HOL-CSP_OpSem – Operational Semantics formally proven in HOL-CSP

Benoît Ballenghien and Burkhart Wolff

March 17, 2025

Abstract

Recently, a modern version of Roscoe and Brookes [3] Failure-Divergence Semantics for CSP has been formalized in Isabelle [7] and extended [1]. The resulting framework is purely denotational and, given the possibility to define arbitrary events in a HOL-type, more expressive than the original.

However, there is a need for an operational semantics for CSP. From the latter, model-checkers, symbolic execution engines for test-case generators, and animators and simultors can be constructed. In the literature, a few versions of operational semantics for CSP have been proposed, where it is assumed, of course, that denotational and operational constructs coincide, but this is not obvious at first glance.Recently, a modern version of Roscoe and Brookes [3] Failure-Divergence Semantics for CSP has been formalized in Isabelle [7] and extended [1]. The resulting framework is purely denotational and, given the possibility to define arbitrary events in a HOL-type, more expressive than the original.

However, there is a need for an operational semantics for CSP. From the latter, model-checkers, symbolic execution engines for test-case generators, and animators and simulators can be constructed. In the literature, a few versions of operational semantics for CSP have been proposed, where it is assumed, of course, that denotational and operational constructs coincide, but this is not obvious at first glance.

The present work addresses this issue by providing the first (to our knowledge) formal theory of operational behavior derived from HOL-CSP via a bridge definition between the denotational and the operational semantics. In fact, the construction is done via locale contexts to be as general as possible, and several possibilities are discussed.

As a bonus, we have proven new "'laws" for HOL-CSP.

Contents

Chapter 1

Introduction

1.1 Motivations

HOL-CSP [7] is a formalization in Isabelle/HOL of the work of Hoare and Roscoe on the denotational semantics of the Failure/Divergence Model of CSP. It follows essentially the presentation of CSP in Roscoe's Book "Theory and Practice of Concurrency" [4] and the semantic details in a joint paper of Roscoe and Brooks "An improved failures model for communicating processes" [3].

Basically, the session HOL-CSP introduces the type $('a, 'r)\ process_{ptick}$, several classic CSP operators and number of "laws" (i.e. derived equations) that govern their interactions. HOL-CSP has been extended by a theory of architectural operators HOL-CSPM inspired by the CSP_M language of the model-checker FDR. While in FDR these operators are basically macros over finite lists and sets, the HOL-CSPM theory treats them in their own right for the most general cases.

The present work addresses the problem of operational semantics for CSP which are the foundations for finite model-checking and process simulation techniques. In the literature, there are a few versions of operational semantics for CSP, which lend themselves to the constructions of labelled transition systems (LTS). Of course, denotational and operational constructs are expected to coincide, but this is not obvious at first glance. As a key contribution, we will define the operational derivation operators $P \rightsquigarrow_\tau Q$ ("P evolves internally to Q) and $P \rightsquigarrow_e Q$ ("P evolves to Q by emitting e") in terms of the denotational semantics and derive the expected laws for operational semantics from these. It has been published in ITP24 [2]

The overall objective of this work is to provide a formal, machine checked foundation for the laws provided by Roscoe in [4, 6].

1.2 The Global Architecture of HOL-CSP_OpSem

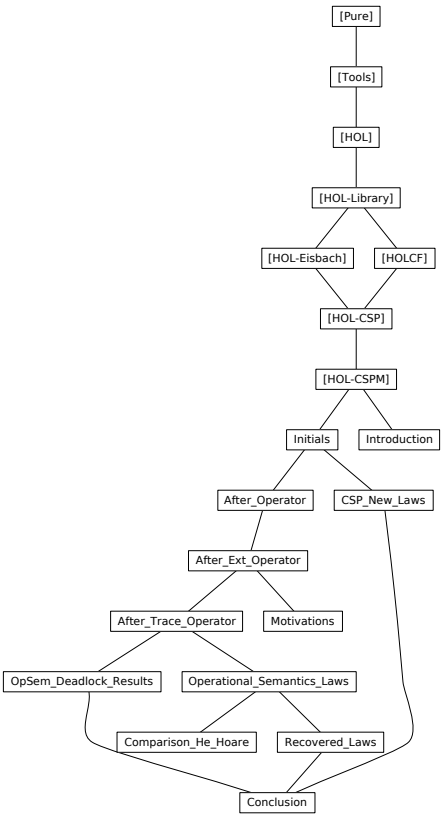

Figure 1.1: The overall architecture

The global architecture of HOL-CSP_OpSem is shown in Figure 1.1.
The package resides on:

- HOL-CSP 2.0 from the Isabelle Archive of Formal Proofs
- HOL-CSPM from the Isabelle Archive of Formal Proofs.

Chapter 2

The Initials Notion

This will be discussed more precisely later, but we want to define a new operator which would in some way be the reciprocal of the prefix operator $e \rightarrow P$.

A first observation is that by prefixing P with e, we force its nonempty traces to begin with *ev e*.

Therefore we must define a notion that captures this idea.

2.1 Definition

The initials notion captures the set of events that can be used to begin a given process.

definition *initials* :: ‹('*a*, '*r*) $process_{ptick}$ ⇒ ('*a*, '*r*) $event_{ptick}$ *set*› (‹(-^{0})› [*1000*] *999*)
  where ‹$P^0 \equiv \{e.\ [e] \in \mathcal{T}\ P\}$›

lemma *initials-memI'* : ‹$[e] \in \mathcal{T}\ P \Longrightarrow e \in P^0$›
  and *initials-memD* : ‹$e \in P^0 \Longrightarrow [e] \in \mathcal{T}\ P$›
  by (*simp-all add*: *initials-def*)

lemma *initials-def-bis*: ‹$P^0 = \{e.\ \exists s.\ e\ \#\ s \in \mathcal{T}\ P\}$›
  by (*simp add*: *set-eq-iff initials-def*)
    (*metis Prefix-Order.prefixI append-Cons append-Nil is-processT3-TR*)

lemma *initials-memI* : ‹$e\ \#\ s \in \mathcal{T}\ P \Longrightarrow e \in P^0$›
  unfolding *initials-def-bis* **by** *blast*

We say here that the *initials* of a process P is the set of events e such that there is a trace of P starting with e.

One could also think about defining P^0 as the set of events that P can not refuse at first: $\{e.\ \{e\} \notin \mathcal{R}\ P\}$. These two definitions are not equivalent (and the second one is more restrictive than the first one). Moreover, the

second does not behave well with the non-deterministic choice ($\sqcap$) (see the **notepad** below).

Therefore, we will keep the first one.

We also have a strong argument of authority: this is the definition given by Roscoe [5, p.40].

notepad
begin
fix $e :: \langle('a, 'r)\ event_{ptick}\rangle$ — just fixing $'a$ type

define *bad-initials*
where ‹*bad-initials* $(P :: ('a, 'r)\ process_{ptick}) \equiv \{e.\ \{e\} \notin \mathcal{R}\ P\}$› **for** P

— This notion is more restrictive than *initials*
have *bad-initials-subset-initials*:
‹*bad-initials* $P \subseteq$ *initials* P› **for** P
unfolding *bad-initials-def initials-def Refusals-iff*
using *is-processT1-TR is-processT5-S7* **by** *force*

— and does not behave well with *Ndet*.
have *bad-behaviour-with-Ndet*:
‹$\exists P\ Q.$ *bad-initials* $(P \sqcap Q) \neq$ *bad-initials* $P \cup$ *bad-initials* Q›
proof (*intro exI*)
show ‹*bad-initials* (*SKIP undefined* $\sqcap \bot$) $\neq$ *bad-initials* (*SKIP undefined*) $\cup$ *bad-initials* $\bot$›
by (*simp add*: *bad-initials-def F-Ndet F-SKIP F-BOT Refusals-iff*)
qed
end

2.2 Anti-Mono Rules

lemma *anti-mono-initials-T*: ‹$P \sqsubseteq_T Q \Longrightarrow$ *initials* $Q \subseteq$ *initials* P›
by (*simp add*: *Collect-mono-iff trace-refine-def initials-def subsetD*)

lemma *anti-mono-initials-F*: ‹$P \sqsubseteq_F Q \Longrightarrow$ *initials* $Q \subseteq$ *initials* P›
unfolding *failure-refine-def*
by (*drule F-subset-imp-T-subset*) (*fact anti-mono-initials-T*[*unfolded trace-refine-def*])

Of course, this anti-monotony does not hold for ($\sqsubseteq_D$).

lemma *anti-mono-initials-FD*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow$ *initials* $Q \subseteq$ *initials* P›
by (*simp add*: *anti-mono-initials-F leFD-imp-leF*)

lemma *anti-mono-initials*: ‹$P \sqsubseteq Q \Longrightarrow$ *initials* $Q \subseteq$ *initials* P›
by (*simp add*: *anti-mono-initials-T le-approx-lemma-T trace-refine-def*)

lemma *anti-mono-initials-DT*: ‹$P \sqsubseteq_{DT} Q \Longrightarrow$ *initials* $Q \subseteq$ *initials* P›
by (*simp add*: *anti-mono-initials-T leDT-imp-leT*)

2.3 Behaviour of *initials* with *STOP*, *SKIP* and $\bot$

lemma *initials-STOP* [*simp*] : ‹$STOP^0 = \{\}$›
by (*simp add*: *initials-def T-STOP*)

We already had (*?P* = *STOP*) = ($\mathcal{T}$ *?P* = $\{[]\}$). As an immediate consequence we obtain a characterization of being *STOP* involving *initials*.

lemma *initials-empty-iff-STOP*: ‹$P^0 = \{\} \longleftrightarrow P = STOP$›
proof (*intro iffI*)
{ assume ‹$\mathcal{T}\ P \neq \{[]\}$›
then obtain *a s* **where** ‹$a\ \#\ s \in \mathcal{T}\ P$›
by (*metis Nil-elem-T is-singleton-the-elem is-singletonI'*
list.exhaust-sel singletonI empty-iff)
with *initials-memD initials-memI* **have** ‹$\exists\, a.\ [a] \in \mathcal{T}\ P$› **by** *blast***}**
thus ‹$P^0 = \{\} \Longrightarrow P = STOP$›
by (*simp add*: *STOP-iff-T initials-def*) *presburger*
next
show ‹$P = STOP \Longrightarrow P^0 = \{\}$› **by** *simp*
qed

lemma *initials-SKIP* [*simp*] : ‹$(SKIP\ r)^0 = \{\checkmark(r)\}$›
by (*simp add*: *initials-def T-SKIP*)

lemma *initials-SKIPS* [*simp*] : ‹$(SKIPS\ R)^0 = tick\ `\ R$›
by (*auto simp add*: *initials-def T-SKIPS*)

lemma *initials-BOT* [*simp*] : ‹$\bot^0 = UNIV$›
by (*simp add*: *initials-def T-BOT*)

