbitrary: i*)

```

case 0
then show ?case by auto
next
  case (Suc n)
  then have step: (S t i) ⊢ (t ! i) ⇔ (S t (i+1))
  by (metis Suc-eq-plus1 all-processes-snapshotted-in-final-state distributed-system.step-Suc
distributed-system-axioms computation.final-is-s-t-len-t computation-axioms linorder-not-le
snapshot-stable-ver-3)
  have ib: i+1 ≤ j ∧ ~ has-snapshotted (S t (i+1)) q ∧ has-snapshotted (S t
(i+1)) p
    ∧ (∀ k. (i+1) ≤ k ∧ k < j ⟶ ~ occurs-on (t ! k) = q) ∧ j - (i+1) = n
    ∧ Marker : set (msgs (S t (i+1)) cid) ∧ cs (S t i) cid = cs (S t (i+1)) cid
  proof -
    have i+1 ≤ j ∧ ~ has-snapshotted (S t (i+1)) q
      ∧ (∀ k. (i+1) ≤ k ∧ k < j ⟶ ~ occurs-on (t ! k) = q) ∧ j - (i+1) = n
    by (metis Suc.hyps(2) Suc.prem(1) Suc.prem(3) Suc.prem(4) diff-Suc-1
diff-diff-left Suc-eq-plus1 Suc-leD Suc-le-eq Suc-neq-Zero cancel-comm-monoid-add-class.diff-cancel
le-neq-implies-less le-refl local.step no-state-change-if-no-event)
    moreover have has-snapshotted (S t (i+1)) p
      using Suc.prem(6) local.step snapshot-state-unchanged by auto
    moreover have Marker : set (msgs (S t (i+1)) cid)
      using Suc calculation(1) local.step recv-marker-means-snapshotted-2 by blast
    moreover have cs (S t i) cid = cs (S t (i+1)) cid
      using Suc calculation(1) no-recording-cs-if-not-snapshotted by auto
    ultimately show ?thesis by simp
  qed
then show ?case
proof (cases t ! i)
  case (Snapshot r)
  then have r ≠ q using Suc by auto
  then have takeWhile ((≠) Marker) (msgs (S t (i+1)) cid) = takeWhile ((≠)
Marker) (msgs (S t i) cid)
  proof (cases occurs-on (t ! i) = p)
  case True
  then show ?thesis
  by (metis (mono-tags, lifting) Snapshot Suc.prem(6) distributed-system.can-occur-def
event.sel(4) event.simps(29) computation-axioms computation-def happen-implies-can-occur
local.step)
  next
  case False
  then have msgs (S t (i+1)) cid = msgs (S t i) cid
  using Snapshot local.step Suc by auto
  then show ?thesis by simp
  qed
then show ?thesis using Suc ib by metis
next
  case (RecvMarker cid' r s)
  then show ?thesis
  proof (cases cid = cid')

```

```

    case True
    then have snd (cs (S t i) cid) = Done
    by (metis RecvMarker Suc.premis(2) Suc-eq-plus1 Suc-le-eq exactly-one-snapshot
    computation.no-change-if-ge-length-t computation.recv-marker-means-cs-Done com-
    putation.snapshot-stable-ver-2 computation-axioms ib nat-le-linear)
    then show ?thesis using Suc.premis by simp
  next
  case False
  then have takeWhile ((≠) Marker) (msgs (S t i) cid) = takeWhile ((≠)
  Marker) (msgs (S t (i+1)) cid)
  proof (cases has-snapshotted (S t i) r)
  case True
  with RecvMarker False step show ?thesis by auto
  next
  case no-snap: False
  then have r ≠ p
  using Suc.premis(6) by auto
  then show ?thesis using no-snap RecvMarker step Suc.premis False by auto
  qed
  then show ?thesis using Suc ib by metis
  qed
next
case (Trans r u u')
then have msgs (S t i) cid = msgs (S t (i+1)) cid using step by auto
then show ?thesis using Suc ib by auto
next
case (Send cid' r s u u' m)
then have r ≠ q
using Suc.hyps(2) Suc.premis(4) by auto
have marker: Marker ∈ set (msgs (S t i) cid) using Suc by simp
with step Send marker have takeWhile ((≠) Marker) (msgs (S t i) cid) =
takeWhile ((≠) Marker) (msgs (S t (i+1)) cid)
by (cases cid = cid', auto)
then show ?thesis using Suc ib by auto
next
case (Recv cid' r s u u' m)
then have cid' ≠ cid
by (metis Suc.hyps(2) Suc.premis(4) Suc.premis(8) distributed-system.can-occur-Recv
distributed-system-axioms event.sel(3) happen-implies-can-occur local.step option.inject
order-refl prod.inject zero-less-Suc zero-less-diff)
then have msgs (S t i) cid = msgs (S t (i+1)) cid using step Recv Suc by
simp
then show ?thesis using Suc ib by auto
qed
qed

lemma at-most-one-marker:
shows
  ⌈ trace init t final; channel cid = Some (p, q) ⌋

```

```

     $\implies \text{Marker} \notin \text{set} (\text{msgs} (S t i) \text{ cid})$ 
     $\vee (\exists !j. j < \text{length} (\text{msgs} (S t i) \text{ cid}) \wedge \text{msgs} (S t i) \text{ cid} ! j = \text{Marker})$ 
proof (induct i)
  case 0
  then show ?case using no-initial-Marker init-is-s-t-0 by auto
next
  case (Suc i)
  then show ?case
  proof (cases i < length t)
    case False
    then show ?thesis
    by (metis Suc.prem(1) final-is-s-t-len-t computation.no-change-if-ge-length-t
computation-axioms le-refl less-imp-le-nat no-marker-left-in-final-state not-less-eq)
  next
  case True
  then have step: (S t i)  $\vdash$  (t ! i)  $\mapsto$  (S t (Suc i)) using step-Suc Suc.prem by
blast
  moreover have  $\text{Marker} \notin \text{set} (\text{msgs} (S t i) \text{ cid})$ 
     $\vee (\exists !j. j < \text{length} (\text{msgs} (S t i) \text{ cid}) \wedge \text{msgs} (S t i) \text{ cid} ! j = \text{Marker})$ 
  using Suc.hyps Suc.prem(1) Suc.prem(2) by linarith
  moreover have  $\text{Marker} \notin \text{set} (\text{msgs} (S t (Suc i)) \text{ cid})$ 
     $\vee (\exists !j. j < \text{length} (\text{msgs} (S t (Suc i)) \text{ cid}) \wedge \text{msgs} (S t (Suc i)) \text{ cid} !$ 
j = Marker)
  proof (cases Marker  $\notin$  set (msgs (S t i) cid))
    case no-marker: True
    then show ?thesis
    proof (cases t ! i)
      case (Snapshot r)
      then show ?thesis
      proof (cases r = p)
        case True
        then have new-msgs: msgs (S t (Suc i)) cid = msgs (S t i) cid @ [Marker]
          using step Snapshot Suc by auto
        then show ?thesis using util-exactly-one-element no-marker by fastforce
      next
      case False
      then show ?thesis
      using Snapshot local.step no-marker Suc by auto
    qed
  next
  case (RecvMarker cid' r s)
  then show ?thesis
  proof (cases cid = cid')
    case True
    then show ?thesis
    using RecvMarker RecvMarker-implies-Marker-in-set local.step no-marker
by blast
  next
  case False

```

```

then show ?thesis
proof (cases has-snapshotted (S t i) r)
  case True
    then show ?thesis using RecvMarker step Suc False by simp
  next
    case no-snap: False
      then show ?thesis
      proof (cases r = p)
        case True
          then have msgs (S t (i+1)) cid = msgs (S t i) cid @ [Marker] using
RecvMarker step Suc.premis no-snap <cid ≠ cid'> by simp
          then show ?thesis
          proof -
            assume a1: msgs (S t (i + 1)) cid = msgs (S t i) cid @ [Marker]
            { fix nn :: nat ⇒ nat
              have ∀ ms m. ∃ n. ∀ na. ((m::'c message) ∈ set ms ∨ n < length
(ms @ [m])) ∧ (m ∈ set ms ∨ (ms @ [m]) ! n = m) ∧ (¬ na < length (ms @ [m])
∨ (ms @ [m]) ! na ≠ m ∨ m ∈ set ms ∨ na = n)
              by (metis (no-types) util-exactly-one-element)
              then have ∃ n. n < length (msgs (S t (Suc i)) cid) ∧ nn n = n ∧
msgs (S t (Suc i)) cid ! n = Marker ∨ n < length (msgs (S t (Suc i)) cid) ∧ msgs
(S t (Suc i)) cid ! n = Marker ∧ ¬ nn n < length (msgs (S t (Suc i)) cid) ∨ n <
length (msgs (S t (Suc i)) cid) ∧ msgs (S t (Suc i)) cid ! n = Marker ∧ msgs (S
t (Suc i)) cid ! nn n ≠ Marker
              using a1 by (metis Suc-eq-plus1 no-marker) }
            then show ?thesis
            by (metis (no-types))
          qed
        next
          case False
            then have msgs (S t i) cid = msgs (S t (i+1)) cid using RecvMarker
step Suc.premis <cid ≠ cid'> no-snap by simp
            then show ?thesis using Suc by simp
          qed
        qed
      next
        case (Trans r u u')
          then show ?thesis using no-marker step by auto
        next
          case (Send cid' r s u u' m)
            then show ?thesis
            proof (cases cid = cid')
              case True
                then have Marker ∉ set (msgs (S t (Suc i)) cid) using step no-marker
Send by auto
                then show ?thesis by simp
              next
                case False

```

```

    then have Marker  $\notin$  set (msgs (S t (Suc i)) cid) using step no-marker
Send by auto
    then show ?thesis by simp
qed
next
case (Recv cid' r s u' m)
with step no-marker Recv show ?thesis by (cases cid = cid', auto)
qed
next
case False
then have asm:  $\exists!j. j < \text{length} (\text{msgs} (S t i) \text{cid}) \wedge \text{msgs} (S t i) \text{cid} ! j =$ 
Marker
using Suc by simp
have len-filter: length (filter ((=) Marker) (msgs (S t i) cid)) = 1
by (metis False  $\langle$ Marker  $\notin$  set (msgs (S t i) cid)  $\vee$  ( $\exists!j. j < \text{length} (\text{msgs}$ 
(S t i) cid)  $\wedge$  msgs (S t i) cid ! j = Marker) $\rangle$  exists-one-iff-filter-one)
have snap-p: has-snapshotted (S t i) p
using False Suc.premis no-marker-if-no-snapshot by blast
show ?thesis
proof (cases t ! i)
case (Snapshot r)
have r  $\neq$  p
proof (rule ccontr)
assume  $\sim r \neq p$ 
moreover have can-occur (t ! i) (S t i) using happen-implies-can-occur
step by blast
ultimately show False using snap-p can-occur-def Snapshot by auto
qed
then have msgs (S t (Suc i)) cid = msgs (S t i) cid using step Snapshot
Suc by auto
then show ?thesis using asm by simp
next
case (RecvMarker cid' r s)
then show ?thesis
proof (cases cid = cid')
case True
then have Marker  $\#$  msgs (S t (Suc i)) cid = msgs (S t i) cid
using RecvMarker step by auto
then have Marker  $\notin$  set (msgs (S t (Suc i)) cid)
proof -
have  $\forall j. j \neq 0 \wedge j < \text{length} (\text{msgs} (S t i) \text{cid}) \longrightarrow \text{msgs} (S t i) \text{cid} ! j$ 
 $\neq$  Marker
by (metis False  $\langle$ Marker  $\#$  msgs (S t (Suc i)) cid = msgs (S t i) cid $\rangle$ 
asm length-pos-if-in-set nth-Cons-0)
then show ?thesis
proof -
assume a1:  $\forall j. j \neq 0 \wedge j < \text{length} (\text{msgs} (S t i) \text{cid}) \longrightarrow \text{msgs} (S t$ 
i) cid ! j  $\neq$  Marker
have  $\bigwedge ms n. ms \neq \text{msgs} (S t i) \text{cid} \vee \text{length} (\text{msgs} (S t (Suc i)) \text{cid})$ 

```

```

≠ n ∨ length ms = Suc n
  by (metis ‹Marker # msgs (S t (Suc i)) cid = msgs (S t i) cid›
length-Suc-conv)
  then show ?thesis
    using a1 by (metis (no-types) Suc-mono Zero-not-Suc ‹Marker #
msgs (S t (Suc i)) cid = msgs (S t i) cid› in-set-conv-nth nth-Cons-Suc)
  qed
  qed
  then show ?thesis by simp
next
case cid-neq-cid': False
then show ?thesis
proof (cases has-snapshotted (S t i) r)
case True
then have msgs (S t (Suc i)) cid = msgs (S t i) cid
  using cid-neq-cid' RecvMarker local.step snap-p by auto
then show ?thesis using asm by simp
next
case False
then have r ≠ p
  using snap-p by blast
then have msgs (S t (Suc i)) cid = msgs (S t i) cid using cid-neq-cid'
RecvMarker step False Suc by auto
then show ?thesis using asm by simp
qed
qed
next
case (Trans r u u')
then show ?thesis using step asm by auto
next
case (Send cid' r s u u' m)
then show ?thesis
proof (cases cid = cid')
case True
then have new-messages: msgs (S t (Suc i)) cid = msgs (S t i) cid @
[Msg m]
  using step Send by auto
then have ∃!j. j < length (msgs (S t (Suc i)) cid) ∧ msgs (S t (Suc i))
cid ! j = Marker
proof -
have length (filter ((=) Marker) (msgs (S t (Suc i)) cid))
  = length (filter ((=) Marker) (msgs (S t i) cid))
  + length (filter ((=) Marker) [Msg m])
  by (simp add: new-messages)
then have length (filter ((=) Marker) (msgs (S t (Suc i)) cid)) = 1
  using len-filter by simp
then show ?thesis using exists-one-iff-filter-one by metis
qed
then show ?thesis by simp

```

```

next
  case False
  then show ?thesis using step Send asm by auto
qed
next
  case (Recv cid' r s u u' m)
  then show ?thesis
  proof (cases cid = cid')
    case True
    then have new-msgs: Msg m # msgs (S t (Suc i)) cid = msgs (S t i) cid
using step Recv by auto
    then show ?thesis
    proof –
      have length (filter ((=) Marker) (msgs (S t i) cid))
        = length (filter ((=) Marker) [Msg m])
        + length (filter ((=) Marker) (msgs (S t (Suc i)) cid))
      by (metis append-Cons append-Nil filter-append len-filter length-append
new-msgs)
      then have length (filter ((=) Marker) (msgs (S t (Suc i)) cid)) = 1
      using len-filter by simp
      then show ?thesis using exists-one-iff-filter-one by metis
    qed
  next
  case False
  then show ?thesis using step Recv asm by auto
qed
qed
qed
then show ?thesis by simp
qed
qed

```

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

```

assumes
  trace init t final and
  msgs (S t i) cid ≠ [] and
  msgs (S t (i+1)) cid ≠ [] and
  last (msgs (S t i) cid) = Marker and
  last (msgs (S t (i+1)) cid) ≠ Marker and
  channel cid = Some (p, q)
shows
  ( $\exists u u' m. t ! i = \text{Send } cid \ p \ q \ u \ u' \ m$ )
proof –
  have step: (S t i) ⊢ (t ! i) ↔ (S t (i+1))
  by (metis Suc-eq-plus1-left add commute assms(1) assms(4) assms(5) le-Suc-eq
nat-le-linear nat-less-le no-change-if-ge-length-t step-Suc)
  then show ?thesis
  proof (cases t ! i)
    case (Snapshot r)

```

```

    then show ?thesis
      by (metis assms(4) assms(5) last-snoc local.step next-snapshot)
  next
    case (RecvMarker cid' r s)
    then show ?thesis
    proof (cases cid = cid')
      case True
      then have last (msgs (S t i) cid) = last (msgs (S t (i+1)) cid)
      proof -
        have Marker # msgs (S t (i + 1)) cid = msgs (S t i) cid
          using RecvMarker local.step True by auto
        then show ?thesis
          by (metis assms(3) last-ConsR)
      qed
      then show ?thesis using assms by simp
    next
      case no-snap: False
      then have last (msgs (S t i) cid) = last (msgs (S t (i+1)) cid)
      proof (cases has-snapshotted (S t i) r)
        case True
        then show ?thesis using RecvMarker step no-snap by simp
      next
        case False
        with RecvMarker step no-snap <cid ≠ cid'> assms show ?thesis by (cases
p = r, auto)
      qed
      then show ?thesis using assms by simp
    qed
  next
    case (Trans r u u')
    then show ?thesis
      using assms(4) assms(5) local.step by auto
  next
    case (Send cid' r s u u' m)
    then have cid = cid'
      by (metis (no-types, opaque-lifting) assms(4) assms(5) local.step next-send)
    moreover have (p, q) = (r, s)
    proof -
      have channel cid = channel cid' using <cid = cid'> by simp
      moreover have channel cid = Some (p, q) using assms by simp
      moreover have channel cid' = Some (r, s) using Send step can-occur-def
    by auto
      ultimately show ?thesis by simp
    qed
    ultimately show ?thesis using Send by auto
  next
    case (Recv cid' r s u u' m)
    then show ?thesis
    proof (cases cid = cid')

```

```

    case True
    then have last (msgs (S t i) cid) = last (msgs (S t (i+1)) cid)
      by (metis (no-types, lifting) Recv assms(3) assms(4) last-ConsR local.step
next-recv)
    then show ?thesis using assms by simp
  next
  case False
  then have msgs (S t i) cid = msgs (S t (i+1)) cid using Recv step by auto
  then show ?thesis using assms by simp
qed
qed
qed

```

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

```

assumes
  trace init t final and
  (S t i) ⊢ RecvMarker cid p q ⇔ (S t (i+1)) and
  channel cid = Some (q, p)
shows
  Marker ∉ set (msgs (S t (i+1)) cid)
proof -
  have new-msgs: msgs (S t i) cid = Marker # msgs (S t (i+1)) cid
    using assms(2) by auto
  have one-marker: ∃!j. j < length (msgs (S t i) cid) ∧ msgs (S t i) cid ! j =
Marker
  by (metis assms(1,3) at-most-one-marker list.set-intros(1) new-msgs)
  then obtain j where j < length (msgs (S t i) cid) msgs (S t i) cid ! j = Marker
  by blast
  then have j = 0 using one-marker new-msgs by auto
  then have ∀j. j ≠ 0 ∧ j < length (msgs (S t i) cid) ⟶ msgs (S t i) cid ! j ≠
Marker
  using one-marker
  using ⟨j < length (msgs (S t i) cid)⟩ ⟨msgs (S t i) cid ! j = Marker⟩ by blast
  then have ∀j. j < length (msgs (S t (i+1)) cid) ⟶ msgs (S t (i+1)) cid ! j
≠ Marker
  by (metis One-nat-def Suc-eq-plus1 Suc-le-eq Suc-mono le-zero-eq list.size(4)
new-msgs not-less0 nth-Cons-Suc)
  then show ?thesis
    by (simp add: in-set-conv-nth)
qed

```

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

```

shows
  [ 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) ]
  ⟹ Marker ∉ set (msgs (S t j) cid)
proof (induct j - i arbitrary: i)

```

```

case 0
then show ?case by auto
next
case (Suc n)
then have step: (S t i) ⊢ (t ! i) ⇔ (S t (i+1))
  by (metis (no-types, opaque-lifting) Suc-eq-plus1 cancel-comm-monoid-add-class.diff-cancel
distributed-system.step-Suc distributed-system-axioms less-le-trans linorder-not-less
old.nat.distinct(2) order-eq-iff)
then have Marker ∉ set (msgs (S t (i+1)) cid)
  using no-marker-and-snapshotted-implies-no-more-markers Suc step by blast
moreover have has-snapshotted (S t (i+1)) p
  using Suc.prem(4) local.step snapshot-state-unchanged by auto
ultimately show ?case
proof –
  assume a1: ps (S t (i + 1)) p ≠ None
  assume a2: Marker ∉ set (msgs (S t (i + 1)) cid)
  have f3: j ≠ i
    using Suc.hyps(2) by force
  have j – Suc i = n
    by (metis (no-types) Suc.hyps(2) Suc.prem(2) add commute add-Suc-right
add-diff-cancel-left' le-add-diff-inverse)
  then show ?thesis
    using f3 a2 a1 by (metis Suc.hyps(1) Suc.prem(1) Suc.prem(2) Suc.prem(3)
Suc.prem(6) Suc-eq-plus1-left add commute less-Suc-eq linorder-not-less)
  qed
qed

```

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

**shows**

```

[[ 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)
]] ⇒ ∀k. i ≤ k ∧ k ≤ j ⇒ Marker : set (msgs (S t k) cid)

```

**proof** (induct j – i arbitrary: i)

**case** 0

**then show** ?case **by** auto

**next**

**case** (Suc n)

**then have** step: (S t i) ⊢ (t ! i) ⇔ (S t (i+1))

**by** (metis (no-types, lifting) Suc-eq-plus1 add-lessD1 distributed-system.step-Suc
distributed-system-axioms le-add-diff-inverse order-le-less zero-less-Suc zero-less-diff)

**have** Marker : set (msgs (S t (i+1)) cid)

**proof** (rule ccontr)

**assume** asm: ~ Marker : set (msgs (S t (i+1)) cid)

**have** snap-p: has-snapshotted (S t i) p

**using** Suc.prem(1) Suc.prem(4,6) no-marker-if-no-snapshot **by** blast

**then have** has-snapshotted (S t (i+1)) p

**using** local.step snapshot-state-unchanged **by** auto

**then have**  $\text{Marker} \notin \text{set} (\text{msgs} (S t j) \text{ cid})$   
**using**  $\text{Suc.hyps}(2) \text{Suc.prem}(1) \text{Suc.prem}(3) \text{Suc.prem}(6) \text{asm}$   
 $\text{no-marker-and-snapshotted-implies-no-more-markers-trace} [of t \langle \text{Suc } i \rangle j p$   
 $\text{cid } q]$   
**by simp**  
**then show**  $\text{False}$  **using**  $\text{Suc.prem}$  **by simp**  
**qed**  
**then show**  $?case$   
**by**  $(\text{meson } \text{Suc.prem}(1) \text{Suc.prem}(3) \text{Suc.prem}(4) \text{Suc.prem}(5) \text{Suc.prem}(6)$   
 $\text{computation.snapshot-stable-ver-3} \text{computation-axioms} \text{no-marker-and-snapshotted-implies-no-more-markers-}$   
 $\text{no-marker-if-no-snapshot})$   
**qed**