These two, on the other hand, are not characterizations.

lemma ‹$\exists\, P.\ P^0 = \{\checkmark(r)\} \wedge P \neq (SKIP\ r)$›
proof (*intro exI*)
show ‹$(STOP \sqcap SKIP\ r)^0 = \{\checkmark(r)\} \wedge STOP \sqcap SKIP\ r \neq SKIP\ r$›
by (*simp add*: *initials-def T-Ndet T-STOP T-SKIP*)
(*metis Ndet-FD-self-left SKIP-FD-iff SKIP-Neq-STOP*)
qed

lemma ‹$\exists\, P.\ P^0 = UNIV \wedge P \neq \bot$›
proof (*intro exI*)
show ‹$((\Box a \in UNIV \rightarrow \bot) \sqcap SKIPS\ UNIV)^0 = UNIV \wedge (\Box a \in UNIV \rightarrow \bot) \sqcap SKIPS\ UNIV \neq \bot$›
by (*simp add*: *set-eq-iff BOT-iff-Nil-D Ndet-projs Mprefix-projs*
SKIPS-projs initials-def) (*metis* $event_{ptick}$*.exhaust*)
qed

But when $\checkmark(r) \in P^0$, we can still have this refinement:

lemma *initial-tick-iff-FD-SKIP* : ‹$\checkmark(r) \in P^0 \longleftrightarrow P \sqsubseteq_{FD} SKIP\ r$›
proof (*intro iffI*)

show ‹✓(r) ∈ P^0 ⟹ $P \sqsubseteq_{FD}$ *SKIP r*›
unfolding *failure-divergence-refine-def failure-refine-def divergence-refine-def*
by (*simp add*: *F-SKIP D-SKIP subset-iff initials-def*)
(*metis append-Nil is-processT6-TR-notin tick-T-F*)
next
show ‹$P \sqsubseteq_{FD}$ *SKIP r* ⟹ ✓(r) ∈ P^0› **by** (*auto dest*: *anti-mono-initials-FD*)
qed

lemma *initial-ticks-iff-FD-SKIPS* : ‹*R* ≠ {} ⟹ *tick* ‘ *R* ⊆ P^0 ⟷ $P \sqsubseteq_{FD}$ *SKIPS R*›
proof (*rule iffI*)
show ‹*R* ≠ {} ⟹ *tick* ‘ *R* ⊆ P^0 ⟹ $P \sqsubseteq_{FD}$ *SKIPS R*›
by (*unfold SKIPS-def*, *rule mono-GlobalNdet-FD-const*)
(*simp-all add*: *image-subset-iff initial-tick-iff-FD-SKIP*)
next
show ‹$P \sqsubseteq_{FD}$ *SKIPS R* ⟹ *tick* ‘ *R* ⊆ P^0›
by (*metis anti-mono-initials-FD initials-SKIPS*)
qed

We also obtain characterizations for $P \,;\, Q = \bot$.

lemma *Seq-is-BOT-iff* : ‹$P \,;\, Q = \bot$ ⟷ $P = \bot \vee (\exists\, r.$ ✓(r) ∈ $P^0 \wedge Q = \bot$)›
by (*simp add*: *BOT-iff-Nil-D D-Seq initials-def*)

2.4 Behaviour of *initials* with Operators of HOL-CSP

lemma *initials-Mprefix* : ‹$(\Box a \in A \to P\ a)^0 = ev$ ‘ *A*›
and *initials-Mndetprefix* : ‹$(\sqcap a \in A \to P\ a)^0 = ev$ ‘ *A*›
and *initials-write0* : ‹$(a \to Q)^0 = \{ev\ a\}$›
and *initials-write* : ‹$(c!a \to Q)^0 = \{ev\ (c\ a)\}$›
and *initials-read* : ‹$(c?a{\in}A \to P\ a)^0 = ev$ ‘ *c* ‘ *A*›
and *initials-ndet-write* : ‹$(c!!a{\in}A \to P\ a)^0 = ev$ ‘ *c* ‘ *A*›
by (*auto simp*: *initials-def T-Mndetprefix write0-def write-def T-Mprefix read-def ndet-write-def*)

As discussed earlier, *initials* behaves very well with (□), (⊓) and (▷).

lemma *initials-Det* : ‹$(P \,\Box\, Q)^0 = P^0 \cup Q^0$›
and *initials-Ndet* : ‹$(P \sqcap Q)^0 = P^0 \cup Q^0$›
and *initials-Sliding* : ‹$(P \rhd Q)^0 = P^0 \cup Q^0$›
by (*auto simp add*: *initials-def T-Det T-Ndet T-Sliding*)

lemma *initials-Seq*:
‹$(P \,;\, Q)^0 =$ (*if* $P = \bot$ *then UNIV*
else P^0 − *range tick* ∪ $(\bigcup r{\in}\{r.$ ✓(r) ∈ $P^0\}.\ Q^0)$)›
(**is** ‹- = (*if* - *then* - *else* *?rhs*)›)
proof (*split if-split*, *intro conjI impI*)
show ‹$P = \bot$ ⟹ $(P \,;\, Q)^0$ = *UNIV*› **by** *simp*

next
show ‹$(P\ ;;\ Q)^0 = P^0 - \mathit{range}\ \mathit{tick} \cup (\bigcup r{\in}\{r.\ \checkmark(r) \in P^0\}.\ Q^0)$› **if** ‹$P \neq \bot$›
proof (*intro subset-antisym subsetI*)
fix e **assume** ‹$e \in (P\ ;;\ Q)^0$›
from $\mathit{event}_{ptick}.\mathit{exhaust}$ **consider** a **where** ‹$e = \mathit{ev}\ a$› | r **where** ‹$e = \checkmark(r)$›
by *blast*
thus ‹$e \in$ *?rhs*›
proof *cases*
from ‹$e \in (P\ ;;\ Q)^0$› **show** ‹$e = \mathit{ev}\ a \Longrightarrow e \in$ *?rhs*› **for** a
by (*simp add*: *image-iff initials-def T-Seq*)
(*metis* (*no-types*, *lifting*) *D-T append-Cons append-Nil*
is-processT3-TR-append list.exhaust-sel list.sel(*1*))
next
from ‹$e \in (P\ ;;\ Q)^0$› ‹$P \neq \bot$› **show** ‹$e = \checkmark(r) \Longrightarrow e \in$ *?rhs*› **for** r
by (*simp add*: *image-iff initials-def T-Seq BOT-iff-tick-D*)
(*metis* (*no-types*, *opaque-lifting*) *append-T-imp-tickFree append-eq-Cons-conv*
$\mathit{event}_{ptick}.\mathit{disc}(2)$ *not-Cons-self2 tickFree-Cons-iff*)
qed
next
fix e **assume** ‹$e \in$ *?rhs*›
then consider a **where** ‹$e = \mathit{ev}\ a$› ‹$\mathit{ev}\ a \in P^0$›
| r **where** ‹$\checkmark(r) \in P^0$› ‹$e \in Q^0$›
by *simp* (*metis empty-iff* $\mathit{event}_{ptick}.\mathit{exhaust}$ *rangeI*)
thus ‹$e \in (P\ ;;\ Q)^0$›
proof *cases*
show ‹$e = \mathit{ev}\ a \Longrightarrow \mathit{ev}\ a \in P^0 \Longrightarrow e \in (P\ ;;\ Q)^0$› **for** a
by (*simp add*: *initials-def T-Seq*)
next
show ‹$\checkmark(r) \in P^0 \Longrightarrow e \in Q^0 \Longrightarrow e \in (P\ ;;\ Q)^0$› **for** r
by (*simp add*: *initials-def T-Seq*) (*metis append-Nil*)
qed
qed
qed

lemma *initials-Sync*:
‹$(P\ [\![S]\!]\ Q)^0 = (\mathit{if}\ P = \bot \vee Q = \bot\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}$
$P^0 \cup Q^0 - (\mathit{range}\ \mathit{tick} \cup \mathit{ev}\ `\ S) \cup P^0 \cap Q^0 \cap (\mathit{range}\ \mathit{tick} \cup \mathit{ev}\ `\ S))$›
(**is** ‹$(P\ [\![S]\!]\ Q)^0 = (\mathit{if}\ P = \bot \vee Q = \bot\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ ?\mathit{rhs})$›)
proof (*split if-split*, *intro conjI impI*)
show ‹$P = \bot \vee Q = \bot \Longrightarrow (P\ [\![S]\!]\ Q)^0 = \mathit{UNIV}$›
by (*metis Sync-is-BOT-iff initials-BOT*)
next
show ‹$(P\ [\![S]\!]\ Q)^0 =$ *?rhs*› **if** *non-BOT* : ‹$\neg\ (P = \bot \vee Q = \bot)$›
proof (*intro subset-antisym subsetI*)
show ‹$e \in$ *?rhs* $\Longrightarrow e \in (P\ [\![S]\!]\ Q)^0$› **for** e
proof (*elim UnE*)

```
    assume ‹e ∈ P⁰ ∪ Q⁰ − (range tick ∪ ev ‘ S)›
    hence ‹[e] ∈ 𝒯 P ∨ [e] ∈ 𝒯 Q› ‹e ∉ range tick ∪ ev ‘ S›
      by (auto dest: initials-memD)
    have ‹[e] setinterleaves (([e], []), range tick ∪ ev ‘ S)›
      using ‹e ∉ range tick ∪ ev ‘ S› by simp
    with ‹[e] ∈ 𝒯 P ∨ [e] ∈ 𝒯 Q› is-processT1-TR setinterleaving-sym
    have ‹[e] ∈ 𝒯 (P ⟦S⟧ Q)› by (simp (no-asm) add: T-Sync) blast
    thus ‹e ∈ (P ⟦S⟧ Q)⁰› by (simp add: initials-memI)
  next
    assume ‹e ∈ P⁰ ∩ Q⁰ ∩ (range tick ∪ ev ‘ S)›
    hence ‹[e] ∈ 𝒯 P› ‹[e] ∈ 𝒯 Q› ‹e ∈ range tick ∪ ev ‘ S›
      by (simp-all add: initials-memD)
    have ‹[e] setinterleaves (([e], [e]), range tick ∪ ev ‘ S)›
      using ‹e ∈ range tick ∪ ev ‘ S› by simp
    with ‹[e] ∈ 𝒯 P› ‹[e] ∈ 𝒯 Q› have ‹[e] ∈ 𝒯 (P ⟦S⟧ Q)›
      by (simp (no-asm) add: T-Sync) blast
    thus ‹e ∈ (P ⟦S⟧ Q)⁰› by (simp add: initials-memI)
  qed
 next
  fix e
  assume ‹e ∈ (P ⟦S⟧ Q)⁰›
   then consider t u where ‹t ∈ 𝒯 P› ‹u ∈ 𝒯 Q› ‹[e] setinterleaves ((t, u),
range tick ∪ ev ‘ S)›
    | (div) t u r v where ‹ftF v› ‹tF r ∨ v = []› ‹[e] = r @ v›
      ‹r setinterleaves ((t, u), range tick ∪ ev ‘ S)›
      ‹t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 P›
    by (simp add: initials-def T-Sync) blast
  thus ‹e ∈ ?rhs›
  proof cases
    show ‹t ∈ 𝒯 P ⟹ u ∈ 𝒯 Q ⟹ [e] setinterleaves ((t, u), range tick ∪ ev ‘
S)
          ⟹ e ∈ ?rhs› for t u
      by (cases t; cases u; simp add: initials-def image-iff split: if-split-asm)
        (use empty-setinterleaving setinterleaving-sym in blast)+
  next
    case div
     have ‹r ≠ []› using div(4, 5) BOT-iff-Nil-D empty-setinterleaving that by
blast
    hence ‹r = [e] ∧ v = []›
     by (metis (no-types, lifting) div(3) Nil-is-append-conv append-eq-Cons-conv)
    also from div(5) BOT-iff-Nil-D non-BOT
    obtain e' t' where ‹t = e' # t'› by (cases t) blast+
    ultimately show ‹e ∈ ?rhs›
      using div(4, 5)
        by (cases u, simp-all add: initials-def subset-iff T-Sync image-iff split:
if-split-asm)
     (metis [[metis-verbose = false]] D-T setinterleaving-sym empty-setinterleaving)+
  qed
 qed
```

qed

lemma *initials-Renaming*:
‹(*Renaming P f g*)^0 = (*if P* = ⊥ *then UNIV else map-event*$_{ptick}$ *f g* ‘ *P*^{0})›
proof (*split if-split*, *intro conjI impI*)
show ‹*P* = ⊥ ⟹ (*Renaming P f g*)^0 = *UNIV*› **by** *simp*
next
assume ‹*P* ≠ ⊥›
hence ‹[] ∉ $\mathcal{D}$ *P*› **by** (*simp add*: *BOT-iff-Nil-D*)
show ‹(*Renaming P f g*)^0 = *map-event*$_{ptick}$ *f g* ‘ *P*^{0}›
proof (*intro subset-antisym subsetI*)
fix *e* **assume** ‹*e* ∈ (*Renaming P f g*)^0›
with ‹[] ∉ $\mathcal{D}$ *P*› **obtain** *s* **where** ∗ : ‹[*e*] = *map* (*map-event*$_{ptick}$ *f g*) *s*› ‹*s* ∈ $\mathcal{T}$ *P*›
by (*simp add*: *initials-def T-Renaming*)
(*metis D-T append.right-neutral append-is-Nil-conv list.map-disc-iff list.sel*(*3*) *tl-append2*)
from ∗(*1*) **obtain** *e*′ **where** ∗∗ : ‹*s* = [*e*′]› ‹*e* = *map-event*$_{ptick}$ *f g e*′› **by** *blast*
from ∗∗(*1*) ∗(*2*) **have** ‹*e*′ ∈ *P*^{0}› **by** (*simp add*: *initials-def*)
with ∗∗(*2*) **show** ‹*e* ∈ *map-event*$_{ptick}$ *f g* ‘ *P*^{0}› **by** *simp*
next
fix *e* **assume** ‹*e* ∈ *map-event*$_{ptick}$ *f g* ‘ *P*^{0}›
then obtain *e*′ **where** ‹*e* = *map-event*$_{ptick}$ *f g e*′› ‹*e*′ ∈ *P*^{0}› **by** *fast*
thus ‹*e* ∈ (*Renaming P f g*)^0› **by** (*auto simp add*: *initials-def T-Renaming*)
qed
qed

Because for the expression of its traces (and more specifically of its divergences), dealing with (\) is much more difficult.

We start with two characterizations:

- the first one to understand $P \setminus S = \bot$
- the other one to understand $[e] \in \mathcal{D}\ (P \setminus S)$.

lemma *Hiding-is-BOT-iff* :
‹*P* \ *S* = ⊥ ⟷ (∃ *t*. *set t* ⊆ *ev* ‘ *S* ∧
(*t* ∈ $\mathcal{D}$ *P* ∨ (∃ *f*. *isInfHiddenRun f P S* ∧ *t* ∈ *range f*)))›
(**is** ‹*P* \ *S* = ⊥ ⟷ *?rhs*›)
proof (*subst BOT-iff-Nil-D*, *intro iffI*)
show ‹[] ∈ $\mathcal{D}$ (*P* \ *S*) ⟹ *?rhs*›
by (*simp add*: *D-Hiding*)
(*metis* (*no-types*, *lifting*) *filter-empty-conv subsetI*)
next
assume ‹*?rhs*›
then obtain *t* **where** ∗ : ‹*set t* ⊆ *ev* ‘ *S*›

```
    ‹t ∈ 𝒟 P ∨ (∃f. isInfHiddenRun f P S ∧ t ∈ range f)› by blast
  hence ‹tickFree t ∧ [] = trace-hide t (ev ‘ S)›
    unfolding tickFree-def by (auto simp add: D-Hiding subset-iff)
  with ∗(2) show ‹[] ∈ 𝒟 (P \ S)› by (simp add: D-Hiding) metis
qed

lemma event-in-D-Hiding-iff :
  ‹[e] ∈ 𝒟 (P \ S) ⟷
   P \ S = ⊥ ∨ (∃x t. e = ev x ∧ x ∉ S ∧ [ev x] = trace-hide t (ev ‘ S) ∧
                    (t ∈ 𝒟 P ∨ (∃f. isInfHiddenRun f P S ∧ t ∈ range f)))›
  (is ‹[e] ∈ 𝒟 (P \ S) ⟷ P \ S = ⊥ ∨ ?ugly-assertion›)
proof (intro iffI)
  assume assm : ‹[e] ∈ 𝒟 (P \ S)›
  show ‹P \ S = ⊥ ∨ ?ugly-assertion›
  proof (cases e)
    fix r assume ‹e = ✓(r)›
    with assm have ‹P \ S = ⊥›
      using BOT-iff-tick-D front-tickFree-Nil is-processT9-tick by blast
    thus ‹P \ S = ⊥ ∨ ?ugly-assertion› by blast
  next
    fix x
    assume ‹e = ev x›
    with assm obtain t u
      where ∗ : ‹front-tickFree u› ‹tickFree t›
        ‹[ev x] = trace-hide t (ev ‘ S) @ u›
        ‹t ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun f P S ∧ t ∈ range f)›
      by (simp add: D-Hiding) blast
    from ∗(3) consider ‹set t ⊆ ev ‘ S› | ‹x ∉ S› ‹ev x ∈ set t›
      by (metis (no-types, lifting) Cons-eq-append-conv empty-filter-conv
          filter-eq-Cons-iff filter-is-subset image-eqI list.set-intros(1) subset-code(1))
    thus ‹P \ S = ⊥ ∨ ?ugly-assertion›
    proof cases
      assume ‹set t ⊆ ev ‘ S›
      hence ‹P \ S = ⊥› by (meson ∗(4) Hiding-is-BOT-iff)
      thus ‹P \ S = ⊥ ∨ ?ugly-assertion› by blast
    next
      assume ‹x ∉ S› ‹ev x ∈ set t›
      with ∗(3) have ‹[ev x] = trace-hide t (ev ‘ S)›
        by (induct t) (auto split: if-split-asm)
      with ∗(4) ‹e = ev x› ‹x ∉ S› have ‹?ugly-assertion› by blast
      thus ‹P \ S = ⊥ ∨ ?ugly-assertion› by blast
    qed
  qed
next
  show ‹P \ S = ⊥ ∨ ?ugly-assertion ⟹ [e] ∈ 𝒟 (P \ S)›
  proof (elim disjE)
    show ‹P \ S = ⊥ ⟹ [e] ∈ 𝒟 (P \ S)› by (simp add: D-BOT)
  next
    show ‹?ugly-assertion ⟹ [e] ∈ 𝒟 (P \ S)›
```

```
    by (elim exE conjE, simp add: D-Hiding)
      (metis Hiding-tickFree append-Cons append-Nil event_ptick.disc(1)
        front-tickFree-Nil non-tickFree-imp-not-Nil tickFree-Cons-iff)
  qed
qed
```