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

**proof**  $(\text{induct } j - (i+1) \text{ arbitrary: } i)$   
**case 0**  
**then have**  $j = i+1$  **by simp**  
**then have**  $\text{step: } (S t i) \vdash (t ! i) \mapsto (S t (i+1))$   
**by**  $(\text{metis } 0(2) 0.\text{prem}(2) 0.\text{prem}(3) \text{Suc-eq-plus1} \text{distributed-system.step-Suc}$   
 $\text{distributed-system-axioms} \text{less-le-trans})$   
**have**  $\text{Marker} = \text{last} (\text{msgs} (S t (i+1)) \text{ cid})$   
**proof**  $(\text{rule ccontr})$   
**assume**  $\sim \text{Marker} = \text{last} (\text{msgs} (S t (i+1)) \text{ cid})$   
**then have**  $\exists u u' m. t ! i = \text{Send cid } p q u u' m$   
**proof**  $-$   
**have**  $\text{msgs} (S t (i+1)) \text{ cid} \neq []$  **using**  $0 \langle j = i+1 \rangle$  **by auto**  
**moreover have**  $\text{msgs} (S t i) \text{ cid} \neq []$  **using**  $0$  **by auto**  
**ultimately show**  $?thesis$   
**using**  $0.\text{prem}(1) 0.\text{prem}(4) 0.\text{prem}(8) \langle \text{Marker} \neq \text{last} (\text{msgs} (S t (i +$   
 $1)) \text{ cid} \rangle \text{last-changes-implies-send-when-msgs-nonempty}$  **by auto**  
**qed**  
**then show**  $\text{False}$  **using**  $0$  **by auto**  
**qed**  
**then show**  $?case$  **using**  $\langle j = i+1 \rangle$  **by simp**

**next**  
**case**  $(\text{Suc } n)$   
**then have**  $\text{step: } (S t i) \vdash (t ! i) \mapsto (S t (i+1))$   
**by**  $(\text{metis} (\text{no-types, opaque-lifting}) \text{Suc-eq-plus1} \text{distributed-system.step-Suc}$   
 $\text{distributed-system-axioms} \text{less-le-trans})$   
**have**  $\text{marker-present: Marker} : \text{set} (\text{msgs} (S t (i+1)) \text{ cid})$   
**using**  $\text{Suc.prem}$   
 $\text{marker-not-vanishing-means-always-present} [of t i j \text{ cid } p q]$  **by simp**

**moreover have**  $\text{Marker} = \text{last} (\text{msgs} (S\ t\ (i+1))\ \text{cid})$   
**proof** (*rule ccontr*)  
**assume**  $\text{asm}: \sim \text{Marker} = \text{last} (\text{msgs} (S\ t\ (i+1))\ \text{cid})$   
**then have**  $\exists u\ u'\ m. t!\ i = \text{Send}\ \text{cid}\ p\ q\ u\ u'\ m$   
**proof** –  
**have**  $\text{msgs} (S\ t\ (i+1))\ \text{cid} \neq []$  **using** *marker-present* **by** *auto*  
**moreover have**  $\text{msgs} (S\ t\ i)\ \text{cid} \neq []$  **using** *Suc* **by** *auto*  
**ultimately show** *?thesis*  
**using** *Suc.prem1* *Suc.prem4* *Suc.prem8* *asm last-changes-implies-send-when-msgs-nonempty*  
**by** *auto*  
**qed**  
**then show** *False* **using** *Suc* **by** *auto*  
**qed**  
**moreover have**  $\forall k. i+1 \leq k \wedge k < j \longrightarrow \sim (\exists u\ u'\ m. t!\ k = \text{Send}\ \text{cid}\ p\ q\ u\ u'\ m)$   
**using** *Suc.prem5* **by** *force*  
**moreover have**  $i+1 < j$  **using** *Suc* **by** *auto*  
**moreover have**  $j \leq \text{length}\ t$  **using** *Suc* **by** *auto*  
**moreover have** *trace init t final* **using** *Suc* **by** *auto*  
**moreover have**  $\text{Marker} : \text{set} (\text{msgs} (S\ t\ j)\ \text{cid})$  **using** *Suc* **by** *auto*  
**ultimately show** *?case* **using** *Suc*  
**by** (*metis diff-Suc-1 diff-diff-left*)  
**qed**

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

**assumes**  
*trace init t final*  
 $i < j$   
 $j \leq \text{length}\ t$   
 $\text{Marker} : \text{set} (\text{msgs} (S\ t\ i)\ \text{cid})$   
 $\text{Marker} : \text{set} (\text{msgs} (S\ t\ j)\ \text{cid})$   
 $\text{last} (\text{msgs} (S\ t\ i)\ \text{cid}) = \text{Marker}$   
 $\text{last} (\text{msgs} (S\ t\ j)\ \text{cid}) \neq \text{Marker}$   
 $\text{channel}\ \text{cid} = \text{Some}\ (p, q)$   
**shows**  
 $\exists k\ u\ u'\ m. i \leq k \wedge k < j \wedge t!\ k = \text{Send}\ \text{cid}\ p\ q\ u\ u'\ m$   
**proof** (*rule ccontr*)  
**assume**  $\sim (\exists k\ u\ u'\ m. i \leq k \wedge k < j \wedge t!\ k = \text{Send}\ \text{cid}\ p\ q\ u\ u'\ m)$   
**then have**  $\forall k. i \leq k \wedge k < j \longrightarrow \sim (\exists u\ u'\ m. t!\ k = \text{Send}\ \text{cid}\ p\ q\ u\ u'\ m)$  **by**  
*blast*  
**then have**  $\text{last} (\text{msgs} (S\ t\ j)\ \text{cid}) = \text{Marker}$  **using** *assms last-stays-if-no-recv-marker-and-no-send*  
**by** *blast*  
**then show** *False* **using** *assms* **by** *simp*  
**qed**

**lemma** *msg-after-marker-and-nonempty-implies-postrecording-event:*

**assumes**  
*trace init t final* **and**  
 $\text{Marker} : \text{set} (\text{msgs} (S\ t\ i)\ \text{cid})$  **and**

```

    Marker  $\neq$  last (msgs (S t i) cid) and
    i  $\leq$  length t and
    channel cid = Some (p, q)
  shows
     $\exists j. j < i \wedge \text{postrecording-event } t \ j \ (\text{is } ?P)$ 
  proof -
    let ?len = length (msgs (S t i) cid)
    have ?len  $\neq$  0 using assms(2) by auto
    have snap-p-i: has-snapshotted (S t i) p
      using assms no-marker-if-no-snapshot by blast
    obtain j where snap-p:  $\sim$  has-snapshotted (S t j) p has-snapshotted (S t (j+1))
    p
      by (metis Suc-eq-plus1 assms(1) exists-snapshot-for-all-p)
    have j < i
      by (meson assms(1) computation.snapshot-stable-ver-2 computation-axioms
        not-less snap-p(1) snap-p-i)
    have step-snap: (S t j)  $\vdash$  (t ! j)  $\mapsto$  (S t (j+1))
      by (metis Suc-eq-plus1 assms(1) l2 nat-le-linear nat-less-le snap-p(1) snapshot-stable-ver-2 step-Suc)
    have re:  $\sim$  regular-event (t ! j)
      by (meson distributed-system.regular-event-cannot-induce-snapshot distributed-system-axioms
        snap-p(1) snap-p(2) step-snap)
    have op: occurs-on (t ! j) = p
      using no-state-change-if-no-event snap-p(1) snap-p(2) step-snap by force
    have marker-last: Marker = last (msgs (S t (j+1)) cid)  $\wedge$  msgs (S t (j+1)) cid
     $\neq$  []
  proof -
    have isSnapshot (t ! j)  $\vee$  isRecvMarker (t ! j) using re nonregular-event by
    auto
    then show ?thesis
    proof (elim disjE, goal-cases)
      case 1
      then have t ! j = Snapshot p
        using op by auto
      then show ?thesis using step-snap assms by auto
    next
      case 2
      then obtain cid' r where RecvMarker: t ! j = RecvMarker cid' p r
        by (metis event.collapse(5) op)
      then have cid  $\neq$  cid'
        using RecvMarker-implies-Marker-in-set assms(1) assms(5) no-marker-if-no-snapshot
        snap-p(1) step-snap by blast
      then show ?thesis
        using assms snap-p(1) step-snap RecvMarker by auto
    qed
  qed
  then have  $\exists k \ u \ u' \ m. j+1 \leq k \wedge k < i \wedge t ! k = \text{Send } \text{cid } p \ q \ u \ u' \ m$ 
  proof -
    have j+1 < i

```

**proof** –  
**have**  $(S\ t\ (j+1)) \neq (S\ t\ i)$   
**using**  $assms(3)$  **marker-last** **by** *auto*  
**then have**  $j+1 \neq i$  **by** *auto*  
**moreover have**  $j+1 \leq i$  **using**  $\langle j < i \rangle$  **by** *simp*  
**ultimately show** *?thesis* **by** *simp*  
**qed**  
**moreover have**  $trace\ init\ t\ final$  **using**  $assms$  **by** *simp*  
**moreover have**  $Marker = last\ (msgs\ (S\ t\ (j+1))\ cid)$  **using** **marker-last** **by**  
*simp*  
**moreover have**  $Marker : set\ (msgs\ (S\ t\ (j+1))\ cid)$  **using** **marker-last** **by**  
*(simp\ add:\ marker-last)*  
**ultimately show** *?thesis* **using**  $assms\ last-changes-implies-send-when-msgs-nonempty-trace$   
**by** *simp*  
**qed**  
**then obtain**  $k$  **where**  $Send : \exists u\ u'\ m.\ j+1 \leq k \wedge k < i \wedge t ! k = Send\ cid\ p\ q$   
 $u\ u'\ m$  **by** *blast*  
**then have**  $postrecording-event\ t\ k$   
**proof** –  
**have**  $k < length\ t$  **using**  $Send\ assms$  **by** *simp*  
**moreover have**  $isSend\ (t ! k)$  **using**  $Send$  **by** *auto*  
**moreover have**  $has-snapshot\ (S\ t\ k)\ p$  **using**  $Send\ snap-p$   
**using**  $assms(1)$   $snapshot-stable-ver-3$  **by** *blast*  
**moreover have**  $occurs-on\ (t ! k) = p$  **using**  $Send$  **by** *auto*  
**ultimately show** *?thesis* **unfolding**  $postrecording-event$  **by** *simp*  
**qed**  
**then show** *?thesis* **using**  $Send$  **by** *auto*  
**qed**

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

**proof** (*induct j – i arbitrary: i*)  
**case**  $0$   
**then show** *?case* **by** *auto*  
**next**  
**case**  $(Suc\ n)$   
**then have**  $step : (S\ t\ i) \vdash (t ! i) \mapsto (S\ t\ (i+1))$   
**by** ( $metis\ (no-types,\ opaque-lifting)\ Suc-eq-plus1\ Suc-n-not-le-n\ diff-self-eq-0\ distributed-system.step-Suc\ distributed-system-axioms\ le0\ le-eq-less-or-eq\ less-le-trans$ )  
**then have**  $msgs\ (S\ t\ i)\ cid = msgs\ (S\ t\ (i+1))\ cid \wedge cs\ (S\ t\ i)\ cid = cs\ (S\ t\ (i+1))\ cid$   
**proof** –  
**have**  $\sim occurs-on\ (t ! i) = p$  **using**  $Suc$  **by** *simp*  
**moreover have**  $\sim occurs-on\ (t ! i) = q$  **using**  $Suc$  **by** *simp*  
**ultimately show** *?thesis* **using**  $step\ Suc\ same-messages-if-no-occurrence$  **by**

**blast**  
**qed**  
**moreover have**  $msgs (S t (i+1)) cid = msgs (S t j) cid \wedge cs (S t (i+1)) cid = cs (S t j) cid$   
**proof** –  
**have**  $i+1 \leq j$  **using** *Suc* **by** *linarith*  
**moreover have**  $\forall k. i+1 \leq k \wedge k < j \longrightarrow occurs-on (t ! k) \neq p \wedge occurs-on (t ! k) \neq q$  **using** *Suc* **by** *force*  
**ultimately show** *?thesis* **using** *Suc* **by** *auto*  
**qed**  
**ultimately show** *?case* **by** *simp*  
**qed**

**lemma** *snapshot-step-cs-preservation-p*:  
**assumes**  
 $c \vdash ev \mapsto c'$  **and**  
 $\sim regular-event\ ev$  **and**  
 $occurs-on\ ev = p$  **and**  
 $channel\ cid = Some\ (p, q)$   
**shows**  
 $map\ Msg\ (fst\ (cs\ c\ cid))\ @\ takeWhile\ ((\neq)\ Marker)\ (msgs\ c\ cid)$   
 $= map\ Msg\ (fst\ (cs\ c'\ cid))\ @\ takeWhile\ ((\neq)\ Marker)\ (msgs\ c'\ cid)$   
 $\wedge snd\ (cs\ c\ cid) = snd\ (cs\ c'\ cid)$   
**proof** –  
**have**  $isSnapshot\ ev \vee isRecvMarker\ ev$  **using** *assms nonregular-event* **by** *blast*  
**then show** *?thesis*  
**proof** (*elim disjE, goal-cases*)  
**case 1**  
**then have**  $Snap: ev = Snapshot\ p$  **by** (*metis event.collapse(4) assms(3)*)  
**then have**  $fst\ (cs\ c\ cid) = fst\ (cs\ c'\ cid)$   
**using** *assms(1) assms(2) regular-event same-cs-if-not-recv* **by** *blast*  
**moreover have**  $takeWhile\ ((\neq)\ Marker)\ (msgs\ c\ cid)$   
 $= takeWhile\ ((\neq)\ Marker)\ (msgs\ c'\ cid)$   
**proof** –  
**have**  $msgs\ c'\ cid = msgs\ c\ cid\ @\ [Marker]$  **using** *assms Snap* **by** *auto*  
**then show** *?thesis*  
**by** (*simp add: takeWhile-tail*)  
**qed**  
**moreover have**  $snd\ (cs\ c\ cid) = snd\ (cs\ c'\ cid)$   
**using** *Snap assms no-self-channel* **by** *fastforce*  
**ultimately show** *?thesis* **by** *simp*  
**next**  
**case 2**  
**then obtain**  $cid' r$  **where**  $RecvMarker: ev = RecvMarker\ cid'\ p\ r$  **by** (*metis event.collapse(5) assms(3)*)  
**have**  $cid \neq cid'$   
**by** (*metis 2 RecvMarker assms(1) assms(4) distributed-system.RecvMarker-given-channel distributed-system.happen-implies-can-occur distributed-system-axioms event.sel(5,10) no-self-channel*)

```

then have fst (cs c cid) = fst (cs c' cid)
  using RecvMarker assms(1) assms(2) regular-event same-cs-if-not-recv by
blast
moreover have takeWhile ((≠) Marker) (msgs c cid)
  = takeWhile ((≠) Marker) (msgs c' cid)
proof (cases has-snapshotted c p)
  case True
    then have msgs c cid = msgs c' cid using RecvMarker ⟨cid ≠ cid'⟩ assms
by auto
    then show ?thesis by simp
  next
    case False
    then have msgs c' cid = msgs c cid @ [Marker] using RecvMarker ⟨cid ≠
cid'⟩ assms by auto
    then show ?thesis
      by (simp add: takeWhile-tail)
    qed
  moreover have snd (cs c cid) = snd (cs c' cid)
proof (cases has-snapshotted c p)
  case True
    then have cs c cid = cs c' cid using RecvMarker ⟨cid ≠ cid'⟩ assms by
simp
    then show ?thesis by simp
  next
    case False
    then show ?thesis
      using RecvMarker ⟨cid ≠ cid'⟩ assms(1) assms(4) no-self-channel by auto
    qed
  ultimately show ?thesis by simp
qed
qed

```

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

```

assumes
  c ⊢ ev ↦ c' and
  ~ regular-event ev and
  occurs-on ev = q and
  channel cid = Some (p, q) and
  Marker ∉ set (msgs c cid) and
  ~ has-snapshotted c q

```

**shows**

```

  map Msg (fst (cs c cid)) @ takeWhile ((≠) Marker) (msgs c cid)
= map Msg (fst (cs c' cid)) @ takeWhile ((≠) Marker) (msgs c' cid)
∧ snd (cs c' cid) = Recording

```

**proof** –

```

have isSnapshot ev ∨ isRecvMarker ev using assms nonregular-event by blast
then show ?thesis
proof (elim disjE, goal-cases)
  case 1

```

**then have**  $Snapshot: ev = Snapshot\ q$  **by**  $(metis\ event.collapse(4)\ assms(3))$   
**then have**  $fst\ (cs\ c\ cid) = fst\ (cs\ c'\ cid)$   
**using**  $assms(1)\ assms(2)\ regular-event\ same-cs-if-not-recv$  **by**  $blast$   
**moreover have**  $takeWhile\ ((\neq)\ Marker)\ (msgs\ c\ cid)$   
 $= takeWhile\ ((\neq)\ Marker)\ (msgs\ c'\ cid)$   
**proof** –  
**have**  $msgs\ c'\ cid = msgs\ c\ cid$  **using**  $assms\ Snapshot$   
**by**  $(metis\ distributed-system.next-snapshot\ distributed-system-axioms\ eq-fst-iff$   
 $no-self-channel\ option.inject)$   
**then show**  $?thesis$  **by**  $simp$   
**qed**  
**moreover have**  $snd\ (cs\ c'\ cid) = Recording$  **using**  $assms\ Snapshot$  **by**  $auto$   
**ultimately show**  $?thesis$  **by**  $simp$   
**next**  
**case**  $2$   
**then obtain**  $cid'\ r$  **where**  $RecvMarker: ev = RecvMarker\ cid'\ q\ r$  **by**  $(metis$   
 $event.collapse(5)\ assms(3))$   
**have**  $cid \neq cid'$   
**using**  $RecvMarker\ RecvMarker-implies-Marker-in-set\ assms(1)\ assms(5)$  **by**  
 $blast$   
**have**  $fst\ (cs\ c\ cid) = fst\ (cs\ c'\ cid)$   
**using**  $assms(1)\ assms(2)\ regular-event\ same-cs-if-not-recv$  **by**  $blast$   
**moreover have**  $takeWhile\ ((\neq)\ Marker)\ (msgs\ c\ cid)$   
 $= takeWhile\ ((\neq)\ Marker)\ (msgs\ c'\ cid)$   
**proof** –  
**have**  $\exists r. channel\ cid = Some\ (q, r)$   
**using**  $assms(4)\ no-self-channel$  **by**  $auto$   
**with**  $RecvMarker\ assms\ \langle cid \neq cid' \rangle$  **have**  $msgs\ c\ cid = msgs\ c'\ cid$  **by**  $(cases$   
 $has-snapshotted\ c\ r, auto)$   
**then show**  $?thesis$  **by**  $simp$   
**qed**  
**moreover have**  $snd\ (cs\ c'\ cid) = Recording$  **using**  $assms\ RecvMarker\ \langle cid \neq$   
 $cid' \rangle$  **by**  $simp$   
**ultimately show**  $?thesis$  **by**  $simp$   
**qed**  
**qed**

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

**assumes**

$trace\ init\ t\ final$  **and**

$Marker : set\ (msgs\ (S\ t\ i)\ cid)$  **and**

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

$i \leq length\ t$

**shows**

$snd\ (cs\ (S\ t\ i)\ cid) \neq Done$

**proof**  $(rule\ ccontr)$

**assume**  $is-done: \sim snd\ (cs\ (S\ t\ i)\ cid) \neq Done$

**let**  $?t = take\ i\ t$

**have**  $tr: trace\ init\ ?t\ (S\ t\ i)$

**using** *assms(1) exists-trace-for-any-i* **by** *blast*  
**have**  $\exists q p. \text{RecvMarker } cid \ q \ p \in \text{set } ?t$   
**by** (*metis (mono-tags, lifting) assms(3) distributed-system.trace.simps distributed-system-axioms done-only-from-recv-marker-trace computation.no-initial-channel-snapshot computation-axioms is-done list.discI recording-state.simps(4) snd-conv tr*)  
**then obtain**  $j$  **where** *RecvMarker:  $\exists q p. t ! j = \text{RecvMarker } cid \ q \ p \ j < i$*   
**by** (*metis (no-types, lifting) assms(4) in-set-conv-nth length-take min.absorb2 nth-take*)  
**then have** *step-j:  $(S \ t \ j) \vdash (t ! j) \mapsto (S \ t \ (j+1))$*   
**by** (*metis Suc-eq-plus1 assms(1) distributed-system.step-Suc distributed-system-axioms assms(4) less-le-trans*)  
**then have**  $t ! j = \text{RecvMarker } cid \ q \ p$   
**by** (*metis RecvMarker(1) RecvMarker-given-channel assms(3) event.disc(25) event.sel(10) happen-implies-can-occur*)  
**then have**  $\text{Marker} \notin \text{set } (\text{msgs } (S \ t \ (j+1)) \ cid)$   
**using** *assms(1) assms(3) no-marker-after-RecvMarker step-j* **by** *presburger*  
**moreover have** *has-snapshotted  $(S \ t \ (j+1)) \ p$*   
**using** *Suc-eq-plus1  $\langle t ! j = \text{RecvMarker } cid \ q \ p \rangle$  assms(1) recv-marker-means-snapshotted snapshot-state-unchanged step-j* **by** *presburger*  
**ultimately have**  $\text{Marker} \notin \text{set } (\text{msgs } (S \ t \ i) \ cid)$   
**by** (*metis RecvMarker(2) Suc-eq-plus1 Suc-leI assms(1,3) assms(4) no-marker-and-snapshotted-implies-no-*  
**then show** *False* **using** *assms* **by** *simp*  
**qed**