Now we can express $(P \setminus S)^0$. This result contains the term $P \setminus S = \bot$ that can be unfolded with *Hiding-is-BOT-iff* and the term $[ev\ x] \in \mathcal{D}\ (P \setminus S)$ that can be unfolded with *event-in-D-Hiding-iff*.

```
lemma initials-Hiding:
  ‹(P \ S)^0 = (if P \ S = ⊥ then UNIV else
              {e. case e of ev x ⇒ x ∉ S ∧ ([ev x] ∈ D (P \ S) ∨ (∃ t. [ev x] =
trace-hide t (ev ` S) ∧ (t, ev ` S) ∈ F P))
                          | ✓(r) ⇒ ∃ t. set t ⊆ ev ` S ∧ t @ [✓(r)] ∈ T P})›
  (is ‹initials (P \ S) = (if P \ S = ⊥ then UNIV else ?set)›)
proof (split if-split, intro conjI impI)
  show ‹P \ S = ⊥ ⟹ initials (P \ S) = UNIV› by simp
next
  assume non-BOT : ‹P \ S ≠ ⊥›
  show ‹initials (P \ S) = ?set›
  proof (intro subset-antisym subsetI)
    fix e
    assume initial : ‹e ∈ initials (P \ S)›
      — This implies e ∉ ev ` S with our other assumptions.
    { fix x
      assume assms : ‹x ∈ S› ‹ev x ∈ initials (P \ S)›
      then consider ‹∃ t. [ev x] = trace-hide t (ev ` S) ∧ (t, ev ` S) ∈ F P›
        | ‹∃ t u. front-tickFree u ∧ tickFree t ∧ [ev x] = trace-hide t (ev ` S) @ u ∧
              (t ∈ D P ∨ (∃ f. isInfHiddenRun f P S ∧ t ∈ range f))›
        by (simp add: initials-def T-Hiding) blast
      hence ‹P \ S = ⊥›
      proof cases
        assume ‹∃ t. [ev x] = trace-hide t (ev ` S) ∧ (t, ev ` S) ∈ F P›
        hence False by (metis Cons-eq-filterD image-eqI assms(1))
        thus ‹P \ S = ⊥› by blast
      next
        assume ‹∃ t u. front-tickFree u ∧ tickFree t ∧ [ev x] = trace-hide t (ev ` S)
@ u ∧
                    (t ∈ D P ∨ (∃ f. isInfHiddenRun f P S ∧ t ∈ range f))›
        then obtain t u
          where ∗ : ‹front-tickFree u› ‹tickFree t› ‹[ev x] = trace-hide t (ev ` S) @
u›
            ‹t ∈ D P ∨ (∃ f. isInfHiddenRun f P S ∧ t ∈ range f)› by blast
        from ∗(3) have ∗∗ : ‹set t ⊆ ev ` S›
          by (induct t) (simp-all add: assms(1) split: if-split-asm)
        from ∗(4) ∗∗ Hiding-is-BOT-iff show ‹P \ S = ⊥› by blast
      qed
    }
    with initial have ∗ : ‹e ∉ ev ` S› using non-BOT by blast
```

```
    from initial consider ‹[e] ∈ 𝒟 (P \ S)›
      | ‹∃ t. [e] = trace-hide t (ev ‘ S) ∧ (t, ev ‘ S) ∈ ℱ P›
      unfolding initials-def by (simp add: T-Hiding D-Hiding) blast
    thus ‹e ∈ ?set›
    proof cases
      assume assm : ‹[e] ∈ 𝒟 (P \ S)›
      then obtain x where ‹e = ev x›
          by (metis BOT-iff-Nil-D append-Nil event_ptick.exhaust non-BOT pro-
cess-charn)
      with assm * show ‹e ∈ ?set› by (simp add: image-iff)
    next
      assume ‹∃ t. [e] = trace-hide t (ev ‘ S) ∧ (t, ev ‘ S) ∈ ℱ P›
      then obtain t where ** : ‹[e] = trace-hide t (ev ‘ S)›
        ‹(t, ev ‘ S) ∈ ℱ P› by blast
      thus ‹e ∈ ?set›
      proof (cases e)
        have ‹e = ✓(r) ⟹ set (butlast t) ⊆ ev ‘ S ∧ butlast t @ [✓(r)] ∈ 𝒯 P›
for r
          using ** by (cases t rule: rev-cases; simp add: F-T empty-filter-conv
subset-eq split: if-split-asm)
        (metis F-T Hiding-tickFree append-T-imp-tickFree neq-Nil-conv non-tickFree-tick)
        thus ‹e = ✓(r) ⟹ e ∈ ?set› for r by auto
      next
        fix x
        assume ‹e = ev x›
        with * have ‹x ∉ S› by blast
        with ‹e = ev x› **(1) **(2) show ‹e ∈ ?set› by auto
      qed
    qed
  next
    fix e
    assume ‹e ∈ ?set›
    then consider r where ‹e = ✓(r)› ‹∃ t. set t ⊆ ev ‘ S ∧ t @ [✓(r)] ∈ 𝒯 P›
      | a where ‹e = ev a› ‹a ∉ S›
        ‹[ev a] ∈ 𝒟 (P \ S) ∨
              (∃ t. [ev a] = trace-hide t (ev ‘ S) ∧ (t, ev ‘ S) ∈ ℱ P)› by (cases e)
simp-all
    thus ‹e ∈ initials (P \ S)›
    proof cases
      fix r assume assms : ‹e = ✓(r)› ‹∃ t. set t ⊆ ev ‘ S ∧ t @ [✓(r)] ∈ 𝒯 P›
      from assms(2) obtain t
        where * : ‹set t ⊆ ev ‘ S› ‹t @ [✓(r)] ∈ 𝒯 P› by blast
      have ** : ‹[e] = trace-hide (t @ [✓(r)]) (ev ‘ S) ∧ (t @ [✓(r)], ev ‘ S) ∈ ℱ P›
        using *(1) by (simp add: assms(1) image-iff tick-T-F[OF *(2)] subset-iff)
      show ‹e ∈ initials (P \ S)›
        unfolding initials-def by (simp add: T-Hiding) (use ** in blast)
    next
      show ‹[ev a] ∈ 𝒟 (P \ S) ∨
```

```
        (∃ t. [ev a] = trace-hide t (ev ‘ S) ∧ (t, ev ‘ S) ∈ ℱ P) ⟹ e ∈ (P \ S)⁰›
      if ‹e = ev a› for a
    proof (elim exE conjE disjE)
      show ‹[ev a] ∈ 𝒟 (P \ S) ⟹ e ∈ (P \ S)⁰›
        by (simp add: ‹e = ev a› D-T initials-memI')
    next
      show ‹[ev a] = trace-hide t (ev ‘ S) ⟹ (t, ev ‘ S) ∈ ℱ P ⟹ e ∈ (P \
S)⁰› for t
        by (metis F-T initials-memI' mem-T-imp-mem-T-Hiding ‹e = ev a›)
    qed
  qed
 qed
qed
```

In the end the result would look something like this:

```
(P \ S)⁰ = (if ∃ t. set t ⊆ ev ‘ S ∧ (t ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun f P S
∧ t ∈ range f)) then UNIV else {e. case e of ev x ⇒ x ∉ S ∧ (((∃ t. set t ⊆
ev ‘ S ∧ (t ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun f P S ∧ t ∈ range f))) ∨ (∃ xa t.
ev x = ev xa ∧ xa ∉ S ∧ [ev xa] = trace-hide t (ev ‘ S) ∧ (t ∈ 𝒟 P ∨ (∃ f.
isInfHiddenRun f P S ∧ t ∈ range f)))) ∨ (∃ t. [ev x] = trace-hide t (ev ‘
S) ∧ (t, ev ‘ S) ∈ ℱ P)) | ✓(r) ⇒ ∃ t. set t ⊆ ev ‘ S ∧ t @ [✓(r)] ∈ 𝒯 P})
```

Obviously, it is not very easy to use. We will therefore rely more on the corollaries below.

```
corollary initial-tick-Hiding-iff :
  ‹✓(r) ∈ (P \ B)⁰ ⟷ P \ B = ⊥ ∨ (∃ t. set t ⊆ ev ‘ B ∧ t @ [✓(r)] ∈ 𝒯 P)›
  by (simp add: initials-Hiding)

corollary initial-tick-imp-initial-tick-Hiding:
  ‹✓(r) ∈ P⁰ ⟹ ✓(r) ∈ (P \ B)⁰›
  by (subst initials-Hiding, simp add: initials-def)
    (metis append-Nil empty-iff empty-set subset-iff)


corollary initial-inside-Hiding-iff :
  ‹e ∈ S ⟹ ev e ∈ (P \ S)⁰ ⟷ P \ S = ⊥›
  by (simp add: initials-Hiding)


corollary initial-notin-Hiding-iff :
  ‹e ∉ S ⟹ ev e ∈ (P \ S)⁰ ⟷
   P \ S = ⊥ ∨
   (∃ t. [ev e] = trace-hide t (ev ‘ S) ∧
        (t ∈ 𝒟 P ∨ (∃ f. isInfHiddenRun f P S ∧ t ∈ range f) ∨ (t, ev ‘ S) ∈ ℱ
P))›
  by (auto simp add: initials-Hiding event-in-D-Hiding-iff split: if-split-asm)
```

corollary *initial-notin-imp-initial-Hiding*:
‹*ev e* ∈ (*P* \ *S*)⁰› **if** *initial* : ‹*ev e* ∈ *P*⁰› **and** *notin* : ‹*e* ∉ *S*›
proof −
from *inf-hidden*[*of S* ‹[*ev e*]› *P*]
consider *f* **where** ‹*isInfHiddenRun f P S*› ‹[*ev e*] ∈ *range f*›
| *t* **where** ‹[*ev e*] = *trace-hide t* (*ev* ‘ *S*)› ‹(*t*, *ev* ‘ *S*) ∈ ℱ *P*›
by (*simp add*: *initials-Hiding image-iff*[*of* ‹*ev e*›] *notin*)
(*metis mem-Collect-eq initial initials-def*)
thus ‹*ev e* ∈ *initials* (*P* \ *S*)›
proof *cases*
show ‹*ev e* ∈ (*P* \ *S*)⁰› **if** ‹*isInfHiddenRun f P S*› ‹[*ev e*] ∈ *range f*› **for** *f*
proof (*intro initials-memI*[*of* ‹*ev e*› ‹[]›] *D-T*)
from *that* **show** ‹[*ev e*] ∈ 𝒟 (*P* \ *S*)›
by (*simp add*: *event-in-D-Hiding-iff image-iff*[*of* ‹*ev e*›] *notin*)
(*metis* (*no-types*, *lifting*) *event*$_{ptick}$.*inject*(1) *filter.simps image-iff notin*)
qed
next
show ‹[*ev e*] = *trace-hide t* (*ev* ‘ *S*) ⟹ (*t*, *ev* ‘ *S*) ∈ ℱ *P* ⟹ *ev e* ∈ (*P* \ *S*)⁰› **for** *t*
by (*auto simp add*: *initials-Hiding notin*)
qed
qed