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

**shows**

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

**proof** (*induct j - i arbitrary: i*)

**case**  $0$

**then show** *?case* **by** *auto*

**next**

**case** (*Suc n*)

**then have** *step:  $(S \ t \ i) \vdash (t ! i) \mapsto (S \ t \ (i+1))$*

**proof**  $-$

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

**using** *Suc.hyps(2) Suc.prem(3)* **by** *linarith*

**then show** *?thesis*

**by** (*metis (full-types) Suc.prem(1) Suc-eq-plus1 step-Suc*)

**qed**

**have**  $\text{msgs } (S \ t \ (i+1)) \ cid = \llbracket$

**proof** (*cases t ! i*)

**case** (*Snapshot r*)

**have**  $r \neq p$

**proof** (*rule ccontr*)

```

    assume  $\sim r \neq p$ 
    moreover have can-occur (t ! i) (S t i)
      using happen-implies-can-occur local.step by blast
    ultimately show False using can-occur-def Snapshot Suc by simp
  qed
  then have msgs (S t i) cid = msgs (S t (i+1)) cid
    using Snapshot local.step Suc by auto
  then show ?thesis using Suc by simp
next
case (RecvMarker cid' r s)
have cid  $\neq$  cid'
proof (rule ccontr)
  assume  $\sim cid \neq cid'$ 
  then have msgs (S t i) cid  $\neq$  []
    by (metis RecvMarker length-greater-0-conv linorder-not-less list.size(3)
local.step nat-less-le recv-marker-other-channels-not-shrinking)
  then show False using Suc by simp
qed
then show ?thesis
proof (cases has-snapshotted (S t i) r)
  case True
  then have msgs (S t (i+1)) cid = msgs (S t i) cid using RecvMarker Suc
step <cid  $\neq$  cid'> by auto
  then show ?thesis using Suc by simp
next
  case False
  have r  $\neq$  p
    using False Suc.prem5 by blast
  then show ?thesis using RecvMarker Suc step <cid  $\neq$  cid'> False by simp
qed
next
case (Trans r u u')
then show ?thesis using Suc step by simp
next
case (Send cid' r s u u' m)
have r  $\neq$  p
proof (rule ccontr)
  assume  $\sim r \neq p$ 
  then have occurs-on (t ! i) = p  $\wedge$  regular-event (t ! i) using Send by simp
  moreover have  $i \leq i \wedge i < j$  using Suc by simp
  ultimately show False using Suc.prem5 by blast
qed
have cid  $\neq$  cid'
proof (rule ccontr)
  assume  $\sim cid \neq cid'$ 
  then have channel cid = channel cid' by auto
  then have channel cid' = Some (r, s) using Send step can-occur-def by
simp
  then show False using Suc <r  $\neq$  p> < $\sim cid \neq cid'$ > by auto

```

```

qed
then have msgs (S t i) cid = msgs (S t (i+1)) cid
  using step Send Suc by simp
then show ?thesis using Suc by simp
next
case (Recv cid' r s u u' m)
have cid ≠ cid'
proof (rule ccontr)
  assume ~ cid ≠ cid'
  then have msgs (S t i) cid ≠ []
    using Recv local.step by auto
  then show False using Suc by simp
qed
then have msgs (S t i) cid = msgs (S t (i+1)) cid using Recv step by auto
then show ?thesis using Suc by simp
qed
moreover have ∀ k. i+1 ≤ k ∧ k < j ∧ regular-event (t ! k) ⟶ ~ occurs-on
(t ! k) = p
  using Suc by simp
moreover have has-snapshotted (S t (i+1)) p
  using Suc.prem5 local.step snapshot-state-unchanged [of ⟨S t i⟩ ⟨t ! i⟩ ⟨S t
(Suc i)⟩]
  by simp
moreover have j - (i+1) = n using Suc by linarith
ultimately show ?case using Suc by auto
qed

lemma last-unchanged-or-empty-if-no-events:
shows
  [| trace init t final; i ≤ j; j ≤ length t;
  msgs (S t i) cid ≠ [];
  last (msgs (S t i) cid) = Marker;
  has-snapshotted (S t i) p;
  channel cid = Some (p, q);
  ∀ k. i ≤ k ∧ k < j ∧ regular-event (t ! k) ⟶ ~ occurs-on (t ! k) = p |]
  ⟹ msgs (S t j) cid = [] ∨ (msgs (S t j) cid ≠ [] ∧ last (msgs (S t j) cid) =
Marker)
proof (induct j - i arbitrary: i)
case 0
then show ?case
  by auto
next
case (Suc n)
then have step: (S t i) ⊢ (t ! i) ⇔ (S t (i+1))
proof -
  have i < length t
  using Suc.hyps(2) Suc.prem3 by linarith
then show ?thesis
  by (metis (full-types) Suc.prem1 Suc-eq-plus1 step-Suc)

```

```

qed
have msgs-s-ip1: msgs (S t (i+1)) cid = [] ∨ (msgs (S t (i+1)) cid ≠ [] ∧ last
(msgs (S t (i+1)) cid) = Marker)
proof (cases t ! i)
  case (Snapshot r)
    have r ≠ p
    proof (rule ccontr)
      assume ~ r ≠ p
      moreover have can-occur (t ! i) (S t i)
        using happen-implies-can-occur local.step by blast
      ultimately show False using can-occur-def Snapshot Suc by simp
    qed
  then have msgs (S t i) cid = msgs (S t (i+1)) cid
    using Snapshot local.step Suc by auto
  then show ?thesis using Suc by simp
next
case (RecvMarker cid' r s)
then show ?thesis
proof (cases cid = cid')
  case True
  then have msgs (S t (i+1)) cid = []
  proof -
    have Marker # msgs (S t (i+1)) cid = msgs (S t i) cid
      using RecvMarker True local.step by auto
    then show ?thesis
  proof -
    assume a1: Marker # msgs (S t (i + 1)) cid = msgs (S t i) cid
    have i < j
      by (metis (no-types) Suc.hyps(2) Suc.prem(2) Suc-neq-Zero diff-is-0-eq
le-neq-implies-less)
    then have i < length t
      using Suc.prem(3) less-le-trans by blast
    then show ?thesis
      using a1 by (metis (no-types) Marker-in-channel-implies-not-done
RecvMarker Suc.prem(1) Suc.prem(5) Suc.prem(7) Suc-eq-plus1 Suc-le-eq True
last-ConsR last-in-set recv-marker-means-cs-Done)
    qed
  qed
  then show ?thesis by simp
next
case False
then show ?thesis
proof (cases has-snapshot (S t i) r)
  case True
  then show ?thesis
    using False RecvMarker Suc.prem(5) local.step by auto
  next
  case False
  then have r ≠ p

```

```

    using Suc.prem(6) by blast
    with RecvMarker False Suc.prem step <cid ≠ cid'> show ?thesis by auto
  qed
qed
next
  case (Trans r u u')
  then show ?thesis using Suc step by simp
next
  case (Send cid' r s u u' m)
  have r ≠ p
  proof (rule ccontr)
    assume ~ r ≠ p
    then have occurs-on (t ! i) = p ∧ regular-event (t ! i) using Send by simp
    moreover have i ≤ i ∧ i < j using Suc by simp
    ultimately show False using Suc.prem by blast
  qed
  have cid ≠ cid'
  proof (rule ccontr)
    assume ~ cid ≠ cid'
    then have channel cid = channel cid' by auto
    then have channel cid' = Some (r, s) using Send step can-occur-def by
simp
    then show False using Suc <r ≠ p> <~ cid ≠ cid'> by auto
  qed
  then have msgs (S t i) cid = msgs (S t (i+1)) cid
    using step Send by simp
  then show ?thesis using Suc by simp
next
  case (Recv cid' r s u u' m)
  then show ?thesis
  proof (cases cid = cid')
    case True
    then have msgs (S t i) cid = Msg m # msgs (S t (i+1)) cid
      using Recv local.step by auto
    then have last (msgs (S t (i+1)) cid) = Marker
      by (metis Suc.prem(5) last.simps message.simps(3))
    then show ?thesis by blast
  next
    case False
    then have msgs (S t i) cid = msgs (S t (i+1)) cid using Recv step by auto
    then show ?thesis using Suc by simp
  qed
qed
then show ?case
proof (elim disjE, goal-cases)
  case 1
  moreover have trace init t final using Suc by simp
  moreover have i+1 ≤ j using Suc by simp
  moreover have j ≤ length t using Suc by simp

```

**moreover have** *has-snapshotted* ( $S\ t\ (i+1)$ )  $p$   
**using** *Suc.prem*s(6) *local.step snapshot-state-unchanged* **by** *auto*  
**moreover have**  $j - (i+1) = n$  **using** *Suc* **by** *linarith*  
**moreover have**  $\forall k. i+1 \leq k \wedge k < j \wedge \text{regular-event } (t\ !\ k) \longrightarrow \sim \text{occurs-on}$   
 $(t\ !\ k) = p$   
**using** *Suc* **by** *auto*  
**ultimately have** *msgs* ( $S\ t\ j$ )  $cid = []$  **using** *keep-empty-if-no-events* *Suc.prem*s(7)  
**by** *blast*  
**then show** *?thesis* **by** *simp*  
**next**  
**case** 2  
**moreover have** *trace init t final* **using** *Suc* **by** *simp*  
**moreover have**  $i+1 \leq j$  **using** *Suc* **by** *simp*  
**moreover have**  $j \leq \text{length } t$  **using** *Suc* **by** *simp*  
**moreover have** *has-snapshotted* ( $S\ t\ (i+1)$ )  $p$   
**using** *Suc.prem*s(6) *local.step snapshot-state-unchanged* **by** *auto*  
**moreover have**  $j - (i+1) = n$  **using** *Suc* **by** *linarith*  
**moreover have**  $\forall k. i+1 \leq k \wedge k < j \wedge \text{regular-event } (t\ !\ k) \longrightarrow \sim \text{occurs-on}$   
 $(t\ !\ k) = p$   
**using** *Suc* **by** *auto*  
**ultimately show** *?thesis* **using** *Suc.prem*s(7) *Suc.hyps* **by** *blast*  
**qed**  
**qed**

**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*  
**proof** (*unfold cs-equal-to-snapshot-def*, *rule allI*, *rule impI*)  
**fix**  $cid$   
**assume** *channel*  $cid \neq \text{None}$   
**then obtain**  $p\ q$  **where** *chan*: *channel*  $cid = \text{Some } (p, q)$  **by** *auto*  
**have** *cs-done*: *snd* (*cs* ( $S\ t\ (\text{length } t)$ )  $cid$ ) = *Done*  
**using** *chan all-channels-done-in-final-state* *assms*(1) *final-is-s-t-len-t* **by** *blast*  
**have** *filter* ( $(\neq)\ \text{Marker}$ ) (*msgs* ( $S\ t\ i$ )  $cid$ ) = *takeWhile* ( $(\neq)\ \text{Marker}$ ) (*msgs* ( $S\ t\ i$ )  $cid$ ) (*is ?B*)  
**proof** (*rule ccontr*)  
**let**  $?m = \text{msgs } (S\ t\ i)\ cid$   
**assume**  $\sim ?B$   
**then obtain**  $j\ k$  **where** *range*:  $j < k\ k < \text{length } ?m$  **and**  $?m\ !\ j = \text{Marker } ?m$   
 $! k \neq \text{Marker}$   
**using** *filter-neq-takeWhile* **by** *metis*  
**then have**  $\text{Marker} \in \text{set } ?m$   
**by** (*metis less-trans nth-mem*)  
**moreover have** *last*  $?m \neq \text{Marker}$

**proof** –  
**have**  $\forall l. l < \text{length } ?m \wedge l \neq j \longrightarrow ?m ! l \neq \text{Marker}$   
**using**  $\text{chan } \langle j < k \rangle \langle k < \text{length } (\text{msgs } (S \ t \ i) \ cid) \rangle \langle \text{msgs } (S \ t \ i) \ cid ! j = \text{Marker} \rangle \text{assms}(1) \text{at-most-one-marker calculation less-trans}$  **by** *blast*  
**moreover have**  $\text{last } ?m = ?m ! (\text{length } ?m - 1)$   
**by**  $(\text{metis } \langle \text{Marker} \in \text{set } (\text{msgs } (S \ t \ i) \ cid) \rangle \text{empty-iff last-conv-nth list.set}(1))$   
**moreover have**  $\text{length } ?m - 1 \neq j$  **using** *range* **by** *auto*  
**ultimately show** *?thesis* **using** *range* **by** *auto*  
**qed**  
**moreover have**  $i \leq \text{length } t$   
**using**  $\text{chan } \text{assms}(1) \text{calculation}(1) \text{computation.exists-next-marker-free-state}$   
 $\text{computation.no-change-if-ge-length-t computation-axioms nat-le-linear}$  **by** *fastforce*  
**ultimately have**  $\exists j. j < i \wedge \text{postrecording-event } t \ j$   
**using**  $\text{chan } \text{assms}(1) \text{msg-after-marker-and-nonempty-implies-postrecording-event}$   
**by** *auto*  
**then show** *False* **using** *assms* **by** *auto*  
**qed**  
**moreover have**  $\text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (S \ t \ i) \ cid) = \text{map } \text{Msg } (\text{fst } (\text{cs } \text{final } \text{cid}))$   
**proof**  $(\text{cases } \text{snd } (\text{cs } (S \ t \ i) \ cid))$   
**case** *NotStarted*

show that q and p have to snapshot, and then reduce it to the case below depending on the order they snapshotted in

**have**  $\text{nsq}: \sim \text{has-snapshotted } (S \ t \ i) \ q$   
**using** *NotStarted*  $\text{chan } \text{assms}(1) \text{cs-in-initial-state-implies-not-snapshotted}$  **by** *auto*  
**obtain**  $j$  **where**  $\text{snap-q}: \sim \text{has-snapshotted } (S \ t \ j) \ q \ \text{has-snapshotted } (S \ t \ (j+1))$   
 $q$   
**by**  $(\text{metis } \text{Suc-eq-plus1 } \text{assms}(1) \ \text{exists-snapshot-for-all-p})$   
**have**  $\text{step-q}: (S \ t \ j) \vdash (t ! j) \mapsto (S \ t \ (j+1))$   
**by**  $(\text{metis } \langle \neg \text{ps } (S \ t \ j) \ q \neq \text{None} \rangle \ \text{add.commute } \text{assms}(1) \ \text{le-SucI } \text{le-eq-less-or-eq}$   
 $\text{le-refl } \text{linorder-neqE-nat no-change-if-ge-length-t plus-1-eq-Suc snap-q step-Suc})$   
**obtain**  $k$  **where**  $\text{snap-p}: \sim \text{has-snapshotted } (S \ t \ k) \ p \ \text{has-snapshotted } (S \ t \ (k+1))$   
 $p$   
**by**  $(\text{metis } \text{Suc-eq-plus1 } \text{assms}(1) \ \text{exists-snapshot-for-all-p})$   
**have**  $\text{bound}: i \leq j$   
**proof**  $(\text{rule } \text{ccontr})$   
**assume**  $\sim i \leq j$   
**then have**  $i \geq j+1$  **by** *simp*  
**then have**  $\text{has-snapshotted } (S \ t \ i) \ q$   
**by**  $(\text{meson } \text{assms}(1) \ \text{computation.snapshot-stable-ver-3 computation-axioms snap-q}(2))$   
**then show** *False* **using** *nsq* **by** *simp*  
**qed**  
**have**  $\text{step-p}: (S \ t \ k) \vdash (t ! k) \mapsto (S \ t \ (k+1))$   
**by**  $(\text{metis } \langle \neg \text{ps } (S \ t \ k) \ p \neq \text{None} \rangle \ \text{add.commute } \text{assms}(1) \ \text{le-SucI } \text{le-eq-less-or-eq}$   
 $\text{le-refl } \text{linorder-neqE-nat no-change-if-ge-length-t plus-1-eq-Suc snap-p step-Suc})$   
**have**  $\text{oq}: \text{occurs-on } (t ! j) = q$

```

proof (rule ccontr)
  assume  $\sim$  occurs-on (t ! j) = q
  then have has-snapshotted (S t j) q = has-snapshotted (S t (j+1)) q
    using no-state-change-if-no-event step-q by auto
  then show False using snap-q by blast
qed
have op: occurs-on (t ! k) = p
proof (rule ccontr)
  assume  $\sim$  occurs-on (t ! k) = p
  then have has-snapshotted (S t k) p = has-snapshotted (S t (k+1)) p
    using no-state-change-if-no-event step-p by auto
  then show False using snap-p by blast
qed
have p  $\neq$  q using chan no-self-channel by blast
then have j  $\neq$  k using oq op event-occurs-on-unique by blast
show ?thesis
proof (cases j < k)
  case True
  then have msgs (S t i) cid = msgs (S t j) cid  $\wedge$  cs (S t i) cid = cs (S t j)
cid
  proof –
  have  $\forall k. i \leq k \wedge k < j \longrightarrow$  occurs-on (t ! k)  $\neq$  p  $\wedge$  occurs-on (t ! k)  $\neq$  q
(is ?Q)
  proof (rule ccontr)
  assume  $\sim$  ?Q
  then obtain l where range: i  $\leq$  l l < j and occurs-on (t ! l) = p  $\vee$ 
occurs-on (t ! l) = q by blast
  then show False
  proof (elim disjE, goal-cases)
  case 1
  then show ?thesis
  proof (cases regular-event (t ! l))
  case True
  have l < k using range <j < k> by simp
  have  $\sim$  has-snapshotted (S t l) p using snap-p(1) range <j < k>
snapshot-stable-ver-3 assms(1) by simp
  then have prerecording-event t l using True 1 prerecording-event
using s-def snap-q(1) snap-q(2) by fastforce
  then show False using assms range by blast
  next
  case False
  then have step-l: (S t l)  $\vdash$  t ! l  $\mapsto$  (S t (l+1))
by (metis (no-types, lifting) Suc-eq-plus1 Suc-lessD True assms(1)
distributed-system.step-Suc distributed-system-axioms less-trans-Suc linorder-not-le
local.range(2) s-def snap-p(1) snap-p(2) take-all)
  then have has-snapshotted (S t (l+1)) p using False nonregu-
lar-event-induces-snapshot
by (metis 1(3) snapshot-state-unchanged)
  then show False

```

```

      by (metis Suc-eq-plus1 Suc-leI True assms(1) less-imp-le-nat
local.range(2) snap-p(1) snapshot-stable-ver-3)
    qed
  next
  case 2
  then show ?thesis
  proof (cases regular-event (t ! l))
    case True
      have ~ has-snapshotted (S t l) q using snap-q(1) range ⟨j < k⟩
snapshot-stable-ver-3 assms(1) by simp
      then have prerecording-event t l using True 2 prerecording-event
using s-def snap-q(2) by fastforce
      then show False using assms range by blast
    next
    case False
      then have step-l: (S t l) ⊢ t ! l ⇨ (S t (l+1))
      by (metis (no-types, lifting) Suc-eq-plus1 Suc-lessD True assms(1)
distributed-system.step-Suc distributed-system-axioms less-trans-Suc linorder-not-le
local.range(2) s-def snap-p(1) snap-p(2) take-all)
      then have has-snapshotted (S t (l+1)) q using False nonregu-
lar-event-induces-snapshot
      by (metis 2(3) snapshot-state-unchanged)
      then show False
      by (metis Suc-eq-plus1 Suc-leI assms(1) range(2) snap-q(1) snap-
shot-stable-ver-3)
    qed
  qed
  moreover have j ≤ length t
  proof (rule ccontr)
    assume ~ j ≤ length t
    then have S t j = S t (j+1) using no-change-if-ge-length-t assms by
simp
    then show False using snap-q by auto
  qed
  ultimately show ?thesis using chan same-messages-if-no-occurrence-trace
assms less-imp-le bound by blast
  qed
  moreover have map Msg (fst (cs (S t j) cid)) @ takeWhile ((≠) Marker)
(msgs (S t j) cid)
    = map Msg (fst (cs (S t (j+1)) cid)) @ takeWhile ((≠) Marker)
(msgs (S t (j+1)) cid)
    ∧ snd (cs (S t (j+1)) cid) = Recording
  proof -
    have ~ regular-event (t ! j) using snap-q
    using regular-event-cannot-induce-snapshot step-q by blast
    moreover have Marker ∉ set (msgs (S t j) cid)
    by (meson chan True assms(1) computation.no-marker-if-no-snapshot
computation.snapshot-stable-ver-2 computation-axioms less-imp-le-nat snap-p(1))

```

**ultimately show** *?thesis* **using** *oq snapshot-step-cs-preservation-q step-q chan snap-q(1)* **by** *blast*

**qed**

**moreover have**  $\text{map } \text{Msg } (\text{fst } (\text{cs } (S \ t \ k) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker})$   
 $(\text{msgs } (S \ t \ k) \ \text{cid})$   
 $= \text{map } \text{Msg } (\text{fst } (\text{cs } (S \ t \ (j+1)) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker})$   
 $(\text{msgs } (S \ t \ (j+1)) \ \text{cid})$

**proof** –

**have**  $\text{snd } (\text{cs } (S \ t \ (j+1)) \ \text{cid}) = \text{Recording}$  **using** *calculation* **by** *simp*

**moreover have**  $\forall a. j+1 \leq a \wedge a < k \longrightarrow \sim \text{occurs-on } (t \ ! \ a) = p$  **(is** *?R***)**

**proof** (*rule ccontr*)

**assume**  $\sim ?R$

**then obtain**  $a$  **where**  $j+1 \leq a < k$  **and** *ocp: occurs-on (t ! a) = p* **by**

*blast*

**have**  $a < \text{length } t$

**proof** –

**have**  $k < \text{length } t$

**proof** (*rule ccontr*)

**assume**  $\sim k < \text{length } t$

**then have**  $S \ t \ k = S \ t \ (k+1)$

**using** *assms(1) no-change-if-ge-length-t* **by** *auto*

**then show** *False* **using** *snap-p* **by** *auto*

**qed**

**then show** *?thesis* **using**  $\langle a < k \rangle$  **by** *simp*

**qed**

**then show** *False*

**proof** (*cases regular-event (t ! a)*)

**case** *True*

**have**  $\sim \text{has-snapshotted } (S \ t \ a) \ p$

**by** (*meson*  $\langle a < k \rangle$  *assms(1) computation.snapshot-stable-ver-2 computation-axioms less-imp-le-nat snap-p(1)*)

**then have** *prerecording-event t a* **using**  $\langle a < \text{length } t \rangle$  *ocp True prerecording-event* **by** *simp*

**then show** *False* **using**  $\langle j+1 \leq a \rangle \langle j \geq i \rangle$  *assms* **by** *auto*

**next**

**case** *False*

**then have**  $(S \ t \ a) \vdash (t \ ! \ a) \mapsto (S \ t \ (a+1))$

**using**  $\langle a < \text{length } t \rangle$  *assms(1) step-Suc* **by** *auto*

**then have**  $\text{has-snapshotted } (S \ t \ (a+1)) \ p$

**by** (*metis False ocp nonregular-event-induces-snapshot snapshot-state-unchanged*)

**then show** *False*

**by** (*metis Suc-eq-plus1 Suc-leI*  $\langle a < k \rangle$  *assms(1) snap-p(1) snapshot-stable-ver-3*)

**qed**

**qed**

**moreover have**  $\sim \text{has-snapshotted } (S \ t \ (j+1)) \ p$

**by** (*metis Suc-eq-plus1 Suc-le-eq True assms(1) computation.snapshot-stable-ver-2 computation-axioms snap-p(1)*)

**ultimately show** *?thesis* **using** *chan cs-when-recording-2 True assms(1)*

```

by auto
  qed
  moreover have map Msg (fst (cs (S t k) cid)) @ takeWhile ((≠) Marker)
    (msgs (S t k) cid)
    = map Msg (fst (cs (S t (k+1)) cid)) @ takeWhile ((≠) Marker)
    (msgs (S t (k+1)) cid)
  proof -
    have ¬ regular-event (t ! k)
    using regular-event-preservation-process-snapshots snap-p(1) snap-p(2) step-p
by force
  then show ?thesis
  using chan computation.snapshot-step-cs-preservation-p computation-axioms
op step-p by fastforce
  qed
  moreover have map Msg (fst (cs (S t (k+1)) cid)) @ takeWhile ((≠) Marker)
    (msgs (S t (k+1)) cid)
    = map Msg (fst (cs final cid))
  proof -
    have f1: ∀ f p pa pb c ca es n a na. ¬ computation f p pa pb (c::('a, 'b, 'c) config-
      uration) ca ∨ ¬ distributed-system.trace f p pa pb c es ca ∨ ps (distributed-system.s
      f p pa pb c es n) a = None ∨ ¬ n ≤ na ∨ ps (distributed-system.s f p pa pb c es
      na) a ≠ None
    by (meson computation.snapshot-stable-ver-2)
    have f2: computation channel trans send recv init (S t (length t))
    using assms(1) final-is-s-t-len-t computation-axioms by blast
    have f3: trace init t (S t (length t))
    using assms(1) final-is-s-t-len-t by blast
    have f4: ps (S t k) p = None
    by (meson snap-p(1))
    then have f5: k < length t
    using f3 f2 f1 by (metis le-eq-less-or-eq not-le s-def snap-p(2) take-all)
    have ¬ regular-event (t ! k)
    using f4 by (meson distributed-system.regular-event-cannot-induce-snapshot
      distributed-system-axioms snap-p(2) step-p)
    then have f6: map Msg (fst (cs (S t k) cid)) @ takeWhile ((≠) Marker)
      (msgs (S t k) cid) = map Msg (fst (cs (S t (k + 1)) cid)) @ takeWhile ((≠)
      Marker) (msgs (S t (k + 1)) cid) ∧ snd (cs (S t k) cid) = snd (cs (S t (k + 1))
      cid)
    using chan computation.snapshot-step-cs-preservation-p computation-axioms
op step-p by fastforce
    then have f7: snd (cs (S t (k + 1)) cid) ≠ Done
    using f5 f4 by (metis (no-types) assms(1) chan cs-done-implies-both-snapshotted(1))
    have j + 1 ≤ k + 1
    using True by linarith
    then have snd (cs (S t (k + 1)) cid) = Recording
    using f7 f3 f2 f1 by (meson chan computation.cs-in-initial-state-implies-not-snapshotted
      recording-state.exhaust snap-q(2))
    then show ?thesis
    using f6 f5 by (metis (no-types) Suc-eq-plus1 Suc-leI assms(1) chan cs-done

```

*cs-done-implies-both-snapshotted(1) cs-when-recording final-is-s-t-len-t le-eq-less-or-eq*  
*snap-p(2))*  
**qed**  
**ultimately show** *?thesis*  
**by** (*metis (no-types, lifting) chan Nil-is-map-conv assms(1) computation.no-initial-channel-snapshot computation-axioms fst-conv no-recording-cs-if-not-snapshotted self-append-conv2 snap-q(1)*)  
**next**  
**case** *False*  
**then have**  $k < j$  **using**  $\langle j \neq k \rangle$  *False* **by** *simp*  
**then have**  $\text{map } \text{Msg } (\text{fst } (\text{cs } (S \ t \ i) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (S \ t \ i) \ \text{cid})$   
 $= \text{map } \text{Msg } (\text{fst } (\text{cs } (S \ t \ j) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (S \ t \ j) \ \text{cid})$   
**proof** (*cases*  $i \leq k$ )  
**case** *True*  
**then have**  $\text{msgs } (S \ t \ i) \ \text{cid} = \text{msgs } (S \ t \ k) \ \text{cid} \wedge \text{cs } (S \ t \ i) \ \text{cid} = \text{cs } (S \ t \ k) \ \text{cid}$   
**proof** –  
**have**  $\forall j. i \leq j \wedge j < k \longrightarrow \text{occurs-on } (t \ ! \ j) \neq p \wedge \text{occurs-on } (t \ ! \ j) \neq q$   
**(is** *?Q*)  
**proof** (*rule ccontr*)  
**assume**  $\sim ?Q$   
**then obtain**  $l$  **where**  $\text{range: } i \leq l \ l < k$  **and**  $\text{occurs-on } (t \ ! \ l) = p \vee \text{occurs-on } (t \ ! \ l) = q$  **by** *blast*  
**then show** *False*  
**proof** (*elim disjE, goal-cases*)  
**case** *1*  
**then show** *?thesis*  
**proof** (*cases regular-event (t ! l)*)  
**case** *True*  
**have**  $l < k$  **using**  $\text{range } \langle k < j \rangle$  **by** *simp*  
**have**  $\sim \text{has-snapshotted } (S \ t \ l) \ p$  **using** *snap-p(1) range <k < j> snapshot-stable-ver-3 assms(1)* **by** *simp*  
**then have** *prerecording-event t l* **using** *True 1 prerecording-event*  
**using** *s-def snap-p* **by** *fastforce*  
**then show** *False* **using** *assms range* **by** *blast*  
**next**  
**case** *False*  
**then have** *step-l: (S t l) ⊢ t ! l ⇔ (S t (l+1))*  
**by** (*metis (no-types, lifting) Suc-eq-plus1 Suc-lessD assms(1) distributed-system.step-Suc distributed-system-axioms less-trans-Suc linorder-not-le local.range(2) s-def snap-p(1) snap-p(2) take-all*)  
**then have** *has-snapshotted (S t (l+1)) p* **using** *False nonregular-event-induces-snapshot*  
**by** (*metis 1(3) snapshot-state-unchanged*)  
**then show** *False*  
**by** (*metis Suc-eq-plus1 Suc-leI assms(1) local.range(2) snap-p(1) snapshot-stable-ver-3*)

```

    qed
  next
  case 2
  then show ?thesis
  proof (cases regular-event (t ! l))
    case True
    have ~ has-snapshotted (S t l) p using snap-p(1) range ⟨k < j⟩
    snapshot-stable-ver-3 assms(1) by simp
    moreover have l < length t
    using ⟨k < j⟩ local.range(2) s-def snap-q(1) snap-q(2) by force
    ultimately have prerecording-event t l using True 2 prerecording-event

    proof -
      have l ≤ j
      by (meson False ⟨l < k⟩ less-trans not-less)
      then show ?thesis
      by (metis (no-types) True ⟨l < length t⟩ ⟨occurs-on (t ! l)
= q⟩ assms(1) computation.prerecording-event computation.snapshot-stable-ver-2
computation-axioms snap-q(1))
      qed
    then show False using assms range by blast
  next
  case False
  then have step-l: (S t l) ⊢ t ! l ⇔ (S t (l+1))
  by (metis (no-types, lifting) Suc-eq-plus1 Suc-lessD assms(1)
distributed-system.step-Suc distributed-system-axioms less-trans-Suc linorder-not-le
local.range(2) s-def snap-p(1) snap-p(2) take-all)
  then have has-snapshotted (S t (l+1)) q using False nonregu-
lar-event-induces-snapshot
  by (metis 2(3) snapshot-state-unchanged)
  then show False
  by (metis Suc-eq-plus1 Suc-leI ⟨k < j⟩ assms(1) less-imp-le-nat
local.range(2) snap-q(1) snapshot-stable-ver-3)
  qed
  qed
  moreover have k ≤ length t
  proof (rule ccontr)
    assume ~ k ≤ length t
    then have S t k = S t (k+1) using no-change-if-ge-length-t assms by
simp
    then show False using snap-p by auto
  qed
  ultimately show ?thesis using chan same-messages-if-no-occurrence-trace
assms True less-imp-le by auto
  qed
  moreover have map Msg (fst (cs (S t k) cid)) @ takeWhile ((≠) Marker)
(msgs (S t k) cid)
= map Msg (fst (cs (S t (k+1)) cid)) @ takeWhile ((≠) Marker)

```

```

(msgs (S t (k+1)) cid)
  ∧ snd (cs (S t (k+1)) cid) = NotStarted
proof –
  have ~ regular-event (t ! k) using snap-p
  using regular-event-cannot-induce-snapshot step-p by blast
  then show ?thesis
using calculation op snapshot-step-cs-preservation-p step-p chan NotStarted
by auto
qed
moreover have map Msg (fst (cs (S t (k+1)) cid)) @ takeWhile ((≠)
Marker) (msgs (S t (k+1)) cid)
  = map Msg (fst (cs (S t j) cid)) @ takeWhile ((≠) Marker) (msgs
(S t j) cid)
proof –
have ∀ a. k+1 ≤ a ∧ a < j → ~ occurs-on (t ! a) = q (is ?R)
proof (rule ccontr)
  assume ~ ?R
  then obtain a where k+1 ≤ a a < j and ocp: occurs-on (t ! a) = q
by blast
  have a < length t
  proof –
  have j < length t
  proof (rule ccontr)
    assume ~ j < length t
    then have S t j = S t (j+1)
    using assms(1) no-change-if-ge-length-t by auto
    then show False using snap-q by auto
  qed
  then show ?thesis using ⟨a < j⟩ by simp
qed
then show False
proof (cases regular-event (t ! a))
  case True
    have ~ has-snapshotted (S t a) q
    by (meson ⟨a < j⟩ assms(1) computation.snapshot-stable-ver-2
computation-axioms less-imp-le-nat snap-q(1))
    then have prerecording-event t a using ⟨a < length t⟩ ocp True
prerecording-event by simp
    then show False using ⟨k+1 ≤ a⟩ ⟨k ≥ i⟩ assms by auto
  next
  case False
    then have (S t a) ⊢ (t ! a) ⇔ (S t (a+1))
    using ⟨a < length t⟩ assms(1) step-Suc by auto
    then have has-snapshotted (S t (a+1)) q
    by (metis False ocp nonregular-event-induces-snapshot snap-
shot-state-unchanged)
    then show False
    by (metis Suc-eq-plus1 Suc-leI ⟨a < j⟩ assms(1) snap-q(1) snap-
shot-stable-ver-3)

```

```

    qed
  qed
  moreover have Marker : set (msgs (S t (k+1)) cid)
  using chan ⟨map Msg (fst (cs (S t k) cid)) @ takeWhile ((≠) Marker) (msgs
(S t k) cid) = map Msg (fst (cs (S t (k + 1)) cid)) @ takeWhile ((≠) Marker)
(msgs (S t (k + 1)) cid) ∧ snd (cs (S t (k + 1)) cid) = NotStarted⟩ assms(1)
cs-in-initial-state-implies-not-snapshotted marker-must-stay-if-no-snapshot snap-p(2)
by blast
  moreover have has-snapshotted (S t (k+1)) p
  using snap-p(2) by blast
  moreover have ~ has-snapshotted (S t (k+1)) q
  using chan ⟨map Msg (fst (cs (S t k) cid)) @ takeWhile ((≠) Marker)
(msgs (S t k) cid) = map Msg (fst (cs (S t (k + 1)) cid)) @ takeWhile ((≠)
Marker) (msgs (S t (k + 1)) cid) ∧ snd (cs (S t (k + 1)) cid) = NotStarted⟩
assms(1) cs-in-initial-state-implies-not-snapshotted by blast
  moreover have k+1 ≤ j
  using ⟨k < j⟩ by auto
  moreover have trace init t final using assms by simp
  moreover have snd (cs (S t (k+1)) cid) = NotStarted
  using ⟨map Msg (fst (cs (S t k) cid)) @ takeWhile ((≠) Marker) (msgs
(S t k) cid) = map Msg (fst (cs (S t (k + 1)) cid)) @ takeWhile ((≠) Marker)
(msgs (S t (k + 1)) cid) ∧ snd (cs (S t (k + 1)) cid) = NotStarted⟩ by blast
  ultimately show ?thesis using cs-when-recording-3 chan by simp
  qed
  ultimately show ?thesis by simp
next
case False
show ?thesis
proof -
  have has-snapshotted (S t i) p
  by (metis False Suc-eq-plus1 assms(1) not-less-eq-eq snap-p(2) snap-
shot-stable-ver-3)
  moreover have ~ has-snapshotted (S t i) q
  using nsq by auto
  moreover have Marker : set (msgs (S t i) cid)
  using chan assms(1) calculation(1) marker-must-stay-if-no-snapshot nsq
by blast
  moreover have ∀ k. i ≤ k ∧ k < j ⟶ ~ occurs-on (t ! k) = q (is ?R)
  proof (rule ccontr)
    assume ~ ?R
    then obtain k where i ≤ k k < j and ocp: occurs-on (t ! k) = q by
blast
  have k < length t
  proof -
    have j < length t
    proof (rule ccontr)
      assume ~ j < length t
      then have S t j = S t (j+1)
      using assms(1) no-change-if-ge-length-t by auto

```

```

      then show False using snap-q by auto
    qed
    then show ?thesis using  $\langle k < j \rangle$  by simp
  qed
  then show False
  proof (cases regular-event (t ! k))
    case True
      have  $\sim$  has-snapshotted (S t k) q
        by (meson  $\langle k < j \rangle$  assms(1) computation.snapshot-stable-ver-2
computation-axioms less-imp-le-nat snap-q(1))
        then have prerecording-event t k using  $\langle k < \text{length } t \rangle$  ocp True
prerecording-event by simp
        then show False using  $\langle i \leq j \rangle \langle k \geq i \rangle$  assms by auto
      next
        case False
        then have (S t k)  $\vdash$  (t ! k)  $\mapsto$  (S t (k+1))
          using  $\langle k < \text{length } t \rangle$  assms(1) step-Suc by auto
        then have has-snapshotted (S t (k+1)) q
          by (metis False ocp nonregular-event-induces-snapshot snapshot-state-unchanged)
        then show False
          by (metis Suc-eq-plus1 Suc-leI  $\langle k < j \rangle$  assms(1) snap-q(1) snapshot-stable-ver-3)
        qed
      qed
      ultimately show ?thesis using cs-when-recording-3
        using NotStarted assms(1) bound chan by auto
    qed
  qed
  moreover have map Msg (fst (cs (S t j) cid)) @ takeWhile (( $\neq$ ) Marker)
    (msgs (S t j) cid)
    = map Msg (fst (cs final cid))
  proof (cases  $\exists q p. t ! j = \text{RecvMarker } cid \ q \ p$ )
    case True
      then have fst (cs (S t j) cid) = fst (cs (S t (j+1)) cid)
        using step-q by auto
      moreover have RecvMarker: t ! j = RecvMarker cid q p
      proof –
        have can-occur (t ! j) (S t j) using happen-implies-can-occur step-q by
simp
        then show ?thesis
          using RecvMarker-given-channel True chan by force
        qed
      moreover have takeWhile (( $\neq$ ) Marker) (msgs (S t j) cid) = []
      proof –
        have can-occur (t ! j) (S t j) using happen-implies-can-occur step-q by
simp
        then have length (msgs (S t j) cid) > 0  $\wedge$  hd (msgs (S t j) cid) = Marker
          using RecvMarker can-occur-def by auto

```

```

    then show ?thesis
    by (metis (mono-tags, lifting) hd-conv-nth length-greater-0-conv nth-mem
set-takeWhileD takeWhile-nth)
  qed
  moreover have snd (cs (S t (j+1)) cid) = Done using step-q True by
auto
  moreover have cs (S t (j+1)) cid = cs final cid using chan calculation
cs-done-implies-same-snapshots assms(1)
  by (metis final-is-s-t-len-t nat-le-linear no-change-if-ge-length-t)
  ultimately show ?thesis
  by simp
next
case False
have ~ regular-event (t ! j)
using regular-event-preserves-process-snapshots snap-q(1) snap-q(2) step-q
by auto
then have isSnapshot (t ! j)  $\vee$  isRecvMarker (t ! j) using nonregular-event
by auto
then have map Msg (fst (cs (S t j) cid)) @ takeWhile (( $\neq$ ) Marker) (msgs
(S t j) cid)
= map Msg (fst (cs (S t (j+1)) cid)) @ takeWhile (( $\neq$ ) Marker)
(msgs (S t (j+1)) cid)
 $\wedge$  snd (cs (S t (j+1)) cid) = Recording
proof (elim disjE, goal-cases)
case 1
have Snapshot: t ! j = Snapshot q
using 1 oq by auto
then have msgs (S t j) cid = msgs (S t (j+1)) cid
using <p  $\neq$  q> step-q chan by auto
moreover have cs (S t (j+1)) cid = (fst (cs (S t j) cid), Recording)
using step-q Snapshot chan by simp
ultimately show ?thesis by simp
next
case 2
obtain cid' r where RecvMarker: t ! j = RecvMarker cid' q r
by (metis 2 event.collapse(5) oq)
have cid  $\neq$  cid'
proof (rule ccontr)
assume ~ cid  $\neq$  cid'
then have channel cid = channel cid' by simp
then have channel cid' = Some (r, q)
using False RecvMarker < $\neg$  cid  $\neq$  cid'> by blast
then show False
using False RecvMarker < $\neg$  cid  $\neq$  cid'> by blast
qed
then have msgs (S t j) cid = msgs (S t (j+1)) cid
using <cid  $\neq$  cid'> step-q snap-q RecvMarker chan <p  $\neq$  q> by simp
moreover have cs (S t (j+1)) cid = (fst (cs (S t j) cid), Recording)
using <p  $\neq$  q> <cid  $\neq$  cid'> step-q snap-q RecvMarker chan by auto

```

**ultimately show** *?thesis* **by simp**  
**qed**  
**moreover have**  $\text{map } \text{Msg } (\text{fst } (\text{cs } (S \ t \ (j+1)) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (S \ t \ (j+1)) \ \text{cid})$   
 $= \text{map } \text{Msg } (\text{fst } (\text{cs } \ \text{final} \ \text{cid}))$   
**proof** –  
**have**  $\text{snd } (\text{cs } (S \ t \ (j+1)) \ \text{cid}) = \text{Recording}$   
**using** *calculation by blast*  
**moreover have**  $\text{has-snapshotted } (S \ t \ (j+1)) \ p$   
**by** (*metis Suc-eq-plus1 Suc-leI <k < j> assms(1) le-add1 snap-p(2) snapshot-stable-ver-3*)  
**moreover have**  $\text{has-snapshotted } (S \ t \ (j+1)) \ q$  **using** *snap-q* **by auto**  
**moreover have**  $j < \text{length } t$   
**by** (*metis (no-types, lifting) chan Suc-eq-plus1 assms(1) cs-done cs-done-implies-both-snapshotted(2) computation.no-change-if-ge-length-t computation.snapshot-stable-ver-3 computation-axioms leI le-Suc-eq snap-q(1) snap-q(2)*)  
**ultimately show** *?thesis* **using** *cs-when-recording assms(1) cs-done final-is-s-t-len-t*  
**proof** –  
**assume**  $a1: j < \text{length } t$   
**assume**  $a2: \text{trace } \text{init } t \ \text{final}$   
**assume**  $a3: \text{snd } (\text{cs } (S \ t \ (\text{length } t)) \ \text{cid}) = \text{Done}$   
**assume**  $a4: \text{snd } (\text{cs } (S \ t \ (j + 1)) \ \text{cid}) = \text{Recording}$   
**assume**  $a5: \text{ps } (S \ t \ (j + 1)) \ p \neq \text{None}$   
**assume**  $a6: \bigwedge t. \text{trace } \text{init } t \ \text{final} \implies \text{final} = S \ t \ (\text{length } t)$   
**assume**  $a7: \bigwedge i \ j \ t \ p \ \text{cid} \ q. \llbracket i < j; j \leq \text{length } t; \text{trace } \text{init } t \ \text{final}; \text{ps } (S \ t \ i) \ p \neq \text{None}; \text{snd } (\text{cs } (S \ t \ i) \ \text{cid}) = \text{Recording}; \text{snd } (\text{cs } (S \ t \ j) \ \text{cid}) = \text{Done}; \text{channel } \text{cid} = \text{Some } (p, q) \rrbracket \implies \text{map } \text{Msg } (\text{fst } (\text{cs } (S \ t \ j) \ \text{cid})) = \text{map } \text{Msg } (\text{fst } (\text{cs } (S \ t \ i) \ \text{cid})) \ @ \ \text{takeWhile } ((\neq) \ \text{Marker}) \ (\text{msgs } (S \ t \ i) \ \text{cid})$   
**have**  $\text{Suc } j < \text{length } t$   
**using**  $a3 \ a2 \ a1$  **by** (*metis (no-types) False Suc-eq-plus1 Suc-lessI chan cs-done-implies-has-snapshotted done-only-from-recv-marker snap-q(1) step-q*)  
**then show** *?thesis*  
**using**  $a7 \ a6 \ a5 \ a4 \ a3 \ a2$  **by** (*metis (no-types) Suc-eq-plus1 chan nat-le-linear*)  
**qed**  
**qed**  
**ultimately show** *?thesis* **by simp**  
**qed**  
**ultimately show** *?thesis*  
**by** (*metis (no-types, lifting) Nil-is-map-conv assms(1) assms(3) chan cs-done cs-done-implies-has-snapshotted cs-not-nil-implies-postrecording-event nat-le-linear nsq self-append-conv2 snapshot-stable-ver-3*)  
**qed**  
**next**  
**case** *Recording*  
**then obtain**  $j$  **where**  $\text{snap-p}: \sim \text{has-snapshotted } (S \ t \ j) \ p \ \text{has-snapshotted } (S \ t \ (j+1)) \ p$   
**by** (*metis Suc-eq-plus1 assms(1) exists-snapshot-for-all-p*)

```

have snap-q: has-snapshotted (S t i) q
  using Recording assms(1) chan cs-recording-implies-snapshot by blast
have fst-cs-empty: cs (S t i) cid = [], Recording) (is ?P)
proof (rule ccontr)
  assume ~ ?P
  have snd (cs (S t i) cid) = Recording using Recording by simp
  moreover have fst (cs (S t i) cid) ≠ [] using ⟨~ ?P⟩ prod.collapse calculation
by metis
  ultimately have ∃j. j < i ∧ postrecording-event t j
    using assms(1) assms(4) chan cs-not-nil-implies-postrecording-event by
blast
  then show False using assms by auto
qed
then show ?thesis
proof –
  have i-less-len-t: i < length t
  proof (rule ccontr)
    assume ~ i < length t
    then have snd (cs (S t i) cid) = Done
    by (metis assms(1) cs-done le-eq-less-or-eq nat-le-linear no-change-if-ge-length-t)
    then show False using Recording by simp
  qed
  then have map Msg (fst (cs final cid))
    = map Msg (fst (cs (S t i) cid)) @ takeWhile ((≠) Marker) (msgs (S t i)
cid)
  proof (cases j < i)
    case True
      then have has-snapshotted (S t i) p
        by (metis Suc-eq-plus1 Suc-leI assms(1) snap-p(2) snapshot-stable-ver-3)
      moreover have length t ≤ length t by simp
      ultimately show ?thesis
        using Recording chan assms(1) cs-done cs-when-recording final-is-s-t-len-t
i-less-len-t by blast
    next
      case False