2.5 Behaviour of *initials* with Operators of `HOL-CSPM`

lemma *initials-GlobalDet*:
‹(□*a* ∈ *A*. *P a*)⁰ = (⋃ *a* ∈ *A*. *initials* (*P a*))›
by (*auto simp add*: *initials-def T-GlobalDet*)

lemma *initials-GlobalNdet*:
‹(⊓*a* ∈ *A*. *P a*)⁰ = (⋃ *a* ∈ *A*. *initials* (*P a*))›
by (*auto simp add*: *initials-def T-GlobalNdet*)

lemma *initials-MultiSync*:
‹*initials* (⟦*S*⟧ *m* ∈# *M*. *P m*) =
(*if M* = {#} *then* {}
else if ∃ *m* ∈# *M*. *P m* = ⊥ *then UNIV*
else if ∃ *m*. *M* = {#*m*#} *then initials* (*P* (*THE m*. *M* = {#*m*#}))
else {*e*. ∃ *m* ∈# *M*. *e* ∈ *initials* (*P m*) − (*range tick* ∪ *ev* ‘ *S*)} ∪
{*e* ∈ *range tick* ∪ *ev* ‘ *S*. ∀ *m* ∈# *M*. *e* ∈ *initials* (*P m*)})›
proof −
have ∗ : ‹*initials* (⟦*S*⟧ *m* ∈# (*M* + {#*a*, *a*′#}). *P m*) =
{*e*. ∃ *m* ∈# *M* + {#*a*, *a*′#}. *e* ∈ *initials* (*P m*) − (*range tick* ∪ *ev* ‘ *S*)}
∪

```
        {e ∈ range tick ∪ ev ‘ S. ∀ m ∈# M + {#a, a′#}. e ∈ initials (P m)}›
    if non-BOT : ‹∀ m ∈# M + {#a, a′#}. P m ≠ ⊥› for a a′ M
  proof (induct M rule: msubset-induct′[OF subset-mset.refl])
    case 1
    then show ?case by (auto simp add: non-BOT initials-Sync)
  next
    case (2 a′′ M′)
    have ∗ : ‹MultiSync S (add-mset a′′ M′ + {#a, a′#}) P =
          P a′′ ⟦S⟧ (MultiSync S (M′ + {#a, a′#}) P)›
      by (simp add: add-mset-commute)
    have ∗∗ : ‹¬ (P a′′ = ⊥ ∨ MultiSync S (M′ + {#a, a′#}) P = ⊥)›
      using 2.hyps(1, 2) in-diffD non-BOT
      by (auto simp add: MultiSync-is-BOT-iff Sync-is-BOT-iff, fastforce, meson
mset-subset-eqD)
    show ?case
      by (auto simp only: ∗ initials-Sync ∗∗ 2.hyps(3), auto)
  qed

  show ?thesis
  proof (cases ‹∃ m ∈# M. P m = ⊥›)
    show ‹∃ m ∈# M. P m = ⊥ ⟹ ?thesis›
      by simp (metis MultiSync-BOT-absorb initials-BOT)
  next
    show ‹¬ (∃ m∈#M. P m = ⊥) ⟹ ?thesis›
    proof (cases ‹∃ a a′ M′. M = M′ + {#a, a′#}›)
      assume assms : ‹¬ (∃ m∈#M. P m = ⊥)› ‹∃ a a′ M′. M = M′ + {#a, a′#}›
      from assms(2) obtain a a′ M′ where ‹M = M′ + {#a, a′#}› by blast
      with ∗ assms(1) show ?thesis by simp
    next
      assume ‹∄ a a′ M′. M = M′ + {#a, a′#}›
      hence ‹M = {#} ∨ (∃ m. M = {#m#})›
        by (metis add.right-neutral multiset-cases union-mset-add-mset-right)
      thus ?thesis by auto
    qed
  qed
qed



lemma initials-Throw : ‹(P Θ a ∈ A. Q a)0 = P0›
proof (cases ‹P = ⊥›)
  show ‹P = ⊥ ⟹ (P Θ a ∈ A. Q a)0 = P0› by simp
next
  show ‹(P Θ a ∈ A. Q a)0 = P0› if ‹P ≠ ⊥›
  proof (intro subset-antisym subsetI)
    from ‹P ≠ ⊥› show ‹e ∈ (P Θ a ∈ A. Q a)0 ⟹ e ∈ P0› for e
    by (auto simp add: T-Throw D-T is-processT7 Cons-eq-append-conv BOT-iff-Nil-D
        intro!: initials-memI′ dest!: initials-memD)
  next
```

show ‹$e \in P^0 \Longrightarrow e \in (P\ \Theta\ a \in A.\ Q\ a)^0$› **for** e
by (*auto simp add*: *initials-memD T-Throw Cons-eq-append-conv*
intro!: *initials-memI'*)
qed
qed

lemma *initials-Interrupt*: ‹$(P \triangle Q)^0 = P^0 \cup Q^0$›
proof (*intro subset-antisym subsetI*)
show ‹$e \in (P \triangle Q)^0 \Longrightarrow e \in P^0 \cup Q^0$› **for** e
by (*auto simp add*: *T-Interrupt Cons-eq-append-conv*
dest!: *initials-memD intro*: *initials-memI*)
next
show ‹$e \in P^0 \cup Q^0 \Longrightarrow e \in (P \triangle Q)^0$› **for** e
by (*force simp add*: *initials-def T-Interrupt*)
qed

2.6 Behaviour of *initials* with Reference Processes

lemma *initials-DF*: ‹$(DF\ A)^0 = ev\ `\ A$›
by (*subst DF-unfold*) (*simp add*: *initials-Mndetprefix*)

lemma *initials-DF$_{SKIPS}$*: ‹$(DF_{SKIPS}\ A\ R)^0 = ev\ `\ A \cup tick\ `\ R$›
by (*subst DF$_{SKIPS}$-unfold*)
(*simp add*: *initials-Mndetprefix initials-Ndet*)

lemma *initials-RUN*: ‹$(RUN\ A)^0 = ev\ `\ A$›
by (*subst RUN-unfold*) (*simp add*: *initials-Mprefix*)

lemma *initials-CHAOS*: ‹$(CHAOS\ A)^0 = ev\ `\ A$›
by (*subst CHAOS-unfold*)
(*simp add*: *initials-Mprefix initials-Ndet*)

lemma *initials-CHAOS$_{SKIPS}$*: ‹$(CHAOS_{SKIPS}\ A\ R)^0 = ev\ `\ A \cup tick\ `\ R$›
by (*subst CHAOS$_{SKIPS}$-unfold*)
(*auto simp add*: *initials-Mprefix initials-Ndet*)

lemma *empty-ev-initials-iff-empty-events-of* :
‹$\{a.\ ev\ a \in P^0\} = \{\} \longleftrightarrow \alpha(P) = \{\}$›
proof (*rule iffI*)
show ‹$\alpha(P) = \{\} \Longrightarrow \{a.\ ev\ a \in P^0\} = \{\}$›
by (*auto simp add*: *initials-def events-of-def*)
next
show ‹$\alpha(P) = \{\}$› **if** ‹$\{a.\ ev\ a \in P^0\} = \{\}$›
proof (*rule ccontr*)
assume ‹$\alpha(P) \neq \{\}$›
then obtain $a\ t$ **where** ‹$t \in \mathcal{T}\ P$› ‹$ev\ a \in set\ t$›
by (*meson equals0I events-of-memD*)

from ‹$t \in \mathcal{T}\ P$› **consider** ‹$t = []$› | r **where** ‹$t = [\checkmark(r)]$› | $b\ t'$ **where** ‹$t = ev\ b \#\ t'$›
by (*metis T-imp-front-tickFree* ‹*ev a* ∈ *set t*› *front-tickFree-Cons-iff is-ev-def list.distinct*(*1*) *list.set-cases*)
thus *False*
proof *cases*
from ‹*ev a* ∈ *set t*› **show** ‹$t = [] \Longrightarrow$ *False*› **by** *simp*
next
from ‹*ev a* ∈ *set t*› **show** ‹$t = [\checkmark(r)] \Longrightarrow$ *False*› **for** r **by** *simp*
next
fix $b\ t'$ **assume** ‹$t = ev\ b \#\ t'$›
with ‹$t \in \mathcal{T}\ P$› **have** ‹$b \in \{a.\ ev\ a \in P^0\}$› **by** (*simp add*: *initials-memI*)
with ‹$\{a.\ ev\ a \in P^0\} = \{\}$› **show** *False* **by** *simp*
qed
qed
qed

2.7 Properties of *initials* related to continuity

We prove here some properties that we will need later in continuity or admissibility proofs.

lemma *initials-LUB*:
‹*chain Y* $\Longrightarrow$ $(\bigsqcup i.\ Y\ i)^0 = (\bigcap P \in (range\ Y).\ P^0)$›
unfolding *initials-def* **by** (*auto simp add*: *limproc-is-thelub T-LUB*)

lemma *adm-in-F*: ‹*cont u* $\Longrightarrow$ *adm* $(\lambda x.\ (s,\ X) \in \mathcal{F}\ (u\ x))$›
by (*simp add*: *adm-def cont2contlubE limproc-is-thelub ch2ch-cont F-LUB*)

lemma *adm-in-D*: ‹*cont u* $\Longrightarrow$ *adm* $(\lambda x.\ s \in \mathcal{D}\ (u\ x))$›
by (*simp add*: *D-LUB-2 adm-def ch2ch-cont cont2contlubE limproc-is-thelub*)

lemma *adm-in-T*: ‹*cont u* $\Longrightarrow$ *adm* $(\lambda x.\ s \in \mathcal{T}\ (u\ x))$›
by (*fold T-F-spec*) (*fact adm-in-F*)

lemma *initial-adm*[*simp*] : ‹*cont u* $\Longrightarrow$ *adm* $(\lambda x.\ e \in (u\ x)^0)$›
unfolding *initials-def* **by** (*simp add*: *adm-in-T*)

Chapter 3

Construction of the After Operator

Now that we have defined P^0, we can talk about what happens to P after an event belonging to this set.

3.1 Definition

We want to define a new operator on a process P which would in some way be the reciprocal of the prefix operator $a \rightarrow P$.

The intuitive way of doing so is to only keep the tails of the traces beginning by *ev a* (and similar for failures and divergences). However we have an issue if $ev\ a \notin P^0$ i.e. if no trace of P begins with *ev a*: the result would no longer verify the invariant *is-process* because its trace set would be empty. We must therefore distinguish this case.

In the previous version, we agreed to get *STOP* after an event *ev a* that was not in the *initials* of P. But even if its repercussions were minimal, this choice seemed a little artificial and arbitrary. In this new version we use a placeholder instead: Ψ. When $ev\ a \in P^0$ we use our intuitive definition, and $ev\ a \notin P^0$ we define P after a being equal to $\Psi\ P\ a$.