```

need to show that next message that comes into the channel must be marker

```

have ∀k. i ≤ k ∧ k < j → ~ occurs-on (t ! k) = p (is ?P)
proof (rule ccontr)
  assume ~ ?P
  then obtain k where i ≤ k < j occurs-on (t ! k) = p by blast
  then show False
  proof (cases regular-event (t ! k))
    case True
      then have prerecording-event t k
        by (metis (no-types, opaque-lifting) ⟨k < j⟩ ⟨occurs-on (t ! k) = p⟩
all-processes-snapshotted-in-final-state assms(1) final-is-s-t-len-t computation.prerecording-event
computation-axioms less-trans nat-le-linear not-less snap-p(1) snapshot-stable-ver-2)
      then show ?thesis using assms ⟨i ≤ k⟩ by auto

```

```

next
  case False
  then have step-k:  $(S\ t\ k) \vdash (t\ !\ k) \mapsto (S\ t\ (Suc\ k))$ 
  by (metis (no-types, lifting) Suc-leI  $\langle k < j \rangle$  all-processes-snapshotted-in-final-state
assms(1) final-is-s-t-len-t le-Suc-eq less-imp-Suc-add linorder-not-less no-change-if-ge-length-t
snap-p(1) step-Suc)
  then have has-snapshotted  $(S\ t\ (Suc\ k))\ p$ 
  by (metis False  $\langle occurs-on\ (t\ !\ k) = p \rangle$  nonregular-event-induces-snapshot
snapshot-state-unchanged)
  then have  $k \geq j$ 
  by (metis Suc-leI  $\langle k < j \rangle$  assms(1) snap-p(1) snapshot-stable-ver-3)
  then show False using  $\langle k < j \rangle$  by simp
qed
qed
moreover have  $\sim has-snapshotted\ (S\ t\ i)\ p$ 
using False assms(1) snap-p(1) snapshot-stable-ver-3 by auto
ultimately have to-snapshot:  $map\ Msg\ (fst\ (cs\ (S\ t\ j)\ cid))\ @\ takeWhile\ ((\neq)\ Marker)\ (msgs\ (S\ t\ j)\ cid)$ 
 $=\ map\ Msg\ (fst\ (cs\ (S\ t\ i)\ cid))\ @\ takeWhile\ ((\neq)\ Marker)\ (msgs\ (S\ t\ i)\ cid)$ 
using False chan Recording assms(1) cs-when-recording-2 by auto
have step-j:  $(S\ t\ j) \vdash (t\ !\ j) \mapsto (S\ t\ (j+1))$ 
by (metis Suc-eq-plus1 Suc-le-eq assms(1) distributed-system.step-Suc distributed-system-axioms
computation.no-change-if-ge-length-t computation-axioms le-add1 not-less-eq-eq snap-p(1) snap-p(2))
then have  $map\ Msg\ (fst\ (cs\ (S\ t\ j)\ cid))\ @\ takeWhile\ ((\neq)\ Marker)\ (msgs\ (S\ t\ j)\ cid)$ 
 $=\ map\ Msg\ (fst\ (cs\ (S\ t\ (j+1))\ cid))\ @\ takeWhile\ ((\neq)\ Marker)\ (msgs\ (S\ t\ (j+1))\ cid)$ 
proof –
have  $o: \sim regular-event\ (t\ !\ j) \wedge occurs-on\ (t\ !\ j) = p$ 
by (metis (no-types, opaque-lifting) distributed-system.no-state-change-if-no-event
distributed-system.regular-event-cannot-induce-snapshot distributed-system-axioms snap-p(1)
snap-p(2) step-j)
then show ?thesis
using chan snapshot-step-cs-preservation-p step-j by blast
qed
moreover have  $map\ Msg\ (fst\ (cs\ final\ cid))$ 
 $=\ map\ Msg\ (fst\ (cs\ (S\ t\ (j+1))\ cid))\ @\ takeWhile\ ((\neq)\ Marker)\ (msgs\ (S\ t\ (j+1))\ cid)$ 
proof –
have  $snd\ (cs\ (S\ t\ (j+1))\ cid) = Recording$ 
proof –
have  $f1: ps\ (S\ t\ j)\ p = None$ 
by (meson snap-p(1))
then have  $f2: j < length\ t$ 
by (metis (no-types) all-processes-snapshotted-in-final-state assms(1)
final-is-s-t-len-t linorder-not-le snapshot-stable-ver-3)
have  $t\ !\ j \neq RecoMarker\ cid\ q\ p$ 

```

```

    using f1 by (metis (no-types) Suc-eq-plus1 assms(1) recv-marker-means-snapshotted
step-j)
    then show ?thesis
    using f2 f1 by (meson False assms(1) chan cs-done-implies-both-snapshotted(1)
cs-in-initial-state-implies-not-snapshotted cs-not-not-started-stable done-only-from-recv-marker
linorder-not-le recording-state.exhaust snap-q snapshot-stable-ver-3 step-j)
    qed
    moreover have  $j+1 < \text{length } t$ 
    proof (rule ccontr)
    assume  $\sim j+1 < \text{length } t$ 
    then have  $\text{snd } (cs (S t (j+1)) \text{ cid}) = \text{Done}$ 
    by (metis assms(1) cs-done le-Suc-eq less-imp-Suc-add linorder-not-le
no-change-if-ge-length-t)
    then show False using calculation by auto
    qed
    ultimately show ?thesis
    using chan snap-p(2) assms final-is-s-t-len-t cs-when-recording cs-done
by blast
    qed
    ultimately show ?thesis using to-snapshot by simp
    qed
    then show ?thesis using fst-cs-empty by simp
    qed
next
case Done

```

msgs must be empty, and cs must also be empty

```

    have fst-cs-empty:  $\text{fst } (cs (S t i) \text{ cid}) = []$ 
    proof (rule ccontr)
    assume  $\sim \text{fst } (cs (S t i) \text{ cid}) = []$ 
    then have  $\text{fst } (cs (S t 0) \text{ cid}) \neq \text{fst } (cs (S t i) \text{ cid})$ 
    by (metis chan assms(1) cs-not-nil-implies-postrecording-event gr-implies-not0
le0)
    then have  $\exists j. j < i \wedge \text{postrecording-event } t j$ 
    using chan  $\langle \text{fst } (cs (S t i) \text{ cid}) \neq [] \rangle$  assms(1) assms(4) cs-not-nil-implies-postrecording-event
by blast
    then show False using assms by auto
    qed
    moreover have  $\text{msgs } (S t i) \text{ cid} = []$ 
    proof -
    have no-marker:  $\text{Marker} \notin \text{set } (\text{msgs } (S t i) \text{ cid})$  (is ?P)
    proof (rule ccontr)
    assume  $\sim ?P$ 
    then have  $\text{Marker} : \text{set } (\text{msgs } (S t i) \text{ cid})$  by simp
    then have  $\text{snd } (cs (S t i) \text{ cid}) \neq \text{Done}$ 
    by (metis Marker-in-channel-implies-not-done chan assms(1) nat-le-linear
s-def take-all)
    then show False using Done by simp
    qed

```

**have** *snap-both*: *has-snapshotted* ( $S\ t\ i$ )  $p \wedge$  *has-snapshotted* ( $S\ t\ i$ )  $q$   
**by** (*metis chan Done assms(1) cs-done-implies-both-snapshotted(1) cs-done-implies-has-snapshotted*  
*final-is-s-t-len-t computation.all-processes-snapshotted-in-final-state computation-axioms*  
*le-refl not-less s-def take-all*)  
**obtain**  $j$  **where** *snap-p*:  $\sim$  *has-snapshotted* ( $S\ t\ j$ )  $p$  *has-snapshotted* ( $S\ t$   
 $(j+1)$ )  $p$   
**by** (*metis Suc-eq-plus1 assms(1) exists-snapshot-for-all-p*)  
**have**  $j < i$   
**by** (*meson assms(1) not-le-imp-less snap-both snap-p(1) snapshot-stable-ver-2*)  
**have** *step-j*: ( $S\ t\ j$ )  $\vdash$  ( $t\ !\ j$ )  $\mapsto$  ( $S\ t\ (j+1)$ )  
**by** (*metis Suc-eq-plus1 assms(1) distributed-system.step-Suc distributed-system-axioms*  
*computation.no-change-if-ge-length-t computation-axioms le-add1 linorder-not-less*  
*snap-p(1) snap-p(2)*)  
**have** *nonreg-j*:  $\sim$  *regular-event* ( $t\ !\ j$ )  
**by** (*meson distributed-system.regular-event-cannot-induce-snapshot dis-*  
*tributed-system-axioms snap-p(1) snap-p(2) step-j*)  
**have** *oc-j*: *occurs-on* ( $t\ !\ j$ ) =  $p$   
**using** *no-state-change-if-no-event snap-p(1) snap-p(2) step-j* **by** *force*  
**have** *msgs* ( $S\ t\ i$ ) *cid* =  $\square \vee$  (*msgs* ( $S\ t\ i$ ) *cid*  $\neq$   $\square \wedge$  *last* (*msgs* ( $S\ t\ i$ ) *cid*)  
= *Marker*)  
**proof** –  
**have** *msgs* ( $S\ t\ (j+1)$ ) *cid*  $\neq$   $\square \wedge$  *last* (*msgs* ( $S\ t\ (j+1)$ ) *cid*) = *Marker*  
**proof** –  
**have** *msgs* ( $S\ t\ (j+1)$ ) *cid* = *msgs* ( $S\ t\ j$ ) *cid* @ [*Marker*]  
**proof** –  
**have** *isSnapshot* ( $t\ !\ j$ )  $\vee$  *isRecvMarker* ( $t\ !\ j$ ) **using** *nonregular-event*  
*nonreg-j* **by** *blast*  
**then show** *?thesis*  
**proof** (*elim disjE, goal-cases*)  
**case** 1  
**then have**  $t\ !\ j =$  *Snapshot*  $p$  **using** *oc-j* **by** *auto*  
**then show** *?thesis* **using** *step-j chan* **by** *auto*  
**next**  
**case** 2  
**then obtain**  $cid'\ r$  **where** *RecvMarker*:  $t\ !\ j =$  *RecvMarker*  $cid'\ p\ r$   
**by** (*metis event.collapse(5) oc-j*)  
**have**  $cid \neq cid'$   
**proof** (*rule ccontr*)  
**assume**  $\sim cid \neq cid'$   
**then have** *channel*  $cid =$  *channel*  $cid'$  **by** *auto*  
**then have** *Some* ( $p, q$ ) = *Some* ( $r, p$ )  
**by** (*metis RecvMarker RecvMarker-implies-Marker-in-set assms(1)*  
*chan computation.no-marker-if-no-snapshot computation-axioms snap-p(1) step-j*)  
**then show** *False* **using** *no-self-channel chan* **by** *simp*  
**qed**  
**then show** *?thesis* **using** *oc-j snap-p step-j chan RecvMarker* **by** *auto*  
**qed**  
**qed**  
**then show** *?thesis* **by** *auto*

**qed**  
**moreover have**  $i \leq \text{length } t$  **using** *assms* **by** *simp*  
**moreover have**  $j+1 \leq i$  **using**  $\langle j < i \rangle$  **by** *simp*  
**moreover have**  $\forall k. j+1 \leq k \wedge k < i \wedge \text{regular-event } (t ! k) \longrightarrow \sim \text{occurs-on}$   
 $(t ! k) = p$  **(is ?R)**  
**proof** (*rule ccontr*)  
**assume**  $\sim ?R$   
**then obtain**  $k$  **where** *range:  $j+1 \leq k < i$  and regular-event  $(t ! k)$*   
*occurs-on  $(t ! k) = p$*   
**by** *blast*  
**then have** *postrecording-event  $t k$  using snap-p*  
**by** (*meson assms(1) calculation(2) le-trans linorder-not-less pre-if-regular-and-not-post*  
*prerecording-event snapshot-stable-ver-2*)  
**then show** *False* **using** *assms range* **by** *auto*  
**qed**  
**ultimately show** *?thesis*  
**using** *assms(1) chan last-unchanged-or-empty-if-no-events snap-p(2)* **by**  
*auto*  
**qed**  
**then show** *?thesis* **using** *no-marker last-in-set* **by** *fastforce*  
**qed**  
**ultimately show** *?thesis*  
**using** *chan Done assms(1) assms(4) final-is-s-t-len-t computation.cs-done-implies-same-snapshots*  
*computation-axioms* **by** *fastforce*  
**qed**  
**ultimately show** *filter (( $\neq$ ) Marker) (msgs (S t i) cid) = map Msg (fst (cs final*  
*cid))* **by** *simp*  
**qed**

**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*  
**proof** (*unfold state-equal-to-snapshot-def, rule conjI*)  
**show** *ps-equal-to-snapshot (S t i) final*  
**using** *assms ps-after-all-prerecording-events* **by** *auto*  
**show** *cs-equal-to-snapshot (S t i) final*  
**using** *assms cs-after-all-prerecording-events* **by** *auto*  
**qed**

## 5.4 Obtaining the desired traces

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

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

**definition** *count-violations* :: ('a, 'b, 'c) trace  $\Rightarrow$  nat **where**

$$\begin{aligned} \text{count-violations } t = & \text{sum } (\lambda i. \text{if } \text{postrecording-event } t \ i \\ & \text{then } \text{card } \{j \in \{i+1..<\text{length } t\}. \text{prerecording-event } t \ j\} \\ & \text{else } 0) \\ & \{0..<\text{length } t\} \end{aligned}$$

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

**shows**

$$\begin{aligned} & (\text{if } \text{postrecording-event } t \ i \\ & \text{then } \text{card } \{j \in \{i+1..<\text{length } t\}. \text{prerecording-event } t \ j\} \\ & \text{else } 0) \geq 0 \end{aligned}$$

**by** *simp*

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

**shows**

$$\text{count-violations } t \geq 0$$

**by** *simp*

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

**assumes**

$$\text{count-violations } t = 0$$

**shows**

$$\begin{aligned} \forall i \in \{0..<\text{length } t\}. & (\text{if } \text{postrecording-event } t \ i \\ & \text{then } \text{card } \{j \in \{i+1..<\text{length } t\}. \text{prerecording-event } t \ j\} \\ & \text{else } 0) = 0 \end{aligned}$$

**proof** –

**have**  $\text{sum } (\lambda i. \text{if } \text{postrecording-event } t \ i$

$$\text{then } \text{card } \{j \in \{i+1..<\text{length } t\}. \text{prerecording-event } t \ j\}$$

$$\text{else } 0)$$

$$\{0..<\text{length } t\} = 0 \text{ using } \text{count-violations-def } \text{assms } \text{by } \text{simp}$$

**then show**  $\forall i \in \{0..<\text{length } t\}. (\text{if } \text{postrecording-event } t \ i$

$$\text{then } \text{card } \{j \in \{i+1..<\text{length } t\}. \text{prerecording-event } t \ j\}$$

$$\text{else } 0) = 0$$

**by** *auto*

**qed**

**lemma** *exists-postrecording-violation-if-count-greater-0*:

**assumes**

$$\text{count-violations } t > 0$$

**shows**

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

**proof** (rule *ccontr*)

**assume**  $\sim ?P$

**then have**  $\forall i. \sim \text{postrecording-event } t \ i \vee \text{card } \{j \in \{i+1..<\text{length } t\}. \text{prere-$

```

ording-event t j = 0 by simp
have count-violations t = 0
proof (unfold count-violations-def)
  have  $\forall i. (if\ postrecording-event\ t\ i$ 
     $then\ card\ \{j \in \{i+1..<length\ t\}. prerecording-event\ t\ j\}$ 
     $else\ 0) = 0$ 
    using  $\langle \forall i. \neg\ postrecording-event\ t\ i \vee card\ \{j \in \{i + 1..<length\ t\}. prere-$ 
ording-event t j = 0  $\rangle$  by auto
  then show  $sum\ (\lambda i. if\ postrecording-event\ t\ i$ 
     $then\ card\ \{j \in \{i+1..<length\ t\}. prerecording-event\ t\ j\}$ 
     $else\ 0)\ \{0..<length\ t\} = 0$  by simp

qed
then show False using assms by simp
qed

```

```

lemma exists-prerecording-violation-when-card-greater-0:
assumes
   $card\ \{j \in \{i+1..<length\ t\}. prerecording-event\ t\ j\} > 0$ 
shows
   $\exists j \in \{i+1..<length\ t\}. prerecording-event\ t\ j$ 
by (metis (no-types, lifting) Collect-empty-eq assms card-0-eq empty-subsetI fi-
nite-atLeastLessThan not-gr-zero subset-card-intvl-is-intvl)

```

```

lemma card-greater-0-if-post-after-pre:
assumes
   $i < j$  and
  postrecording-event t i and
  prerecording-event t j
shows
   $card\ \{j \in \{i+1..<length\ t\}. prerecording-event\ t\ j\} > 0$ 
proof -
  have  $j < length\ t$  using prerecording-event assms by auto
  have  $\{j \in \{i+1..<length\ t\}. prerecording-event\ t\ j\} \neq empty$ 
    using Suc-eq-plus1  $\langle j < length\ t \rangle$  assms(1) assms(3) less-imp-triv by auto
  then show ?thesis by fastforce
qed

```

```

lemma exists-neighboring-violation-pair:
assumes
  trace init t final and
  count-violations t > 0
shows
   $\exists i\ j. i < j \wedge postrecording-event\ t\ i \wedge prerecording-event\ t\ j$ 
   $\wedge (\forall k. (i < k \wedge k < j) \longrightarrow \sim\ regular-event\ (t\ !\ k)) \wedge j < length\ t$ 
proof -
  let  $?I = \{i. postrecording-event\ t\ i \wedge card\ \{j \in \{i+1..<length\ t\}. prerecord-$ 
ing-event t j  $> 0\}$ 
  have nonempty-I:  $?I \neq empty$  using assms exists-postrecording-violation-if-count-greater-0
by blast

```

```

have fin-I: finite ?I
proof (rule ccontr)
  assume ~ finite ?I
  then obtain i where i > length t postrecording-event t i
    by (simp add: postrecording-event)
  then show False using postrecording-event by simp
qed
let ?i = Max ?I
have no-greater-postrec-violation:  $\forall i. i > ?i \longrightarrow \sim$  (postrecording-event t i  $\wedge$ 
card {j  $\in$  {i+1..\in {?i+1..\in ?I
    using Max-in fin-I nonempty-I by blast
  then show ?thesis by simp
qed
let ?J = {j  $\in$  {?i+1..\neq empty
  using <card {j  $\in$  {?i+1..\forall j \in {?i+1..\longrightarrow \sim prere-
cording-event t j
  using Min-less-iff fin-J by blast
have j-less-len-t: ?j < length t
  using pre-j prerecording-event by blast
have  $\forall k. (?i < k \wedge k < ?j) \longrightarrow \sim$  regular-event (t ! k)
proof (rule allI, rule impI)
  fix k
  assume asm: ?i < k  $\wedge$  k < ?j
  then show  $\sim$  regular-event (t ! k)
  proof -
    have 0-le-k: 0  $\leq$  k by simp
    have k-less-len-t: k < length t using j-less-len-t asm by auto
    show ?thesis
  proof (rule ccontr)
    assume reg-event:  $\sim \sim$  regular-event (t ! k)
    then show False
  proof (cases has-snapshotted (S t k) (occurs-on (t ! k)))
    case True
      then have post-k: postrecording-event t k using reg-event k-less-len-t
postrecording-event by simp
      moreover have card {j  $\in$  {k+1..