For the moment we have no additional assumption on Ψ.

locale *After* =
  fixes Ψ :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
begin

lift-definition *After* :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$› (**infixl** ‹*after*› *86*)
  is ‹$\lambda P\ a.$ *if* $ev\ a \in P^0$

then ({(*t*, *X*). (*ev a* # *t*, *X*) ∈ $\mathcal{F}$ *P*},
{ *t* . *ev a* # *t* ∈ $\mathcal{D}$ *P*})
else ($\mathcal{F}$ (Ψ *P a*), $\mathcal{D}$ (Ψ *P a*))›
proof −
show ‹*?thesis P a*› (**is** ‹*is-process* (*if - then* (*?f*, *?d*) *else -*)›) **for** *P a*
proof (*split if-split*, *intro conjI impI*)
show ‹*is-process* ($\mathcal{F}$ (Ψ *P a*), $\mathcal{D}$ (Ψ *P a*))›
by (*metis CollectD Divergences.rep-eq Failures.rep-eq process*$_0$*-of-process process-surj-pair*)
next
show ‹*is-process* (*?f*, *?d*)› **if** ‹*ev a* ∈ P^0›
unfolding *is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv*
proof (*intro conjI impI allI*)
from ‹*ev a* ∈ P^0› **show** ‹([], {}) ∈ *?f*› **by** (*simp add*: *initials-memD T-F-spec*)
next
show ‹(*t*, *X*) ∈ *?f* ⟹ *ftF t*› **for** *t X*
by *simp* (*metis front-tickFree-Cons-iff front-tickFree-Nil is-processT2*)
next
show ‹(*t* @ *u*, {}) ∈ *?f* ⟹ (*t*, {}) ∈ *?f*› **for** *t u* **by** (*simp add*: *is-processT3*)
next
from *is-processT4* **show** ‹(*t*, *Y*) ∈ *?f* ∧ *X* ⊆ *Y* ⟹ (*t*, *X*) ∈ *?f*› **for** *t X Y*
by *blast*
next
from ‹*ev a* ∈ P^0› **show** ‹(*t*, *X*) ∈ *?f* ∧ (∀ *e*. *e* ∈ *Y* ⟶ (*t* @ [*e*], {}) ∉ *?f*)
⟹ (*t*, *X* ∪ *Y*) ∈ *?f*› **for** *t X Y*
by (*simp add*: *initials-memD is-processT5*)
next
show ‹(*t* @ [✓(*r*)], {}) ∈ *?f* ⟹ (*t*, *X* − {✓(*r*)}) ∈ *?f*› **for** *t r X*
by (*simp add*: *is-processT6*)
next
from *is-processT7* **show** ‹*t* ∈ *?d* ∧ *tF t* ∧ *ftF u* ⟹ *t* @ *u* ∈ *?d*› **for** *t u* **by** *force*
next
from *D-F* **show** ‹*t* ∈ *?d* ⟹ (*t*, *X*) ∈ *?f*› **for** *t X* **by** *blast*
next
show ‹*t* @ [✓(*r*)] ∈ *?d* ⟹ *t* ∈ *?d*› **for** *t r*
by *simp* (*metis append-Cons process-charn*)
qed
qed
qed

3.2 Projections

lemma *F-After* :
‹$\mathcal{F}$ (*P after a*) = (*if ev a* ∈ P^0 *then* {(*t*, *X*). (*ev a* # *t*, *X*) ∈ $\mathcal{F}$ *P*} *else* $\mathcal{F}$ (Ψ *P a*))›
by (*simp add*: *Failures-def After.rep-eq FAILURES-def*)

lemma *D-After* :

‹$\mathcal{D}$ (P *after* a) = (*if* $ev\ a \in P^0$ *then* {s. $ev\ a$ # $s \in \mathcal{D}\ P$} *else* $\mathcal{D}$ ($\Psi\ P\ a$))›
by (*simp add*: *Divergences-def After.rep-eq DIVERGENCES-def*)

lemma *T-After* :
‹$\mathcal{T}$ (P *after* a) = (*if* $ev\ a \in P^0$ *then* {s. $ev\ a$ # $s \in \mathcal{T}\ P$} *else* $\mathcal{T}$ ($\Psi\ P\ a$))›
by (*auto simp add*: *T-F-spec*[*symmetric*] *F-After*)

lemmas *After-projs* = *F-After D-After T-After*

lemma *not-initial-After* : ‹$ev\ a \notin P^0 \Longrightarrow P$ *after* $a = \Psi\ P\ a$›
by (*simp add*: *After.abs-eq Process-spec*)

lemma *initials-After* :
‹(P *after* a)^0 = (*if* $ev\ a \in P^0$ *then* {e. $ev\ a$ # $[e] \in \mathcal{T}\ P$} *else* ($\Psi\ P\ a$)^0)›
by (*simp add*: *T-After initials-def*)

3.3 Monotony

lemma *mono-After* : ‹P *after* $a \sqsubseteq Q$ *after* a›
if ‹$P \sqsubseteq Q$› **and** ‹$ev\ a \notin Q^0 \Longrightarrow \Psi\ P\ a \sqsubseteq \Psi\ Q\ a$›
proof (*subst le-approx-def*, *safe*)
from *that*(*1*)[*THEN anti-mono-initials*] *that*(*1*)[*THEN le-approx2T*] *that*[*THEN le-approx1*]
show ‹$t \in \mathcal{D}$ (Q *after* a) $\Longrightarrow t \in \mathcal{D}$ (P *after* a)› **for** t
by (*simp add*: *D-After initials-def subset-iff split*: *if-split-asm*)
(*metis D-imp-front-tickFree append-Cons append-Nil butlast.simps*(*2*) $event_{ptick}$.*disc*(*1*) *is-processT7 last.simps tickFree-Nil tickFree-butlast*)
next
from *that*(*1*)[*THEN anti-mono-initials*] *that*(*2*)[*THEN le-approx2*] *that*(*1*)[*THEN proc-ord2a*]
show ‹$t \notin \mathcal{D}$ (P *after* a) $\Longrightarrow X \in \mathcal{R}_a$ (P *after* a) $t \Longrightarrow X \in \mathcal{R}_a$ (Q *after* a) t›
for $t\ X$
by (*simp add*: *Refusals-after-def After-projs initials-def subset-iff split*: *if-split-asm*)
(*metis initials-memI F-T initials-def mem-Collect-eq*, *blast*)
next
from *that*(*1*)[*THEN anti-mono-initials*] *that*[*THEN le-approx2*] *that*(*1*)[*THEN le-approx2T*]
show ‹$t \notin \mathcal{D}$ (P *after* a) $\Longrightarrow X \in \mathcal{R}_a$ (Q *after* a) $t \Longrightarrow X \in \mathcal{R}_a$ (P *after* a) t›
for $t\ X$
by (*simp add*: *Refusals-after-def After-projs initials-def subset-iff split*: *if-split-asm*)
(*metis F-imp-front-tickFree append-Cons append-Nil butlast.simps*(*2*)
$event_{ptick}$.*disc*(*1*) *is-processT7 last.simps tickFree-Nil tickFree-butlast*)
next
show ‹$t \in$ *min-elems* ($\mathcal{D}$ (P *after* a)) $\Longrightarrow t \in \mathcal{T}$ (Q *after* a)› **for** t
proof (*cases* ‹$P = \bot$›)
assume ‹$t \in$ *min-elems* ($\mathcal{D}$ (P *after* a))› **and** ‹$P = \bot$›
hence ‹$t = []$›
by (*simp add*: *BOT-iff-Nil-D D-After D-BOT min-elems-def*)

```
      (metis append-butlast-last-id front-tickFree-single nil-less2)
   thus ‹t ∈ 𝒯 (Q after a)› by simp
 next
   from that(1)[THEN anti-mono-initials] that(1)[THEN le-approx2T]
     that(2)[THEN le-approx3] min-elems6[OF - - that(1)]
   show ‹t ∈ min-elems (𝒟 (P after a)) ⟹ P ≠ ⊥ ⟹ t ∈ 𝒯 (Q after a)›
   apply (auto simp add: min-elems-def After-projs BOT-iff-Nil-D split: if-split-asm)
     by (metis T-F-spec append-butlast-last-id butlast.simps(2) less-self)
      (metis (mono-tags, lifting) T-F-spec append-Nil initials-def mem-Collect-eq)
 qed
qed


lemma mono-After-T : ‹P ⊑_T Q ⟹ P after a ⊑_T Q after a›
 and mono-After-F : ‹P ⊑_F Q ⟹ P after a ⊑_F Q after a›
 and mono-After-D : ‹P ⊑_D Q ⟹ P after a ⊑_D Q after a›
 and mono-After-FD : ‹P ⊑_FD Q ⟹ P after a ⊑_FD Q after a›
 and mono-After-DT : ‹P ⊑_DT Q ⟹ P after a ⊑_DT Q after a›
 if ‹ev a ∈ Q⁰›
 using that F-subset-imp-T-subset[of Q P] D-T[of ‹ev a # -› P]
 unfolding failure-divergence-refine-def trace-divergence-refine-def
   failure-refine-def divergence-refine-def trace-refine-def
 by (auto simp add: After-projs initials-def)
   (metis initials-memI in-mono mem-Collect-eq initials-def)

lemmas monos-After = mono-After mono-After-FD mono-After-DT
 mono-After-F mono-After-D mono-After-T
```

3.4 Behaviour of *After* with *STOP*, *SKIP* and ⊥

```
lemma After-STOP : ‹STOP after a = Ψ STOP a›
 by (simp add: Process-eq-spec After-projs)

lemma After-SKIP : ‹SKIP r after a = Ψ (SKIP r) a›
 by (simp add: Process-eq-spec After-projs)

lemma After-BOT : ‹⊥ after a = ⊥›
 by (force simp add: BOT-iff-Nil-D D-After D-BOT)

lemma After-is-BOT-iff :
 ‹P after a = ⊥ ⟷ (if ev a ∈ P⁰ then [ev a] ∈ 𝒟 P else Ψ P a = ⊥)›
 using hd-Cons-tl by (force simp add: BOT-iff-Nil-D D-After initials-def D-T)
```

3.5 Behaviour of *After* with Operators of HOL-CSP

In future theories, we will need to know how *After* behaves with other operators of CSP. More specifically, we want to know how *After* can be "distributed" over a sequential composition, a synchronization, etc.

In some way, we are looking for reversing the "step-laws" (laws of distributivity of *Mprefix* over other operators). Given the difficulty in establishing these results in `HOL-CSP` and `HOL-CSPM`, one can easily imagine that proving *After* versions will require a lot of work.