```

using *post-k pre-j card-greater-0-if-post-after-pre asm pre-j* by *blast*  
 ultimately show *False* using *no-greater-postrec-violation asm* by *blast*  
 next  
 case *False*  
 then have *pre-k: prerecording-event t k* using *reg-event k-less-len-t*  
*prerecording-event* by *simp*  
 moreover have  $k \in \{?i+1..<length\ t\}$  using *asm k-less-len-t* by *simp*  
 ultimately show *False* using *no-smaller-prerec-violation asm* by *blast*  
 qed  
 qed  
 qed  
 moreover have  $?i < ?j$  using *nonempty-J* by *auto*  
 ultimately show *?thesis* using *pre-j post-i j-less-len-t* by *blast*  
 qed

**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  
    $\forall k. (i < k \wedge k < j) \longrightarrow \sim\ regular-event\ (t\ !\ k)$  and  
   *trace init t final*  
 shows  
    $\{k \in \{0..<i\}. prerecording-event\ t\ k\}$   
    $= \{k \in \{0..<i\}. prerecording-event\ (swap-events\ i\ j\ t)\ k\}$   
 proof –  
   let  $?t = swap-events\ i\ j\ t$   
   have *same-begin: take i t = take i ?t* using *swap-identical-heads assms* by *blast*  
   have *same-length: length t = length (swap-events i j t)* using *swap-identical-length*  
   *assms* by *blast*  
   have  $a: \forall k. k < i \longrightarrow t\ !\ k = ?t\ !\ k$   
   by (*metis nth-take same-begin*)  
   then have  $\forall k. k < i \longrightarrow prerecording-event\ t\ k = prerecording-event\ ?t\ k$   
   proof –  
     have  $\forall k. k < i \longrightarrow S\ t\ k = S\ ?t\ k$  using *assms swap-events* by *simp*  
     then show *?thesis* using *unfolding prerecording-event* using *a same-length* by  
     *presburger*  
   qed  
   then show *?thesis* by *auto*  
 qed

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

assumes  
   *prerecording-event t j* and  
   *postrecording-event t i* and  
    $i < j$  and

$j < \text{length } t$  **and**  
 $\text{count-violations } t = \text{Suc } n$  **and**  
 $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t ! k)$  **and**  
 $\text{trace init } t \text{ final}$   
**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** –  
**let**  $?t = \text{swap-events } i j t$   
**have**  $\text{card } \{k \in \{i..<j+1\}. \text{prerecording-event } t k\} = 1$   
**proof** –  
**have**  $\forall k. i \leq k \wedge k < j \longrightarrow \sim \text{prerecording-event } t k$   
**proof** (*rule allI, rule impI*)  
**fix**  $k$   
**assume** *asm*:  $i \leq k \wedge k < j$   
**then show**  $\sim \text{prerecording-event } t k$   
**proof** (*cases*  $k = i$ )  
**case** *True*  
**then have**  $\text{postrecording-event } t k$  **using** *assms* **by** *simp*  
**then show** *?thesis*  
**by** (*meson computation.postrecording-event computation.prerecording-event computation-axioms*)  
**next**  
**case** *False*  
**then have**  $i < k \wedge k < j$  **using** *asm* **by** *force*  
**then have**  $\sim \text{regular-event } (t ! k)$  **using** *assms* **by** *simp*  
**then show** *?thesis* **unfolding**  $\text{prerecording-event}$  **by** *simp*  
**qed**  
**qed**  
**then have**  $\{k \in \{i..<j\}. \text{prerecording-event } t k\} = \text{empty}$  **by** *simp*  
**moreover have**  $\{k \in \{j..<j+1\}. \text{prerecording-event } t k\} = \{j\}$   
**proof** –  
**have**  $\{j..<j+1\} = \{j\}$  **by** *auto*  
**moreover have**  $\text{prerecording-event } t j$  **using** *assms* **by** *simp*  
**ultimately show** *?thesis* **by** *blast*  
**qed**  
**ultimately have**  $\{k \in \{i..<j+1\}. \text{prerecording-event } t k\} = \{j\}$  **using** *assms*(*3-4*)  
**by** *auto*  
**then show** *?thesis* **by** *simp*  
**qed**  
**moreover have**  $\text{card } \{k \in \{i..<j+1\}. \text{prerecording-event } ?t k\} = 1$   
**proof** –  
**have** *swap-ind*:  $\text{prerecording-event } ?t i$   
 $\wedge \text{postrecording-event } ?t (i+1)$   
 $\wedge (\forall k. k > i+1 \wedge k < j+1 \longrightarrow \sim \text{regular-event } (?t ! k))$   
**using** *assms*  $\text{swap-events}$  **by** *blast*  
**have**  $\forall k. i+1 \leq k \wedge k < j+1 \longrightarrow \sim \text{prerecording-event } ?t k$   
**proof** (*rule allI, rule impI*)  
**fix**  $k$

```

assume asm:  $i+1 \leq k \wedge k < j+1$ 
then show  $\sim$  prerecording-event ?t k
proof (cases  $k = i+1$ )
  case True
    then have postrecording-event ?t k using swap-ind by blast
    then show ?thesis
      by (meson computation.postrecording-event computation.prerecording-event
computation-axioms)
  next
    case False
      then have  $i+1 < k \wedge k < j+1$  using asm by linarith
      then have  $\sim$  regular-event (?t ! k) using asm assms swap-ind by blast
      then show ?thesis unfolding prerecording-event by simp
  qed
qed
then have  $\{k \in \{i+1..<j+1\}. \text{prerecording-event } ?t\ k\} = \text{empty}$  by simp
moreover have  $\{k \in \{i..<i+1\}. \text{prerecording-event } ?t\ k\} = \{i\}$ 
proof –
  have  $\{i..<i+1\} = \{i\}$  by simp
  moreover have prerecording-event ?t i using swap-ind by blast
  ultimately show ?thesis by blast
qed
  ultimately have  $\{k \in \{i..<j+1\}. \text{prerecording-event } ?t\ k\} = \{i\}$  using
assms( $\beta-4$ ) by auto
  then show ?thesis by simp
qed
ultimately show ?thesis by simp
qed

```

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

```

assumes
  prerecording-event t j and
  postrecording-event t i and
   $i < j$  and
   $j < \text{length } t$  and
  count-violations t = Suc n and
   $\forall k. (i < k \wedge k < j) \longrightarrow \sim$  regular-event (t ! k) and
  trace init t final
shows
   $\{k \in \{j+1..<\text{length } t\}. \text{prerecording-event } t\ k\}$ 
  =  $\{k \in \{j+1..<\text{length } (\text{swap-events } i\ j\ t)\}. \text{prerecording-event } (\text{swap-events } i\ j\ t)\ k\}$ 
proof –
  let ?t = swap-events i j t
  have drop (j+1) t = drop (j+1) ?t
  using assms( $\beta$ ) assms( $\gamma$ ) swap-identical-tails by blast
  have same-length: length t = length ?t using swap-identical-length assms by
blast
  have a:  $\forall k. j+1 \leq k \wedge k < \text{length } t \longrightarrow ?t ! k = t ! k$ 

```

**proof** (*rule allI, rule impI*)  
**fix**  $k$   
**assume**  $j+1 \leq k \wedge k < \text{length } t$   
**then have**  $?t ! k = \text{drop } (j+1) (\text{swap-events } i \ j \ t) ! (k-(j+1))$   
**by** (*metis (no-types, lifting) Suc-eq-plus1 Suc-leI assms(4) le-add-diff-inverse nth-drop same-length*)  
**moreover have**  $t ! k = \text{drop } (j+1) t ! (k-(j+1))$   
**using**  $\langle j+1 \leq k \wedge k < \text{length } t \rangle$  **by** *auto*  
**ultimately have**  $\text{drop } (j+1) ?t ! (k-(j+1)) = \text{drop } (j+1) t ! (k-(j+1))$   
**using** *assms swap-identical-tails by metis*  
**then show**  $?t ! k = t ! k$   
**using**  $\langle ?t ! k = \text{drop } (j+1) ?t ! (k-(j+1)) \rangle \langle t ! k = \text{drop } (j+1) t ! (k-(j+1)) \rangle$  **by** *auto*  
**qed**  
**then have**  $\forall k. j+1 \leq k \wedge k < \text{length } t \longrightarrow \text{prerecording-event } t \ k = \text{prerecording-event } ?t \ k$   
**proof** –  
**have**  $\forall k. k \geq (j+1) \longrightarrow S \ t \ k = S \ ?t \ k$  **using** *assms swap-events by simp*  
**then show** *?thesis unfolding prerecording-event using a by auto*  
**qed**  
**then have**  $\{k \in \{j+1..<\text{length } t\}. \text{prerecording-event } t \ k\} = \{k \in \{j+1..<\text{length } t\}. \text{prerecording-event } ?t \ k\}$   
**by** *auto*  
**then show** *?thesis using same-length by metis*  
**qed**

**lemma** *card-ip1-to-j-is-1-in-normal-events:*

**assumes**  
*prerecording-event*  $t \ j$  **and**  
*postrecording-event*  $t \ i$  **and**  
 $i < j$  **and**  
 $j < \text{length } t$  **and**  
*count-violations*  $t = \text{Suc } n$  **and**  
 $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t ! k)$  **and**  
*count-violations*  $t = \text{Suc } n$  **and**  
*trace* *init*  $t$  *final*  
**shows**  
 $\text{card } \{k \in \{i+1..<j+1\}. \text{prerecording-event } t \ k\} = 1$   
**proof** –  
**have**  $\forall k. i < k \wedge k < j \longrightarrow \sim \text{prerecording-event } t \ k$   
**proof** (*rule allI, rule impI*)  
**fix**  $k$   
**assume** *asm*:  $i < k \wedge k < j$   
**then show**  $\sim \text{prerecording-event } t \ k$   
**proof** –  
**have**  $\sim \text{regular-event } (t ! k)$  **using** *asm assms by blast*  
**then show** *?thesis unfolding prerecording-event by simp*  
**qed**  
**qed**

**then have**  $\{k \in \{i+1..<j\}. \text{prerecording-event } t \ k\} = \text{empty}$  **by auto**  
**moreover have**  $\{k \in \{j..<j+1\}. \text{prerecording-event } t \ k\} = \{j\}$   
**proof** –  
    **have**  $\{j..<j+1\} = \{j\}$  **by auto**  
    **moreover have**  $\text{prerecording-event } t \ j$  **using** *assms* **by simp**  
    **then show** *?thesis* **by auto**  
**qed**  
**ultimately have**  $\{k \in \{i+1..<j+1\}. \text{prerecording-event } t \ k\} = \{j\}$  **using** *assms*  
**by auto**  
    **then show** *?thesis* **by simp**  
**qed**

**lemma** *card-ip1-to-j-is-0-in-swapped-events*:  
**assumes**  
    *prerecording-event*  $t \ j$  **and**  
    *postrecording-event*  $t \ i$  **and**  
     $i < j$  **and**  
     $j < \text{length } t$  **and**  
    *count-violations*  $t = \text{Suc } n$  **and**  
     $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t \ ! \ k)$  **and**  
    *count-violations*  $t = \text{Suc } n$  **and**  
    *trace init*  $t \ \text{final}$   
**shows**  
     $\text{card } \{k \in \{i+1..<j+1\}. \text{prerecording-event } (\text{swap-events } i \ j \ t) \ k\} = 0$   
**proof** –  
    **let**  $?t = \text{swap-events } i \ j \ t$   
    **have** *postrec-ip1*: *postrecording-event*  $?t \ (i+1)$  **using** *assms* *swap-events* **by blast**  
    **have** *neigh-shift*:  $\forall k. i+1 < k \wedge k < j+1 \longrightarrow \sim \text{regular-event } (?t \ ! \ k)$  **using**  
*assms* *swap-events* **by blast**  
    **have**  $\forall k. i+1 \leq k \wedge k < j+1 \longrightarrow \sim \text{prerecording-event } ?t \ k$   
    **proof** (*rule allI*, *rule impI*)  
        **fix**  $k$   
        **assume** *asm*:  $i+1 \leq k \wedge k < j+1$   
        **then show**  $\sim \text{prerecording-event } ?t \ k$   
        **proof** (*cases*  $k = i+1$ )  
            **case** *True*  
            **then show** *?thesis* **using** *postrec-ip1*  
            **by** (*meson* *computation.postrecording-event computation.prerecording-event*  
*computation-axioms*)  
            **next**  
            **case** *False*  
            **then have**  $i+1 < k \wedge k < j+1$  **using** *asm* **by simp**  
            **then have**  $\sim \text{regular-event } (?t \ ! \ k)$  **using** *neigh-shift* **by blast**  
            **then show** *?thesis* **unfolding** *prerecording-event* **by simp**  
        **qed**  
    **qed**  
    **then have**  $\{k \in \{i+1..<j+1\}. \text{prerecording-event } ?t \ k\} = \text{empty}$  **by auto**  
    **then show** *?thesis* **by simp**  
**qed**

**lemma** *count-violations-swap*:

**assumes**

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

**shows**

*count-violations* (*swap-events*  $i\ j\ t$ ) =  $n$

**proof** –

**let**  $?t = \text{swap-events } i\ j\ t$

**let**  $?f = (\lambda i. \text{if } \text{postrecording-event } t\ i \text{ then } \text{card } \{j \in \{i+1..<\text{length } t\}. \text{prerecording-event } t\ j\} \text{ else } 0)$

**let**  $?f' = (\lambda i. \text{if } \text{postrecording-event } ?t\ i \text{ then } \text{card } \{j \in \{i+1..<\text{length } ?t\}. \text{prerecording-event } ?t\ j\} \text{ else } 0)$

**have** *same-postrec-prefix*:  $\forall k. k < i \longrightarrow \text{postrecording-event } t\ k = \text{postrecording-event } ?t\ k$

**proof** –

**have**  $\forall k. k < i \longrightarrow S\ t\ k = S\ ?t\ k$  **using** *assms* *swap-events* **by** *auto*

**then show** *thesis* **unfolding** *postrecording-event*

**proof** –

**assume**  $a1: \forall k < i. S\ t\ k = S\ (\text{swap-events } i\ j\ t)\ k$

**{ fix**  $nn :: \text{nat}$

**have**  $\bigwedge n\ na\ es\ nb. \neg n < na \vee \neg na < \text{length } es \vee \neg nb < n \vee \text{swap-events } n\ na\ es\ !\ nb = (es\ !\ nb :: ('a, 'b, 'c)\ \text{event})$

**by** (*metis* (*no-types*) *nth-take* *swap-identical-heads*)

**then have**  $\neg nn < i \vee \neg nn < \text{length } t \wedge \neg nn < \text{length } (\text{swap-events } i\ j\ t) \vee \neg \text{regular-event } (t\ !\ nn) \wedge \neg \text{regular-event } (\text{swap-events } i\ j\ t\ !\ nn) \vee ps\ (S\ t\ nn)\ (\text{occurs-on } (t\ !\ nn)) = \text{None} \wedge ps\ (S\ (\text{swap-events } i\ j\ t)\ nn)\ (\text{occurs-on } (\text{swap-events } i\ j\ t\ !\ nn)) = \text{None} \vee \text{regular-event } (t\ !\ nn) \wedge \text{regular-event } (\text{swap-events } i\ j\ t\ !\ nn) \wedge nn < \text{length } t \wedge nn < \text{length } (\text{swap-events } i\ j\ t) \wedge ps\ (S\ t\ nn)\ (\text{occurs-on } (t\ !\ nn)) \neq \text{None} \wedge ps\ (S\ (\text{swap-events } i\ j\ t)\ nn)\ (\text{occurs-on } (\text{swap-events } i\ j\ t\ !\ nn)) \neq \text{None}$

**using**  $a1$  **by** (*metis* (*no-types*) *assms*(3) *assms*(4) *swap-identical-length*)

**}**

**then show**  $\forall n < i. (n < \text{length } t \wedge \text{regular-event } (t\ !\ n) \wedge ps\ (S\ t\ n)\ (\text{occurs-on } (t\ !\ n)) \neq \text{None}) = (n < \text{length } (\text{swap-events } i\ j\ t) \wedge \text{regular-event } (\text{swap-events } i\ j\ t\ !\ n) \wedge ps\ (S\ (\text{swap-events } i\ j\ t)\ n)\ (\text{occurs-on } (\text{swap-events } i\ j\ t\ !\ n)) \neq \text{None})$

**by** (*metis* (*no-types*))

**qed**

**qed**

**have** *same-postrec-suffix*:  $\forall k. k \geq j+1 \longrightarrow \text{postrecording-event } t\ k = \text{postrecording-event } ?t\ k$

**proof** –

**have** *post-equal-states*:  $\forall k. k \geq j+1 \longrightarrow S\ t\ k = S\ ?t\ k$  **using** *assms* *swap-events*

```

by presburger
show ?thesis
proof (rule allI, rule impI)
  fix k
  assume  $j+1 \leq k$ 
  then show  $\text{postrecording-event } t \ k = \text{postrecording-event } ?t \ k$ 
  proof (cases  $k < \text{length } t$ )
    case False
    then have  $\sim \text{postrecording-event } t \ k$  using  $\text{postrecording-event}$  by simp
    moreover have  $\sim \text{postrecording-event } ?t \ k$ 
      using  $\text{postrecording-event swap-identical-length False}$  assms by metis
    ultimately show ?thesis by simp
  next
  case True
  then show  $\text{postrecording-event } t \ k = \text{postrecording-event } ?t \ k$ 
    using  $\text{post-equal-states}$ 
  proof -
    assume  $a1: \forall k \geq j + 1. S \ t \ k = S \ (\text{swap-events } i \ j \ t) \ k$ 
    assume  $a2: k < \text{length } t$ 
    have  $f3: \text{length } t = \text{length } (\text{swap-events } i \ j \ t)$ 
      using  $\text{assms}(3) \ \text{assms}(4) \ \text{swap-identical-length}$  by blast
    have  $f4: k - (j + 1) + (j + 1) = k$ 
      using  $\langle j + 1 \leq k \rangle \ \text{le-add-diff-inverse2}$  by blast
    have  $\text{drop } (j + 1) \ t = \text{drop } (j + 1) \ (\text{swap-events } i \ j \ t)$ 
      using  $\text{assms}(3) \ \text{assms}(4) \ \text{swap-identical-tails}$  by blast
    then have  $\text{swap-events } i \ j \ t ! k = t ! k$ 
      using  $f4 \ f3 \ a2$  by (metis (no-types)  $\text{drop-drop hd-drop-conv-nth}$ )
    then show ?thesis
      using  $f3 \ a1 \ \langle j + 1 \leq k \rangle \ \text{postrecording-event}$  by presburger
  qed
qed
qed
qed

```

**have**  $\text{sum-decomp-g}: \langle \text{sum } g \ \{0..<\text{length } t\} = \text{sum } g \ \{0..<i\} + \text{sum } g \ \{i..<j+1\} + \text{sum } g \ \{j+1..<\text{length } t\} \rangle$   
**for**  $g :: \langle \text{nat} \Rightarrow \text{nat} \rangle$   
**using**  $\text{sum.atLeastLessThan-concat} \ [\text{of } 0 \ i \ \langle j + 1 \rangle \ g] \ \text{sum.atLeastLessThan-concat} \ [\text{of } 0 \ \langle j + 1 \rangle \ \langle \text{length } t \rangle \ g] \ \text{assms}$   
**by**  $\text{simp}$   
**from**  $\text{sum-decomp-g} \ [\text{of } ?f]$   
**have**  $\text{sum-decomp-f}: \langle \text{sum } ?f \ \{0..<\text{length } t\} = \text{sum } ?f \ \{0..<i\} + \text{sum } ?f \ \{i..<j+1\} + \text{sum } ?f \ \{j+1..<\text{length } t\} \rangle .$   
**from**  $\text{sum-decomp-g} \ [\text{of } ?f']$   
**have**  $\text{sum-decomp-f}': \langle \text{sum } ?f' \ \{0..<\text{length } t\} = \text{sum } ?f' \ \{0..<i\} + \text{sum } ?f' \ \{i..<j+1\} + \text{sum } ?f' \ \{j+1..<\text{length } t\} \rangle .$

**have**  $\text{prefix-sum}: \text{sum } ?f \ \{0..<i\} = \text{sum } ?f' \ \{0..<i\}$   
**proof** -

```

have  $\forall l. 0 \leq l \wedge l < i \longrightarrow ?f l = ?f' l$ 
proof (rule allI, rule impI)
  fix l
  assume  $0 \leq l \wedge l < i$ 
  then have  $l < i$  by simp
  show  $?f l = ?f' l$ 
  proof (cases postrecording-event t l)
  case True
    let  $?G = \{k \in \{l+1..<length\ t\}. \text{prerecording-event } t\ k\}$ 
    let  $?G' = \{k \in \{l+1..<length\ t\}. \text{prerecording-event } ?t\ k\}$ 
    let  $?A = \{k \in \{l+1..<i\}. \text{prerecording-event } t\ k\}$ 
    let  $?B = \{k \in \{i..<j+1\}. \text{prerecording-event } t\ k\}$ 
    let  $?C = \{k \in \{j+1..<length\ t\}. \text{prerecording-event } t\ k\}$ 
    let  $?A' = \{k \in \{l+1..<i\}. \text{prerecording-event } ?t\ k\}$ 
    let  $?B' = \{k \in \{i..<j+1\}. \text{prerecording-event } ?t\ k\}$ 
    let  $?C' = \{k \in \{j+1..<length\ t\}. \text{prerecording-event } ?t\ k\}$ 
    have card-G:  $\text{card } ?G = \text{card } ?A + \text{card } ?B + \text{card } ?C$ 
    proof -
      have  $?G = ?A \cup (?B \cup ?C)$  using assms  $\langle l < i \rangle$  by auto
      then have  $\text{card } ?G = \text{card } (?A \cup (?B \cup ?C))$  by simp
      also have  $\text{card } (?A \cup (?B \cup ?C)) = \text{card } ?A + \text{card } (?B \cup ?C)$ 
      proof -
        have  $?A \cap (?B \cup ?C) = \{\}$  using  $\langle l < i \rangle$  assms by auto
        then show ?thesis by (simp add: card-Un-disjoint disjoint-iff-not-equal)
      qed
      also have  $\text{card } ?A + \text{card } (?B \cup ?C) = \text{card } ?A + \text{card } ?B + \text{card } ?C$ 
      proof -
        have  $?B \cap ?C = \{\}$  by auto
        then show ?thesis by (simp add: card-Un-disjoint disjoint-iff-not-equal)
      qed
      finally show ?thesis by simp
    qed
  case False
    have card-G':  $\text{card } ?G' = \text{card } ?A' + \text{card } ?B' + \text{card } ?C'$ 
    proof -
      have  $?G' = ?A' \cup (?B' \cup ?C')$  using assms  $\langle l < i \rangle$  by auto
      then have  $\text{card } ?G' = \text{card } (?A' \cup (?B' \cup ?C'))$  by simp
      also have  $\text{card } (?A' \cup (?B' \cup ?C')) = \text{card } ?A' + \text{card } (?B' \cup ?C')$ 
      proof -
        have  $?A' \cap (?B' \cup ?C') = \{\}$  using  $\langle l < i \rangle$  assms by auto
        then show ?thesis by (simp add: card-Un-disjoint disjoint-iff-not-equal)
      qed
      also have  $\text{card } ?A' + \text{card } (?B' \cup ?C') = \text{card } ?A' + \text{card } ?B' + \text{card } ?C'$ 
      proof -
        have  $?B' \cap ?C' = \{\}$  by auto
        then show ?thesis by (simp add: card-Un-disjoint disjoint-iff-not-equal)
      qed
      finally show ?thesis by simp
    qed
  qed
  have card ?G = card ?G'

```

```

proof –
  have  $\text{card } ?A = \text{card } ?A'$ 
  proof –
    have  $\{k \in \{0..<i\}. \text{prerecording-event } t\ k\} = \{k \in \{0..<i\}. \text{prerecord-}$ 
ing-event  $?t\ k\}$ 
    using assms same-cardinality-post-swap-1 by blast
    then have  $?A = ?A'$  by auto
    then show ?thesis by simp
  qed
  moreover have  $\text{card } ?B = \text{card } ?B'$  using assms same-cardinality-post-swap-2
by blast
  moreover have  $\text{card } ?C = \text{card } ?C'$ 
  proof –
    have  $?C = ?C'$  using assms same-cardinality-post-swap-3 by auto
    then show ?thesis by simp
  qed
  ultimately show ?thesis using card-G card-G' by linarith
  qed
  moreover have postrecording-event ?t l using True same-postrec-prefix <l
< i> by blast
  moreover have  $\text{length } ?t = \text{length } t$  using assms(3) assms(4) by fastforce
  ultimately show ?thesis using True by presburger
next
  case False
  then have  $\sim \text{postrecording-event } ?t\ l$  using same-postrec-prefix <l <i> by
blast
  then show ?thesis using False by simp
  qed
  qed
  then show ?thesis using sum-eq-if-same-subterms by auto
qed

have infix-sum:  $\text{sum } ?f\ \{i..<j+1\} = \text{sum } ?f'\ \{i..<j+1\} + 1$ 
proof –
  have sum-decomp-f:  $\text{sum } ?f\ \{i..<j+1\} = \text{sum } ?f\ \{i..<i+2\} + \text{sum } ?f\ \{i+2..<j+1\}$ 
by (rule sym, rule sum.atLeastLessThan-concat) (use <i <j> in simp-all)
  have sum-decomp-f':  $\text{sum } ?f'\ \{i..<j+1\} = \text{sum } ?f'\ \{i..<i+2\} + \text{sum } ?f'\$ 
 $\{i+2..<j+1\}$ 
by (rule sym, rule sum.atLeastLessThan-concat) (use <i <j> in simp-all)
  have  $\text{sum } ?f\ \{i+2..<j+1\} = \text{sum } ?f'\ \{i+2..<j+1\}$ 
proof –
  have  $\forall l. i+2 \leq l \wedge l < j+1 \longrightarrow ?f\ l = ?f'\ l$ 
proof (rule allI, rule impI)
  fix l
  assume asm:  $i+2 \leq l \wedge l < j+1$ 
  have  $?f\ l = 0$ 
proof (cases l = j)
  case True
  then have  $\sim \text{postrecording-event } t\ l$ 

```

```

    using assms(1) postrecording-event prerecording-event by auto
    then show ?thesis by simp
  next
    case False
    then have  $i < l \wedge l < j$  using assms asm by simp
    then have  $\sim$  regular-event ( $t ! l$ ) using assms by blast
    then have  $\sim$  postrecording-event  $t l$  unfolding postrecording-event by
simp
    then show ?thesis by simp
  qed
  moreover have  $?f' l = 0$ 
  proof -
    have  $\forall k. i+1 < k \wedge k < j+1 \longrightarrow \sim$  regular-event ( $?t ! k$ ) using assms
swap-events by blast
    then have  $\sim$  regular-event ( $?t ! l$ ) using asm by simp
    then have  $\sim$  postrecording-event  $?t l$  unfolding postrecording-event by
simp
    then show ?thesis by simp
  qed
  ultimately show  $?f l = ?f' l$  by simp
qed
then show ?thesis using sum-eq-if-same-subterms by simp
qed

moreover have  $\text{sum } ?f \{i..<i+2\} = 1 + \text{sum } ?f' \{i..<i+2\}$ 
proof -
  have int-def:  $\{i..<i+2\} = \{i, i+1\}$  by auto
  then have  $\text{sum } ?f \{i, i+1\} = ?f i + ?f (i+1)$  by simp
  moreover have  $\text{sum } ?f' \{i, i+1\} = ?f' i + ?f' (i+1)$  using int-def by simp

  moreover have  $?f (i+1) = 0$ 
  proof (cases j = i+1)
    case True
    then have prerecording-event  $t (i+1)$  using assms by simp
    then have  $\sim$  postrecording-event  $t (i+1)$ 
      unfolding postrecording-event using prerecording-event by simp
    then show ?thesis by simp
  next
    case False
    then have  $\sim$  regular-event ( $t ! (i+1)$ ) using assms by simp
    then have  $\sim$  postrecording-event  $t (i+1)$  unfolding postrecording-event by
simp
    then show ?thesis by simp
  qed
  moreover have  $?f' i = 0$ 
  proof -
    have prerecording-event  $?t i$  using assms swap-events by blast
    then have  $\sim$  postrecording-event  $?t i$ 
      unfolding postrecording-event using prerecording-event by simp

```

```

    then show ?thesis by simp
  qed
  moreover have ?f i = ?f' (i+1) + 1
  proof -
    have pi: postrecording-event t i using assms by simp
    moreover have pip1: postrecording-event ?t (i+1) using assms swap-events
  by blast
    let ?G = {k ∈ {i+1..

```

```

have suffix-sum: sum ?f {j+1..\forall l. l > j \longrightarrow ?f\ l = ?f'\ l
  proof (rule allI, rule impI)
    fix l
    assume l > j
    then show ?f l = ?f' l
    proof (cases postrecording-event t l)
      case True
        let ?G = {k  $\in$  {l+1..\in {l+1..\in {j+1..\in {j+1..\sim postrecording-event ?t l using same-postrec-suffix <l > j> by
simp
        then show ?thesis using False by simp
    qed
  qed
  then have  $\forall k. j+1 \leq k \wedge k < \text{length } t \longrightarrow ?f\ k = ?f'\ k$ 
  by simp
  moreover have length t = length ?t
  using assms(3) assms(4) swap-identical-length by blast
  ultimately show ?thesis by (blast intro:sum-eq-if-same-subterms)
qed
have sum ?f {0..} = sum ?f' {0..} using prefix-sum by simp
  moreover have sum ?f {i..

```

**ultimately have**  $\text{sum } ?f' \{0..<\text{length } ?t\} = n$  **by** *presburger*  
**then show** *?thesis unfolding count-violations-def* **by** *presburger*  
**qed**

**lemma** *desired-trace-always-exists*:  
**assumes**  
*trace init t final*  
**shows**  
 $\exists t'. \text{mset } t' = \text{mset } t$   
 $\wedge \text{all-prerecording-before-postrecording } t'$   
**using** *assms* **proof** (*induct count-violations t arbitrary: t*)  
**case** *0*  
**then show** *?case*  
**proof** (*cases*  $\exists i. \text{prerecording-event } t \ i$ )  
**case** *False*  
**then have**  $\forall j. \sim \text{prerecording-event } t \ j$  **by** *auto*  
**then have**  $\forall j. j \leq 0 \longrightarrow \sim \text{postrecording-event } t \ j$   
**using** *0.premis init-is-s-t-0 no-initial-process-snapshot postrecording-event* **by**  
*auto*  
**moreover have**  $\forall j. j > 0 \longrightarrow \sim \text{prerecording-event } t \ j$  **using** *False* **by** *auto*  
**moreover have** *length t > 0*  
**by** (*metis 0.premis all-processes-snapshotted-in-final-state length-greater-0-conv*  
*no-initial-process-snapshot tr-init trace-and-start-determines-end*)  
**ultimately show** *?thesis using 0.premis False* **by** *auto*  
**next**  
**case** *True*  
**let** *?Is = {i. prerecording-event t i}*  
**have** *?Is ≠ empty*  
**by** (*simp add: True*)  
**moreover have** *fin-Is: finite ?Is*  
**proof** (*rule ccontr*)  
**assume**  $\sim \text{finite } ?Is$   
**then obtain** *i where i > length t prerecording-event t i*  
**by** (*simp add: prerecording-event*)  
**then show** *False using prerecording-event* **by** *auto*  
**qed**  
**let** *?i = Max ?Is*  
**have** *pi: prerecording-event t ?i*  
**using** *Max-in calculation fin-Is* **by** *blast*  
**have** *?i < length t*  
**proof** (*rule ccontr*)  
**assume**  $\sim ?i < \text{length } t$   
**then show** *False*  
**using** *calculation fin-Is computation.prerecording-event computation-axioms*  
**by** *force*  
**qed**  
**moreover have**  $\forall j. j \geq ?i+1 \longrightarrow \sim \text{prerecording-event } t \ j$   
**proof** –  
**have**  $\forall j. j > ?i \longrightarrow \sim \text{prerecording-event } t \ j$

```

    using Max-less-iff fin-Is by auto
    then show ?thesis by auto
qed
moreover have  $\forall j. j < ?i+1 \longrightarrow \sim \text{postrecording-event } t \ j$ 
proof -
  have  $\forall j. j \leq ?i \longrightarrow \sim \text{postrecording-event } t \ j$ 
  proof (rule allI, rule impI, rule ccontr)
    fix j
    assume  $j \leq ?i \sim \sim \text{postrecording-event } t \ j$ 
    then have  $j < ?i$ 
      by (metis add-diff-inverse-nat dual-order.antisym le-add1 pi postrecording-event prerecording-event)
    then have count-violations  $t > 0$ 
    proof -
      have (if postrecording-event  $t \ j$ 
        then card  $\{l \in \{j+1..<\text{length } t\}. \text{prerecording-event } t \ l\}$ 
        else 0) = card  $\{l \in \{j+1..<\text{length } t\}. \text{prerecording-event } t \ l\}$ 
        using  $\langle \sim \sim \text{postrecording-event } t \ j \rangle$  by simp
        moreover have card  $\{l \in \{j+1..<\text{length } t\}. \text{prerecording-event } t \ l\} > 0$ 
        proof -
          have  $j + 1 \leq ?i \wedge ?i < \text{length } t$ 
          using  $\langle \text{Max } \{i. \text{prerecording-event } t \ i\} < \text{length } t \rangle \langle j < \text{Max } \{i. \text{prerecording-event } t \ i\} \rangle$ 
          by simp
          moreover have prerecording-event  $t \ ?i$  using pi by simp
          ultimately have  $\{l \in \{j+1..<\text{length } t\}. \text{prerecording-event } t \ l\} \neq \text{empty}$ 
        by fastforce
        then show ?thesis by fastforce
      qed
    ultimately show ?thesis
      by (metis (no-types, lifting) violations-0-implies-all-subterms-0  $\langle \text{Max } \{i. \text{prerecording-event } t \ i\} < \text{length } t \rangle \langle j < \text{Max } \{i. \text{prerecording-event } t \ i\} \rangle$  atLeast-LessThan-iff less-trans linorder-not-le neq0-conv)
    qed
    then show False using 0 by simp
  qed
  then show ?thesis by auto
qed
moreover have  $?i+1 \leq \text{length } t$ 
  using calculation(2) by simp
  ultimately show ?thesis using 0.prem by blast
qed
next
case (Suc n)
  then obtain  $i \ j$  where ind: postrecording-event  $t \ i$  prerecording-event  $t \ j$ 
     $\forall k. (i < k \wedge k < j) \longrightarrow \sim \text{regular-event } (t \ ! \ k)$ 
     $i < j \ j < \text{length } t$  using exists-neighboring-violation-pair
  Suc by force
  then have trace init (swap-events  $i \ j \ t$ ) final

```

$\wedge (\forall k. k \geq j + 1 \longrightarrow S (\text{swap-events } i j t) k = S t k)$   
 $\wedge (\forall k. k \leq i \longrightarrow S (\text{swap-events } i j t) k = S t k)$   
**using** *Suc swap-events* **by** *presburger*  
**moreover have**  $\text{mset } (\text{swap-events } i j t) = \text{mset } t$  **using** *swap-events-perm* *ind*  
**by** *blast*  
**moreover have**  $\text{count-violations } (\text{swap-events } i j t) = n$   
**using** *count-violations-swap* *Suc* *ind* **by** *simp*  
**ultimately show** *?case* **using** *Suc.hyps* **by** *metis*  
**qed**

**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** –

**obtain**  $t'$  **where**  $\text{mset } t' = \text{mset } t$  **and**  
*all-prerecording-before-postrecording*  $t'$   
**using** *assms* *desired-trace-always-exists* **by** *blast*  
**then show** *?thesis* **using** *snapshot-after-all-prerecording-events*  
**by** *blast*  
**qed**

## 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*

$\text{stable } p \equiv (\forall c. p c \longrightarrow (\forall t c'. \text{trace } c t c' \longrightarrow p c'))$

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

**assumes**  
*trace init t final*  
**shows**  
 $\text{stable } (\lambda c. \text{has-snapshotted } c p)$   
**using** *snapshot-stable* *stable-def* **by** *auto*

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

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

**lemma** *split-S*:

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

```

proof –
  have  $t = \text{take } i \ t \ @ \ \text{drop } i \ t$  by simp
  then show ?thesis
    by (metis split-trace assms exists-trace-for-any-i
          trace-and-start-determines-end)
qed

theorem Stable-Property-Detection:
  assumes
    stable p and
    trace init t final and
     $(t', i) = \text{some-snapshot-state } t$  and
     $p (S \ t' \ i)$ 
  shows
     $p \ \text{final}$ 
proof –
  have  $\exists t' \ i. \ \text{trace init } t \ \text{final}$ 
     $\wedge \ \text{trace init } t' \ \text{final} \wedge \ \text{mset } t' = \text{mset } t$ 
     $\wedge \ \text{state-equal-to-snapshot } (S \ t' \ i) \ \text{final}$ 
    using snapshot-algorithm-is-correct assms(2) by blast
  then have  $\text{trace init } t' \ \text{final}$ 
    using assms
    unfolding some-snapshot-state-def
    by auto (metis (mono-tags, lifting) case-prod-conv tfl-some)
  then show ?thesis
    using assms stable-def split-S by metis
qed

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  $\Rightarrow$  [] | LCons *x t'*  $\Rightarrow$  *x* # *ltake* *i t'*)

**primrec** *ldrop* **where**

*ldrop* 0 *t* = *t*  
| *ldrop* (Suc *i*) *t* = (case *t* of LNil  $\Rightarrow$  LNil | LCons *x t'*  $\Rightarrow$  *ldrop* *i t'*)

**lemma** *ltake-LNil[simp]*: *ltake* *i* LNil = []

**by** (induct *i*) auto

**lemma** *ltake-LCons*:  $0 < i \Longrightarrow \text{ltake } i \text{ (LCons } x \text{ } t) = x \# \text{ltake } (i - 1) \text{ } t$

**by** (induct *i*) auto

**lemma** *take-ltake*:  $i \leq j \Longrightarrow \text{take } i \text{ (ltake } j \text{ } xs) = \text{ltake } i \text{ } xs$

**by** (induct *j* arbitrary: *i xs*) (auto simp: le-Suc-eq take-Cons' ltake-LCons split: llist.splits if-splits)

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

**by** (induct *n* arbitrary: *i xs*)

(auto simp: nth-Cons' gr0-conv-Suc eSuc-enat[symmetric] split: llist.splits)

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

**by** (induct *i* arbitrary: *xs*)

(auto simp: zero-enat-def[symmetric] eSuc-enat split: llist.splits enat.splits)

**lemma** *ltake-prepend*:

*ltake* *i* (prepend *xs t*) = (if  $i \leq \text{length } xs$  then *take* *i xs* else *xs* @ *ltake* (*i* - *length xs*) *t*)

**proof** (induct *i* arbitrary: *xs t*)

**case** 0

**then show** ?case

**by** (cases *xs*) auto

**next**

**case** (Suc *i*)

**then show** ?case

**by** (cases *xs*) auto

**qed**

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

**by** (induct *i* arbitrary: *t*) (auto split: llist.splits)

**context** *distributed-system*

**begin**

**coinductive** *cotrace* **where**

*cotr-init*: *cotrace* *c* LNil

| *contr-step*:  $\llbracket c \vdash ev \mapsto c'; \text{cotrace } c' t \rrbracket \implies \text{cotrace } c (LCons \text{ ev } t)$

**lemma** *cotrace-trace*:  $\text{cotrace } c t \implies \exists !c'. \text{trace } c (\text{ltake } i t) c'$

**proof** (*induct i arbitrary: c t*)  
 case (*Suc i*)  
 then show ?*case*  
 proof (*cases t*)  
 case (*LCons ev t'*)  
 with *Suc(2)* obtain *c'* where  $c \vdash ev \mapsto c' \text{cotrace } c' t'$   
 by (*auto elim: cotrace.cases*)  
 with *Suc(1)*[*OF <cotrace c' t'>*] show ?*thesis*  
 by (*auto simp: LCons elim: trace.intros(2) elim: trace.cases trace-and-start-determines-end*)  
 qed (*auto intro: trace.intros elim: trace.cases*)  
 qed (*auto simp: zero-enat-def[symmetric] intro: trace.intros elim: trace.cases*)

**lemma** *cotrace-trace'*:  $\text{cotrace } c t \implies \exists c'. \text{trace } c (\text{ltake } i t) c'$   
 by (*metis cotrace-trace*)

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

**lemma** *cotrace-trace-cos*:  $\text{cotrace } c t \implies \text{trace } c (\text{ltake } i t) (\text{cos } c t i)$   
 unfolding *cos-def s-def*  
 by (*subst take-ltake, auto dest!: cotrace-trace[of - - i] elim!: theI'*)

**lemma** *s-0[simp]*:  $s c t 0 = c$   
 unfolding *s-def*  
 by (*auto intro!: the-equality[where P = trace c []] trace.intros elim: trace.cases*)

**lemma** *s-chop*:  $i \leq \text{length } t \implies s c t i = s c (\text{take } i t) i$   
 unfolding *s-def*  
 by *auto*

**lemma** *cotrace-prepend*:  $\text{trace } c t c' \implies \text{cotrace } c' u \implies \text{cotrace } c (\text{prepend } t u)$   
 by (*induct c t c' rule: trace.induct*) (*auto intro: cotrace.intros*)

**lemma** *s-Cons*:  $\exists c''. \text{trace } c' xs c'' \implies c \vdash ev \mapsto c' \implies s c (\text{ev} \# xs) (\text{Suc } i) = s c' xs i$   
 by (*smt exists-trace-for-any-i take-Suc-Cons tr-step trace-and-start-determines-end*)

**lemma** *cotrace-ldrop*:  $\text{cotrace } c t \implies i \leq \text{llength } t \implies \text{cotrace } (\text{cos } c t i) (\text{ldrop } i t)$

**proof** (*induct i arbitrary: c t*)  
 case (*Suc i*)  
 then show ?*case*  
 proof (*cases t*)  
 case (*LCons ev t'*)  
 with *Suc(2)* obtain *c'* where  $c \vdash ev \mapsto c' \text{cotrace } c' t'$   
 by (*auto elim: cotrace.cases*)  
 with *Suc(1)*[*OF <cotrace c' t'>*] *Suc(3)* show ?*thesis*

**by** (*auto simp: LCons cos-def eSuc-enat[symmetric] s-chop[symmetric] s-Cons[OF cotrace-trace']*)  
**qed** (*auto intro: cotrace.intros*)  
**qed** (*auto simp: zero-enat-def[symmetric] cos-def intro: cotrace.intros*)

**end**

**locale** *cocomputation = distributed-system +*

**fixes**

*init* :: ('a, 'b, 'c) configuration

**assumes**

*finite-channels:*

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

*strongly-connected-raw:*

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

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

*at-least-two-processes:*

*card* (*UNIV* :: 'a set) > 1 **and**

*finite-processes:*

*finite* (*UNIV* :: 'a set) **and**

*no-initial-Marker:*

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

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

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

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

*no-initial-process-snapshot:*

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

*no-initial-channel-snapshot:*

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

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

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

$\wedge$  Marker  $\in$  set (*msgs (cos init t i) cid*)

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

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

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

**begin**

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

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

**lemma** *has-snapshotted:*

*cotrace init t*  $\implies$  *has-snapshotted (coS t (some-snapshot t p)) p*  $\wedge$  *some-snapshot t p*  $\leq$  *llength t*

**unfolding** *some-snapshot-def* **by** (*rule someI-ex*) (*auto dest!: l2[rule-format]*)

**lemma** *cotrace-cos*:  $\text{cotrace init } t \implies j < \text{length } t \implies$   
 $(\text{coS } t \ j) \vdash \text{lnth } t \ j \mapsto (\text{coS } t \ (\text{Suc } j))$   
**apply** (*drule cotrace-trace-cos*[*of - - Suc j*])  
**apply** (*drule step-Suc*[*rotated, of - - j*])  
**apply** (*auto split: enat.splits llist.splits*) []  
**apply** (*auto simp: s-chop*[*of j - # ltake j -*]) *cos-def nth-Cons' ltake-LCons lnth-LCons'*  
*take-Cons' take-ltake*  
*split: llist.splits enat.splits if-splits elim: order.strict-trans2*[*rotated*])  
**apply** (*subst (asm) s-chop*[*of j - # ltake j -*])  
**apply** (*auto simp: take-Cons' take-ltake split: enat.splits*)  
**done**

**lemma** *snapshot-stable*:  
 $\text{cotrace init } t \implies i \leq j \implies \text{has-snapshotted } (\text{coS } t \ i) \ p \implies \text{has-snapshotted } (\text{coS } t \ j) \ p$   
**apply** (*drule cotrace-trace-cos*[*of - - j*])  
**unfolding** *cos-def*  
**by** (*metis exists-trace-for-any-i-j order-refl s-def snapshot-stable take-ltake*)

**lemma** *no-markers-if-all-snapshotted*:  
 $\text{cotrace init } t \implies i \leq j \implies \forall p. \text{has-snapshotted } (\text{coS } t \ i) \ p \implies$   
 $\text{Marker} \notin \text{set } (\text{msgs } (\text{coS } t \ i) \ c) \implies \text{Marker} \notin \text{set } (\text{msgs } (\text{coS } t \ j) \ c)$   
**apply** (*drule cotrace-trace-cos*[*of - - j*])  
**unfolding** *cos-def*  
**by** (*metis exists-trace-for-any-i-j no-markers-if-all-snapshotted order-refl s-def take-ltake*)

**lemma** *cotrace-all-have-snapshotted*:  
**assumes** *cotrace init t*  
**shows**  $\exists i \leq \text{length } t. \forall p. \text{has-snapshotted } (\text{coS } t \ i) \ p$   
**proof** –  
**let**  $?i = \text{Max } (\text{range } (\text{some-snapshot } t))$   
**show** *?thesis*  
**using** *has-snapshotted*[*OF assms*] *snapshot-stable*[*OF assms, of some-snapshot*  
 $t - ?i$  -]  
**apply** (*intro exI*[*of - ?i*])  
**apply** (*auto simp: finite-processes*)  
**apply** (*cases length t; auto simp:* )  
**apply** (*subst Max-le-iff*)  
**apply** (*auto simp: finite-processes*)  
**apply** *blast*  
**done**

**qed**

**lemma** *no-messages-if-no-channel*:  
**assumes** *cotrace init t*  
**shows**  $\text{channel } \text{cid} = \text{None} \implies \text{msgs } (\text{coS } t \ i) \ \text{cid} = []$   
**using** *no-messages-introduced-if-no-channel*[*OF assms*][*THEN cotrace-trace-cos,*

of  $i$ ] *no-msgs-if-no-channel*, of *cid i*]  
 by (*auto simp: cos-def*)

**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** –

from *cotrace-all-have-snapshotted*[*OF assms*] **obtain**  $j :: \text{nat}$  **where**

$j: j \leq \text{llength } t \ \forall p. \text{has-snapshotted } (\text{coS } t \ j) \ p$  **by** *blast*

from  $j(2)$  **have**  $*$ : *has-snapshotted* (*coS t j*) *p* **if**  $k \geq j$  **for**  $k \ p$

**using** *snapshot-stable*[*OF assms, of j k p*] **that** **by** *auto*

**define**  $C$  **where**  $C = \{c. \text{Marker} \in \text{set } (\text{msgs } (\text{coS } t \ j) \ c)\}$

**have** *finite C*

**using** *no-messages-if-no-channel*[*OF assms, of - j*] **unfolding** *C-def*

**by** (*intro finite-subset*[*OF - finite-channels*]) *fastforce*

**define** *pick* **where**  $\text{pick} = (\lambda c. \text{SOME } k. k \leq \text{llength } t \wedge k \geq j \wedge \text{Marker} \notin \text{set}$   
 $(\text{msgs } (\text{coS } t \ k) \ c))$

{ **fix**  $c$

**assume**  $c \in C$

**then** **have**  $\exists k \leq \text{llength } t. k \geq j \wedge \text{Marker} \notin \text{set } (\text{msgs } (\text{coS } t \ k) \ c)$

**using**  $l1$ [*rule-format, of t j c*] *assms* **unfolding** *C-def* **by** *blast*

**then** **have**  $\text{pick } c \leq \text{llength } t \wedge \text{pick } c \geq j \wedge \text{Marker} \notin \text{set } (\text{msgs } (\text{coS } t \ (\text{pick}$   
 $c)) \ c)$

**unfolding** *pick-def*

**by** (*rule someI-ex*)

} **note**  $\text{pick} = \text{conjunction1} [\text{OF } \text{this}] \ \text{conjunction1} [\text{OF } \text{conjunction2} [\text{OF } \text{this}]] \ \text{conjunction2} [\text{OF}$   
 $\text{conjunction2} [\text{OF } \text{this}]]$

**show** *?thesis*

**proof** (*cases C = {}*)

**case** *True*

**with**  $j$  **show** *?thesis*

**by** (*auto intro!: exI[of - j] simp: C-def*)

**next**

**define**  $m$  **where**  $m = \text{Max } (\text{pick } ' C)$

**case** *False*

**with**  $\langle \text{finite } C \rangle$  **have**  $m: m \in \text{pick } ' C \ \forall x \in \text{pick } ' C. m \geq x$

**unfolding** *m-def* **by** *auto*

**then** **have**  $j \leq m$  **using**  $\text{pick}(2)$  **by** *auto*

**from**  $*$ [*OF <j ≤ m>*] **have**  $\text{Marker} \notin \text{set } (\text{msgs } (\text{coS } t \ m) \ c)$  **for**  $c$

**proof** (*cases c ∈ C*)

**case** *True*

**then** **show** *?thesis*

**using** *no-markers-if-all-snapshotted*[*OF assms, of pick c m c*]  $\text{pick}[of \ c] \ m \ *$

**by** *auto*

**next**

**case** *False*

**then** **show** *?thesis*

**using** *no-markers-if-all-snapshotted*[*OF assms <j ≤ m> j(2), of c*]