3.5.1 Loss of Determinism

A first interesting observation is that the *After* operator leads to the loss of determinism.

lemma *After-Mprefix-is-After-Mndetprefix*:
‹$a \in A \Longrightarrow (\Box a \in A \rightarrow P\ a)$ *after* $a = (\sqcap a \in A \rightarrow P\ a)$ *after* a›
by (*auto simp add*: *Process-eq-spec initials-Mprefix initials-Mndetprefix After-projs Mprefix-projs Mndetprefix-projs*)

lemma *After-Det-is-After-Ndet* :
‹*ev* $a \in P^0 \cup Q^0 \Longrightarrow (P \Box Q)$ *after* $a = (P \sqcap Q)$ *after* a›
by (*auto simp add*: *Process-eq-spec initials-Det initials-Ndet After-projs Det-projs Ndet-projs*)

lemma *After-Sliding-is-After-Ndet* :
‹*ev* $a \in P^0 \cup Q^0 \Longrightarrow (P \rhd Q)$ *after* $a = (P \sqcap Q)$ *after* a›
by (*auto simp add*: *Process-eq-spec initials-Sliding initials-Ndet After-projs Sliding-projs Ndet-projs*)

lemma *After-Ndet*:
‹$(P \sqcap Q)$ *after* $a =$
(*if ev* $a \in P^0 \cap Q^0$ *then* P *after* $a \sqcap Q$ *after* a
else if ev $a \in P^0$ *then* P *after* a
else if ev $a \in Q^0$ *then* Q *after* a
else $\Psi\ (P \sqcap Q)\ a$)› **for** $P\ Q :: ‹('a, 'r)\ process_{ptick}›$
proof −
{ fix $P\ Q :: ‹('a, 'r)\ process_{ptick}›$
assume ‹*ev* $a \in P^0$› ‹*ev* $a \notin Q^0$›
hence ‹$(P \sqcap Q)$ *after* $a = P$ *after* a›
by (*simp add*: *Process-eq-spec After-projs Ndet-projs initials-Ndet*)
(*meson initials-memI D-T F-T*)
}
moreover have ‹*ev* $a \notin P^0 \cup Q^0 \Longrightarrow (P \sqcap Q)$ *after* $a = \Psi\ (P \sqcap Q)\ a$›
by (*simp add*: *Process-eq-spec After-projs initials-Ndet*)
moreover have ‹*ev* $a \in P^0 \cap Q^0 \Longrightarrow$ $(P \sqcap Q)$ *after* $a = P$ *after* $a \sqcap Q$ *after* a›
by (*auto simp add*: *Process-eq-spec After-projs Ndet-projs initials-Ndet*)
ultimately show *?thesis* **by** (*metis Int-iff Ndet-commute Un-iff*)
qed

lemma *After-Det*:
‹$(P \Box Q)$ *after* $a =$
(*if ev* $a \in P^0 \cap Q^0$ *then* P *after* $a \sqcap Q$ *after* a

else *if ev a* $\in P^0$ *then P after a*
else *if ev a* $\in Q^0$ *then Q after a*
else Ψ $(P \,\Box\, Q)$ *a*)›
by (*simp add*: *After-Det-is-After-Ndet After-Ndet not-initial-After initials-Det*)

lemma *After-Sliding*:
‹$(P \rhd Q)$ *after a* =
(*if ev a* $\in P^0 \cap Q^0$ *then P after a* $\sqcap$ *Q after a*
else *if ev a* $\in P^0$ *then P after a*
else *if ev a* $\in Q^0$ *then Q after a*
else Ψ $(P \rhd Q)$ *a*)›
by (*simp add*: *After-Ndet After-Sliding-is-After-Ndet initials-Sliding not-initial-After*)

lemma *After-Mprefix*:
‹$(\Box\, a \in A \to P\ a)$ *after a* = (*if a* $\in A$ *then P a else* Ψ $(\Box\, a \in A \to P\ a)$ *a*)›
by (*simp add*: *Process-eq-spec After-projs initials-Mprefix*
Mprefix-projs image-iff)

lemma *After-Mndetprefix*:
‹$(\sqcap\, a \in A \to P\ a)$ *after a* = (*if a* $\in A$ *then P a else* Ψ $(\sqcap\, a \in A \to P\ a)$ *a*)›
by (*metis* (*full-types*) *After-Mprefix After-Mprefix-is-After-Mndetprefix*
$event_{ptick}$.*inject*(*1*) *imageE not-initial-After initials-Mndetprefix*)

corollary *After-write0* : ‹$(a \to P)$ *after b* = (*if b* = *a then P else* Ψ $(a \to P)$ *b*)›
by (*simp add*: *write0-def After-Mprefix*)

lemma ‹$(a \to P)$ *after a* = *P*› **by** (*simp add*: *After-write0*)

This result justifies seeing P *after* a as the reciprocal operator of the prefix $a \to P$.

However, we lose information with *After*: in general, $a \to P$ *after* $a \neq P$ (even when *ev* $a \in P^0$ and $P \neq \bot$).

lemma ‹$\exists\, P.\ a \to P$ *after* $a \neq P$›
proof (*intro exI*)
show ‹*a* $\to$ *SKIP undefined after a* $\neq$ *SKIP undefined*› **by** *simp*
qed

lemma ‹$\exists\, P.$ *ev* $a \in P^0 \wedge a \to P$ *after* $a \neq P$›
by (*metis UNIV-I initials-BOT write0-neq-BOT*)

lemma ‹$\exists\, P.$ *ev* $a \in P^0 \wedge P \neq \bot \wedge a \to P$ *after* $a \neq P$›
proof (*intro exI*)
define *P* :: ‹('*a*, '*r*) $process_{ptick}$› **where** *P-def*: ‹*P* = (*a* $\to$ *STOP*) $\Box$ *SKIP undefined*›
have $*$: ‹*ev* $a \in P^0$› **by** (*simp add*: *P-def initials-Det initials-write0*)
moreover have ‹$P \neq \bot$› **by** (*simp add*: *Det-is-BOT-iff P-def*)
moreover have ‹*a* $\to$ *P after a* = *a* $\to$ *STOP*›
by (*rule arg-cong*[**where** *f* = ‹$\lambda P.\ a \to P$›])

(*simp add*: *P-def After-Det initials-write0 After-write0*)
moreover have ‹$a \rightarrow STOP \neq P$›
by (*rule contrapos-nn*[*of* ‹$(a \rightarrow STOP)^0 = P^0$› ‹$a \rightarrow STOP = P$›])
(*auto simp add*: *P-def initials-Det initials-write0*)
ultimately show ‹$ev\ a \in P^0 \wedge P \neq \bot \wedge a \rightarrow P\ after\ a \neq P$› **by** *presburger*
qed

corollary *After-write* : ‹$(c!a \rightarrow P)\ after\ b = (if\ b = c\ a\ then\ P\ else\ \Psi\ (c!a \rightarrow P)\ b)$›
by (*simp add*: *write-def After-Mprefix*)

corollary *After-read* :
‹$(c?a{\in}A \rightarrow P\ a)\ after\ b = (if\ b \in c\ `\ A\ then\ P\ (inv\text{-}into\ A\ c\ b)\ else\ \Psi\ (c?a{\in}A \rightarrow P\ a)\ b)$›
by (*simp add*: *read-def After-Mprefix*)

corollary *After-ndet-write* :
‹$(c!!a{\in}A \rightarrow P\ a)\ after\ b = (if\ b \in c\ `\ A\ then\ P\ (inv\text{-}into\ A\ c\ b)\ else\ \Psi\ (c!!a{\in}A \rightarrow P\ a)\ b)$›
by (*simp add*: *ndet-write-def After-Mndetprefix*)

3.5.2 *After* Sequential Composition

The first goal is to obtain an equivalent of $a \rightarrow P\ ;\ Q = a \rightarrow (P\ ;\ Q)$. But in order to be exhaustive we also have to consider the possibility of Q taking the lead when $\checkmark(r) \in P^0$ in the sequential composition $P\ ;\ Q$.

lemma *not-skippable-or-not-initialR-After-Seq*:
‹$(P\ ;\ Q)\ after\ a = (if\ ev\ a \in P^0\ then\ P\ after\ a\ ;\ Q\ else\ \Psi\ (P\ ;\ Q)\ a)$›
if ‹$range\ tick \cap P^0 = \{\} \vee (\forall r.\ \checkmark(r) \in P^0 \longrightarrow ev\ a \notin Q^0)$›
proof (*cases* ‹$P = \bot$›)
show ‹$P = \bot \Longrightarrow$
$(P\ ;\ Q)\ after\ a =$
$(if\ ev\ a \in P^0\ then\ P\ after\ a\ ;\ Q\ else\ \Psi\ (P\ ;\ Q)\ a)$›
by (*simp add*: *After-BOT*)
next
assume *non-BOT*: ‹$P \neq \bot$›
with *that* **have** $: ‹$ev\ a \in (P\ ;\ Q)^0 \longleftrightarrow ev\ a \in P^0$›
by (*auto simp add*: *initials-Seq*)
show ‹$(P\ ;\ Q)\ after\ a =$
$(if\ ev\ a \in P^0\ then\ P\ after\ a\ ;\ Q\ else\ \Psi\ (P\ ;\ Q)\ a)$›
proof (*split if-split*, *intro conjI impI*)
from $ **show** ‹$ev\ a \notin P^0 \Longrightarrow (P\ ;\ Q)\ after\ a = \Psi\ (P\ ;\ Q)\ a$›
by (*simp add*: *not-initial-After*)
next
assume *initial-P* : ‹$ev\ a \in P^0$›
show ‹$(P\ ;\ Q)\ after\ a = P\ after\ a\ ;\ Q$›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*