```

      by (auto simp: C-def)
    qed
    with *[OF ‹j ≤ m›] m pick show ?thesis by auto
  qed
qed

context
  fixes t
  assumes cotrace: cotrace init t
begin

definition final-i ≡
  (SOME i. i ≤ llength t ∧ (∀ p. has-snapshotted (coS t i) p) ∧ (∀ c. Marker ∉ set
  (msgs (coS t i) c)))

definition final where
  final = coS t final-i

lemma final-i: final-i ≤ llength t (∀ p. has-snapshotted (coS t final-i) p) (∀ c.
Marker ∉ set (msgs (coS t final-i) c))
  unfolding final-i-def
  by (rule someI2-ex[OF cotrace-all-have-snapshotted-and-no-markers[OF cotrace]];
  auto intro: cotrace-trace-cos[OF cotrace])+

lemma final: ∃ t. trace init t final (∀ p. has-snapshotted final p) (∀ c. Marker ∉ set
(msgs final c))
  unfolding final-def
  by (rule cotrace-trace-cos[OF cotrace] final-i exI)+

interpretation computation channel trans send recv init final
  apply standard
    apply (rule finite-channels)
    apply (rule strongly-connected-raw)
    apply (rule at-least-two-processes)
    apply (rule finite-processes)
    apply (rule no-initial-Marker)
    apply (rule no-msgs-if-no-channel)
    apply (rule no-initial-process-snapshot)
    apply (rule no-initial-channel-snapshot)
    apply (rule final(1))
    apply (intro allI impI)
  subgoal for t i cid
    apply (rule exI[of - llength t])
    apply (metis exists-trace-for-any-i final(3) le-cases take-all trace-and-start-determines-end)
  done
  apply (intro allI impI)
  subgoal for t p
    apply (rule exI[of - llength t])
    apply (metis exists-trace-for-any-i final(2) order-refl take-all trace-and-start-determines-end)

```

```

done
done

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

lemma copermIL: mset ys = mset xs  $\implies$  t = prepend xs z  $\implies$  coperm (prepend ys z) t
  unfolding coperm-def by auto

lemma snapshot-algorithm-is-cocorrect:
   $\exists$  t' i. cotrace init t'  $\wedge$  coperm t' t  $\wedge$  state-equal-to-snapshot (coS t' i) final  $\wedge$  i
   $\leq$  final-i
proof –
  define prefix where prefix = ltake final-i t
  define suffix where suffix = ldrop final-i t
  have [simp]: prepend prefix suffix = t
    unfolding prefix-def suffix-def prepend-ltake-ldrop-id ..
  have [simp]: cotrace final suffix
    unfolding suffix-def final-def
    by (auto simp: cotrace final-i(1) intro!: cotrace-ldrop)
  from cotrace-trace-cos[OF cotrace] have trace init prefix final
    unfolding final-def prefix-def by blast
  with snapshot-algorithm-is-correct obtain prefix' i where
    trace init prefix' final mset prefix' = mset prefix state-equal-to-snapshot (S prefix'
    i) final
    i  $\leq$  length prefix'
    by blast
  moreover from  $\langle$ mset prefix' = mset prefix $\rangle$   $\langle$ i  $\leq$  length prefix' $\rangle$  have i  $\leq$  final-i
    by (auto dest!: mset-eq-length simp: prefix-def split: enat.splits)
  ultimately show ?thesis
    by (intro exI[of - prepend prefix' suffix] exI[of - i])
    (auto simp: cos-def ltake-prepend s-chop[symmetric] intro!: cotrace-prepend
    elim!: copermIL)
qed

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-Send  $\wedge$  s' = S-Wait  $\wedge$  m = M)
     $\vee$  (c = 1  $\wedge$  p = Q  $\wedge$  q = P
     $\wedge$  s = T-Send  $\wedge$  s' = T-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-Wait  $\wedge$  s' = S-Send  $\wedge$  m = M')
     $\vee$  (c = 0  $\wedge$  p = Q  $\wedge$  q = P
     $\wedge$  s = T-Wait  $\wedge$  s' = T-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

```

$s1 \equiv ($   
 $\text{states} = (\%p. \text{if } p = P \text{ then } S\text{-Send else } T\text{-Send}),$   
 $\text{msgs} = (\%d. \text{if } d = 0 \text{ then } [\text{Marker}] \text{ else } []),$   
 $\text{process-snapshot} = (\%p. \text{if } p = P \text{ then } \text{Some } S\text{-Send else } \text{None}),$   
 $\text{channel-snapshot} = (\%d. \text{if } d = 1 \text{ then } ([], \text{Recording}) \text{ else } ([], \text{NotStarted}))$   
 $)$

**abbreviation**  $t1$  **where**  $t1 \equiv \text{Send } 0 \ P \ Q \ S\text{-Send } S\text{-Wait } M$

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

$s2 \equiv ($   
 $\text{states} = (\%p. \text{if } p = P \text{ then } S\text{-Wait else } T\text{-Send}),$   
 $\text{msgs} = (\%d. \text{if } d = 0 \text{ then } [\text{Marker}, \text{Msg } M] \text{ else } []),$   
 $\text{process-snapshot} = (\%p. \text{if } p = P \text{ then } \text{Some } S\text{-Send else } \text{None}),$   
 $\text{channel-snapshot} = (\%d. \text{if } d = 1 \text{ then } ([], \text{Recording}) \text{ else } ([], \text{NotStarted}))$   
 $)$

**abbreviation**  $t2$  **where**  $t2 \equiv \text{Send } 1 \ Q \ P \ T\text{-Send } T\text{-Wait } M'$

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

$s3 \equiv ($   
 $\text{states} = (\%p. \text{if } p = P \text{ then } S\text{-Wait else } T\text{-Wait}),$   
 $\text{msgs} = (\%d. \text{if } d = 0 \text{ then } [\text{Marker}, \text{Msg } M] \text{ else if } d = 1 \text{ then } [\text{Msg } M'] \text{ else } []),$   
 $\text{process-snapshot} = (\%p. \text{if } p = P \text{ then } \text{Some } S\text{-Send else } \text{None}),$   
 $\text{channel-snapshot} = (\%d. \text{if } d = 1 \text{ then } ([], \text{Recording}) \text{ else } ([], \text{NotStarted}))$   
 $)$

**abbreviation**  $t3$  **where**  $t3 \equiv \text{Snapshot } Q$

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

$s4 \equiv ($   
 $\text{states} = (\%p. \text{if } p = P \text{ then } S\text{-Wait else } T\text{-Wait}),$   
 $\text{msgs} = (\%d. \text{if } d = 0 \text{ then } [\text{Marker}, \text{Msg } M] \text{ else if } d = 1 \text{ then } [\text{Msg } M', \text{Marker}] \text{ else } []),$   
 $\text{process-snapshot} = (\%p. \text{if } p = P \text{ then } \text{Some } S\text{-Send else } \text{Some } T\text{-Wait}),$   
 $\text{channel-snapshot} = (\%d. \text{if } d = 1 \text{ then } ([], \text{Recording}) \text{ else if } d = 0 \text{ then } ([], \text{Recording}) \text{ else } ([], \text{NotStarted}))$   
 $)$

**abbreviation**  $t4$  **where**  $t4 \equiv \text{RecvMarker } 0 \ Q \ P$

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

$s5 \equiv ($   
 $\text{states} = (\%p. \text{if } p = P \text{ then } S\text{-Wait else } T\text{-Wait}),$   
 $\text{msgs} = (\%d. \text{if } d = 0 \text{ then } [\text{Msg } M] \text{ else if } d = 1 \text{ then } [\text{Msg } M', \text{Marker}] \text{ else } []),$   
 $\text{process-snapshot} = (\%p. \text{if } p = P \text{ then } \text{Some } S\text{-Send else } \text{Some } T\text{-Wait}),$   
 $\text{channel-snapshot} = (\%d. \text{if } d = 0 \text{ then } ([], \text{Done}) \text{ else if } d = 1 \text{ then } ([], \text{Done}))$   
 $)$

*Recording*) else ( $\square$ , *NotStarted*)  
 $\rangle$

**abbreviation** *t5* **where**  $t5 \equiv \text{Recv } 1 \ P \ Q \ S\text{-Wait } S\text{-Send } M'$

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

$s6 \equiv \langle$   
 $\text{states} = (\%p. \text{if } p = P \text{ then } S\text{-Send} \text{ else } T\text{-Wait}),$   
 $\text{msgs} = (\%d. \text{if } d = 0 \text{ then } [Msg \ M] \text{ else if } d = 1 \text{ then } [Marker] \text{ else } \square),$   
 $\text{process-snapshot} = (\%p. \text{if } p = P \text{ then } \text{Some } S\text{-Send} \text{ else } \text{Some } T\text{-Wait}),$   
 $\text{channel-snapshot} = (\%d. \text{if } d = 0 \text{ then } (\square, \text{Done}) \text{ else if } d = 1 \text{ then } ([M],$   
*Recording*) else ( $\square$ , *NotStarted*)  
 $\rangle$

**abbreviation** *t6* **where**  $t6 \equiv \text{RecvMarker } 1 \ P \ Q$

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

$s7 \equiv \langle$   
 $\text{states} = (\%p. \text{if } p = P \text{ then } S\text{-Send} \text{ else } T\text{-Wait}),$   
 $\text{msgs} = (\%d. \text{if } d = 0 \text{ then } [Msg \ M] \text{ else if } d = 1 \text{ then } \square \text{ else } \square),$   
 $\text{process-snapshot} = (\%p. \text{if } p = P \text{ then } \text{Some } S\text{-Send} \text{ else } \text{Some } T\text{-Wait}),$   
 $\text{channel-snapshot} = (\%d. \text{if } d = 0 \text{ then } (\square, \text{Done}) \text{ else if } d = 1 \text{ then } ([M],$   
*Done*) else ( $\square$ , *NotStarted*)  
 $\rangle$

**lemma** *s7-no-marker*:

**shows**

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

**by** *simp*

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

**proof**

**have** *distributed-system chan*

**proof**

**show**  $\forall i. \nexists p. \text{chan } i = \text{Some } (p, p)$  **by** *simp*

**qed**

**show**  $\forall p \ q. p \neq q \longrightarrow (\lambda p \ q. \exists i. \text{chan } i = \text{Some } (p, q))^{++} \ p \ q$

**proof** ((*rule allI*)<sup>+</sup>, *rule impI*)

**fix**  $p \ q :: PType$  **assume**  $p \neq q$

**then have**  $(p = P \wedge q = Q) \vee (p = Q \wedge q = P)$

**using** *PType.exhaust* **by** *auto*

**then have**  $\exists i. \text{chan } i = \text{Some } (p, q)$  **by** (*elim disjE*) *auto*

**then show**  $(\lambda p \ q. \exists i. \text{chan } i = \text{Some } (p, q))^{++} \ p \ q$  **by** *blast*

**qed**

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

**proof** –

**have**  $\{i. \exists p \ q. \text{chan } i = \text{Some } (p, q)\} = \{0, 1\}$  **by** *auto*

**then show** *?thesis* **by** *simp*

**qed**

```

show 1 < card (UNIV :: PType set)
proof -
  have (UNIV :: PType set) = {P, Q}
    using PType.exhaust by blast
  then have card (UNIV :: PType set) = 2
    by (metis One-nat-def PType.distinct(1) Suc-1 card.insert card.empty fi-
nite.emptyI finite.insertI insert-absorb insert-not-empty singletonD)
  then show ?thesis by auto
qed
show finite (UNIV :: PType set)
proof -
  have (UNIV :: PType set) = {P, Q}
    using PType.exhaust by blast
  then show ?thesis
    by (metis finite.emptyI finite.insertI)
qed
show  $\forall i. \nexists p. \text{chan } i = \text{Some } (p, p)$  by simp
show  $\forall i. (\exists p q. \text{chan } i = \text{Some } (p, q)) \longrightarrow \text{Marker} \notin \text{set } (\text{msgs } \text{init } i)$  by auto
show  $\forall i. \text{chan } i = \text{None} \longrightarrow \text{msgs } \text{init } i = []$  by auto
show  $\forall p. \neg \text{ps } \text{init } p \neq \text{None}$  by auto
show  $\forall i. \text{cs } \text{init } i = ([], \text{NotStarted})$  by auto
show  $\exists t. \text{distributed-system.trace } \text{chan } \text{Example.trans } \text{send } \text{recv } \text{init } t \ s?$ 
proof -
  let ?t = [t0, t1, t2, t3, t4, t5, t6]
  have distributed-system.next chan trans send recv init t0 s1
  proof -
    have distributed-system.can-occur chan trans send recv t0 init
      using <distributed-system chan> distributed-system.can-occur-def by fastforce
    then show ?thesis
      by (simp add: <distributed-system chan> distributed-system.next-snapshot)
  qed
moreover have distributed-system.next chan trans send recv s1 t1 s2
proof -
  have distributed-system.can-occur chan trans send recv t1 s1
    using <distributed-system chan> distributed-system.can-occur-def by fastforce
  then show ?thesis
    by (simp add: <distributed-system chan> distributed-system.next-send)
  qed
moreover have distributed-system.next chan trans send recv s2 t2 s3
proof -
  have distributed-system.can-occur chan trans send recv t2 s2
    using <distributed-system chan> distributed-system.can-occur-def by fastforce
  moreover have  $\forall r. r \neq P \longrightarrow r = Q$  using PType.exhaust by auto
  ultimately show ?thesis by (simp add: <distributed-system chan> dis-
tributed-system.next-send)
  qed
moreover have distributed-system.next chan trans send recv s3 t3 s4
proof -
  have distributed-system.can-occur chan trans send recv t3 s3

```

```

    using ‹distributed-system chan› distributed-system.can-occur-def by fastforce
    moreover have  $\forall p'. p' \neq P \longrightarrow p' = Q$  using PType.exhaust by auto
    ultimately show ?thesis by (simp add: ‹distributed-system chan› distributed-system.next-snapshot)
  qed
  moreover have distributed-system.next chan trans send recv s4 t4 s5
  proof -
    have distributed-system.can-occur chan trans send recv t4 s4
    using ‹distributed-system chan› distributed-system.can-occur-def by fastforce
    then show ?thesis
    by (simp add: ‹distributed-system chan› distributed-system.next-def)
  qed
  moreover have distributed-system.next chan trans send recv s5 t5 s6
  proof -
    have distributed-system.can-occur chan trans send recv t5 s5
    using ‹distributed-system chan› distributed-system.can-occur-def by fastforce
    then show ?thesis
    by (simp add: ‹distributed-system chan› distributed-system.next-def)
  qed
  moreover have distributed-system.next chan trans send recv s6 t6 s7
  proof -
    have distributed-system.can-occur chan trans send recv t6 s6
    using ‹distributed-system chan› distributed-system.can-occur-def by fastforce
    then show ?thesis
    by (simp add: ‹distributed-system chan› distributed-system.next-def)
  qed
  ultimately have distributed-system.trace chan trans send recv init ?t s7
  by (meson ‹distributed-system chan› distributed-system.trace.simps)
  then show ?thesis by blast
  qed
  show  $\forall t i cid. distributed-system.trace chan Example.trans send recv init t s7 \wedge$ 
    Marker  $\in$  set (msgs (distributed-system.s chan Example.trans send recv init
    t i) cid)  $\longrightarrow$ 
    ( $\exists j \geq i. Marker \notin$  set (msgs (distributed-system.s chan Example.trans send
    recv init t j) cid))
  proof ((rule allI)+, (rule impI)+)
    fix t i cid
    assume asm: distributed-system.trace chan Example.trans send recv init t s7  $\wedge$ 
      Marker  $\in$  set (msgs (distributed-system.s chan Example.trans send
      recv init t i) cid)
    have tr-exists: distributed-system.trace chan Example.trans send recv init t s7
  using asm by blast
    have marker-in-channel: Marker  $\in$  set (msgs (distributed-system.s chan Exam-
    ple.trans send recv init t i) cid) using asm by simp
    have s7-is-fin: s7 = (distributed-system.s chan Example.trans send recv init t
    (length t))
    by (metis (no-types, lifting) ‹distributed-system chan› ‹distributed-system.trace
    chan Example.trans send recv init t s7› distributed-system.exists-trace-for-any-i dis-
    tributed-system.trace-and-start-determines-end order-refl take-all)
  
```

```

have  $i < \text{length } t$ 
proof (rule ccontr)
  assume  $\sim i < \text{length } t$ 
  then have  $\text{distributed-system.trace chan Example.trans send recv}$ 
    ( $\text{distributed-system.s chan Example.trans send recv init } t \text{ (length } t)$ )
     $\square$ 
    ( $\text{distributed-system.s chan Example.trans send recv init } t$ )
  by (metis (no-types, lifting)  $\langle \text{distributed-system chan} \rangle$  distributed-system.exists-trace-for-any-i
    distributed-system.trace.simps distributed-system.trace-and-start-determines-end not-less
    s7-is-fin take-all tr-exists)
  then have  $\text{Marker} \notin \text{set (msgs (distributed-system.s chan Example.trans send}$ 
     $\text{recv init } t \text{ i) cid)}$ 
  proof –
    have  $\text{distributed-system.s chan Example.trans send recv init } t \text{ i} = s7$ 
    using  $\langle \text{distributed-system chan} \rangle \langle \text{distributed-system.trace chan Example.trans}$ 
     $\text{send recv (distributed-system.s chan Example.trans send recv init } t \text{ (length } t)) \square$ 
    ( $\text{distributed-system.s chan Example.trans send recv init } t \text{ i} \rangle$  distributed-system.trace.simps
    s7-is-fin by fastforce
    then show ?thesis using s7-no-marker by simp
  qed
  then show False using marker-in-channel by simp
qed
then show ( $\exists j \geq i. \text{Marker} \notin \text{set (msgs (distributed-system.s chan Example.trans}$ 
   $\text{send recv init } t \text{ j) cid)}$ )
proof –
  have  $\text{distributed-system.trace chan Example.trans send recv}$ 
    ( $\text{distributed-system.s chan Example.trans send recv init } t$ )
    ( $\text{take ((length } t) - i) \text{ (drop } i \text{ t)}$ )
    ( $\text{distributed-system.s chan Example.trans send recv init } t \text{ (length } t)$ )
  using  $\langle \text{distributed-system chan} \rangle \langle i < \text{length } t \rangle$  distributed-system.exists-trace-for-any-i-j
  less-imp-le-nat tr-exists by blast
  then have  $\text{Marker} \notin \text{set (msgs (distributed-system.s chan Example.trans send}$ 
     $\text{recv init } t \text{ (length } t)) \text{ cid)}$ 
  proof –
    have  $\text{distributed-system.s chan Example.trans send recv init } t \text{ (length } t) =$ 
     $s7$ 
    by (simp add: s7-is-fin)
    then show ?thesis using s7-no-marker by simp
  qed
  then show ?thesis
    using  $\langle i < \text{length } t \rangle$  less-imp-le-nat by blast
qed
qed
show  $\forall t \ p. \text{distributed-system.trace chan Example.trans send recv init } t \ s7 \longrightarrow$ 
  ( $\exists i. \text{ps (distributed-system.s chan Example.trans send recv init } t \text{ i) } p \neq$ 
  None  $\wedge i \leq \text{length } t$ )
proof ((rule allI)+, rule impI)
  fix  $t \ p$ 
  assume  $\text{distributed-system.trace chan Example.trans send recv init } t \ s7$ 

```

```

have s7-is-fin: s7 = (distributed-system.s chan Example.trans send recv init t
(length t))
  by (metis (no-types, lifting) <distributed-system chan> <distributed-system.trace
chan Example.trans send recv init t s7> distributed-system.exists-trace-for-any-i dis-
tributed-system.trace-and-start-determines-end order-refl take-all)
  moreover have has-snapshotted s7 p by simp
  ultimately show ( $\exists i. ps$  (distributed-system.s chan Example.trans send recv
init t i)  $p \neq None \wedge i \leq length t$ )
    by auto
  qed
qed

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.