assume ‹$s \in \mathcal{D}\ ((P\ ;;\ Q)\ after\ a)$›
hence $*$: ‹$ev\ a\ \#\ s \in \mathcal{D}\ (P\ ;;\ Q)$›
by (*simp add*: *D-After* $ *initial-P split*: *if-split-asm*)
then consider ‹$ev\ a\ \#\ s \in \mathcal{D}\ P$›
| *t1 t2 r* **where** ‹$ev\ a\ \#\ s = t1\ @\ t2$› ‹$t1\ @\ [\checkmark(r)] \in \mathcal{T}\ P$› ‹$t2 \in \mathcal{D}\ Q$›
by (*simp add*: *D-Seq*) *blast*
thus ‹$s \in \mathcal{D}\ (P\ after\ a\ ;;\ Q)$›
proof *cases*
show ‹$ev\ a\ \#\ s \in \mathcal{D}\ P \Longrightarrow s \in \mathcal{D}\ (P\ after\ a\ ;;\ Q)$›
by (*simp add*: *D-Seq D-After initial-P*)
next
fix *t1 t2 r* **assume** $**$: ‹$ev\ a\ \#\ s = t1\ @\ t2$› ‹$t1\ @\ [\checkmark(r)] \in \mathcal{T}\ P$› ‹$t2 \in \mathcal{D}\ Q$›
have ‹$t1 \neq []$› **by** (*metis* $**$ *initials-memI D-T append-self-conv2 disjoint-iff rangeI that*)
with $**(1)$ **obtain** *t1′*
where ‹$t1 = ev\ a\ \#\ t1'$› ‹$s = t1' @ t2$› **by** (*metis Cons-eq-append-conv*)
with $**(2, 3)$ **show** ‹$s \in \mathcal{D}\ (P\ after\ a\ ;;\ Q)$›
by (*simp add*: *D-Seq T-After*) (*blast intro*: *initial-P*)
qed
next
fix *s*
assume ‹$s \in \mathcal{D}\ (P\ after\ a;;\ Q)$›
hence ‹$ev\ a\ \#\ s \in \mathcal{D}\ P \vee (\exists\, t1\ t2\ r.\ s = t1\ @\ t2 \wedge ev\ a\ \#\ t1\ @\ [\checkmark(r)] \in \mathcal{T}\ P \wedge t2 \in \mathcal{D}\ Q)$›
by (*simp add*: *D-Seq After-projs initial-P*)
hence ‹$ev\ a\ \#\ s \in \mathcal{D}\ (P\ ;;\ Q)$›
by (*elim disjE*; *simp add*: *D-Seq*) (*metis append-Cons*)
thus ‹$s \in \mathcal{D}\ ((P\ ;;\ Q)\ after\ a)$›
by (*simp add*: *D-After* $ *initial-P*)
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}\ ((P\ ;;\ Q)\ after\ a) = \mathcal{D}\ (P\ after\ a\ ;;\ Q)$›
assume ‹$(s, X) \in \mathcal{F}\ ((P\ ;;\ Q)\ after\ a)$›
then consider ‹$ev\ a \in (P\ ;;\ Q)^0$› ‹$(ev\ a\ \#\ s, X) \in \mathcal{F}\ (P\ ;;\ Q)$›
| ‹$ev\ a \notin (P\ ;;\ Q)^0$› ‹$s = []$› **by** (*simp add*: *F-After* $ *initial-P*)
thus ‹$(s, X) \in \mathcal{F}\ (P\ after\ a\ ;;\ Q)$›
proof *cases*
assume *assms* : ‹$ev\ a \in (P\ ;;\ Q)^0$› ‹$(ev\ a\ \#\ s, X) \in \mathcal{F}\ (P\ ;;\ Q)$›
from *assms(2)* **consider** ‹$ev\ a\ \#\ s \in \mathcal{D}\ (P\ ;;\ Q)$›
| ‹$(ev\ a\ \#\ s, X \cup range\ tick) \in \mathcal{F}\ P$› ‹$tF\ s$›
| *t1 t2 r* **where** ‹$ev\ a\ \#\ s = t1\ @\ t2$› ‹$t1\ @\ [\checkmark(r)] \in \mathcal{T}\ P$› ‹$(t2, X) \in \mathcal{F}\ Q$›
by (*simp add*: *F-Seq D-Seq*) *blast*
thus ‹$(s, X) \in \mathcal{F}\ (P\ after\ a\ ;;\ Q)$›
proof *cases*
assume ‹$ev\ a\ \#\ s \in \mathcal{D}\ (P\ ;;\ Q)$›
hence ‹$s \in \mathcal{D}\ ((P\ ;;\ Q)\ after\ a)$›
by (*simp add*: *D-After assms(1)*)

with *same-div D-F* **show** ‹$(s,\ X) \in \mathcal{F}\ (P\ after\ a\ ;;\ Q)$› **by** *blast*
next
show ‹$(ev\ a\ \#\ s,\ X \cup range\ tick) \in \mathcal{F}\ P \Longrightarrow tF\ s \Longrightarrow (s,\ X) \in \mathcal{F}\ (P\ after\ a\ ;;\ Q)$›
by (*simp add*: *F-Seq F-After initial-P*)
next
fix *t1 t2 r* **assume** $*$: ‹$ev\ a\ \#\ s = t1\ @\ t2$› ‹$t1\ @\ [\checkmark(r)] \in \mathcal{T}\ P$› ‹$(t2,\ X) \in \mathcal{F}\ Q$›
have ‹$t1 \neq []$› **by** (*metis* $*$ *initials-memI F-T disjoint-iff rangeI self-append-conv2 that*)
with $*(1)$ **obtain** *t1'*
where ‹$t1 = ev\ a\ \#\ t1'$› ‹$s = t1'\ @\ t2$› **by** (*metis Cons-eq-append-conv*)
with $*(2,\ 3)$ **show** ‹$(s,\ X) \in \mathcal{F}\ (P\ after\ a\ ;;\ Q)$›
by (*simp add*: *F-Seq T-After*) (*blast intro*: *initial-P*)
qed
next
show ‹$ev\ a \notin (P\ ;;\ Q)^0 \Longrightarrow s = [] \Longrightarrow (s,\ X) \in \mathcal{F}\ (P\ after\ a\ ;;\ Q)$›
by (*simp add*: *F-Seq F-After* $ *initial-P*)
qed
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}\ ((P\ ;;\ Q)\ after\ a) = \mathcal{D}\ (P\ after\ a\ ;;\ Q)$›
assume ‹$(s,\ X) \in \mathcal{F}\ (P\ after\ a\ ;;\ Q)$›
then consider ‹$s \in \mathcal{D}\ (P\ after\ a\ ;;\ Q)$›
| ‹$(s,\ X \cup range\ tick) \in \mathcal{F}\ (P\ after\ a)$› ‹$tF\ s$›
| *t1 t2 r* **where** ‹$s = t1\ @\ t2$› ‹$t1\ @\ [\checkmark(r)] \in \mathcal{T}\ (P\ after\ a)$› ‹$(t2,\ X) \in \mathcal{F}\ Q$›
by (*simp add*: *F-Seq D-Seq*) *blast*
thus ‹$(s,\ X) \in \mathcal{F}\ ((P\ ;;\ Q)\ after\ a)$›
proof *cases*
from *same-div D-F* **show** ‹$s \in \mathcal{D}\ (P\ after\ a\ ;;\ Q) \Longrightarrow (s,\ X) \in \mathcal{F}\ ((P\ ;;\ Q)\ after\ a)$› **by** *blast*
next
show ‹$(s,\ X \cup range\ tick) \in \mathcal{F}\ (P\ after\ a) \Longrightarrow tF\ s \Longrightarrow (s,\ X) \in \mathcal{F}\ ((P\ ;;\ Q)\ after\ a)$›
by (*simp add*: *F-After* $ *initial-P F-Seq*)
next
fix *t1 t2 r* **assume** $*$: ‹$s = t1\ @\ t2$› ‹$t1\ @\ [\checkmark(r)] \in \mathcal{T}\ (P\ after\ a)$› ‹$(t2,\ X) \in \mathcal{F}\ Q$›
from $*(2)$ **have** ‹$ev\ a\ \#\ t1\ @\ [\checkmark(r)] \in \mathcal{T}\ P$› **by** (*simp add*: *T-After initial-P*)
with $*(1,\ 3)$ **show** ‹$(s,\ X) \in \mathcal{F}\ ((P\ ;;\ Q)\ after\ a)$›
by (*simp add*: *F-After* $ *initial-P F-Seq*) (*metis append-Cons*)
qed
qed
qed
qed

lemma *skippable-not-initialL-After-Seq*:

```
  ‹(P ; Q) after a = (  if (∃ r. ✓(r) ∈ P⁰) ∧ ev a ∈ Q⁰
                      then Q after a else Ψ (P ; Q) a)›
  (is ‹(P ; Q) after a = (if ?prem then ?rhs else -)›) if ‹ev a ∉ P⁰›
proof −
  from initials-BOT ‹ev a ∉ P⁰› have non-BOT : ‹P ≠ ⊥› by blast
  with ‹ev a ∉ P⁰› have $ : ‹ev a ∈ (P ; Q)⁰ ⟷ ?prem›
    by (auto simp add: initials-Seq)
  show ‹(P ; Q) after a = (if ?prem then ?rhs else Ψ (P ; Q) a)›
  proof (split if-split, intro conjI impI)
    show ‹¬ ?prem ⟹ (P ; Q) after a = Ψ (P ; Q) a›
      by (rule not-initial-After, use $ in blast)
  next
    show ‹(P ; Q) after a = ?rhs› if ?prem
    proof (subst Process-eq-spec, safe)
      fix t
      assume ‹t ∈ 𝒟 ((P ; Q) after a)›
      with ‹?prem› have ‹ev a # t ∈ 𝒟 (P ; Q)› by (simp add: D-After $)
       with ‹ev a ∉ P⁰› obtain t1 t2 r where ∗ : ‹ev a # t = t1 @ t2› ‹t1 @
[✓(r)] ∈ 𝒯 P› ‹t2 ∈ 𝒟 Q›
        by (simp add: D-Seq) (meson initials-memI D-T)
      from ∗(1, 2) ‹ev a ∉ P⁰› initials-memI have ‹t1 = []›
        by (cases t1; simp) blast
      with ∗ show ‹t ∈ 𝒟 ?rhs›
        by (auto simp add: D-After intro: initials-memI D-T)
    next
      from ‹?prem› show ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ((P ; Q) after a)› for s
        by (simp add: D-After $ D-Seq split: if-split-asm)
          (metis append-Nil initials-memD)
    next
      fix t X
      assume ‹(t, X) ∈ ℱ ((P ; Q) after a)›
      hence ‹ev a ∈ (P ; Q)⁰› ‹(ev a # t, X) ∈ ℱ (P ; Q)›
        by (simp-all add: F-After $ split: if-split-asm) (use ‹?prem› in blast)+
      thus ‹(t, X) ∈ ℱ ?rhs›
        by (simp add: F-After F-Seq $)
          (metis (no-types, lifting) initials-memI D-T F-T Nil-is-append-conv
            append-self-conv2 hd-append2 list.exhaust-sel list.sel(1) ‹ev a ∉ P⁰›)
    next
      show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ((P ; Q) after a)› for s X
        by (simp add: F-After $ F-Seq split: if-split-asm)
          (metis append-Nil initials-memD ‹?prem›, use ‹?prem› in blast)
    qed
  qed
qed


lemma skippable-initialL-initialR-After-Seq:
  ‹(P ; Q) after a = (P after a ; Q) ⊓ Q after a›
  (is ‹(P ; Q) after a = (P after a ; Q) ⊓ ?rhs›)
```

```
  if assms : ‹(∃ r. ✓(r) ∈ P⁰) ∧ ev a ∈ Q⁰› ‹ev a ∈ P⁰›
proof (cases ‹P = ⊥›)
  show ‹P = ⊥ ⟹ (P ; Q) after a = (P after a ; Q) ⊓ ?rhs›
    by (simp add: After-BOT)
next
  show ‹(P ; Q) after a = (P after a ; Q) ⊓ ?rhs› if ‹P ≠ ⊥›
  proof (subst Process-eq-spec-optimized, safe)
    fix s
    assume ‹s ∈ 𝒟 ((P ; Q) after a)›
    hence ‹ev a # s ∈ 𝒟 (P ; Q)›
      by (simp add: D-After initials-Seq ‹P ≠ ⊥› ‹ev a ∈ P⁰› image-iff)
    then consider ‹ev a # s ∈ 𝒟 P›
      | t1 t2 r where ‹ev a # s = t1 @ t2› ‹t1 @ [✓(r)] ∈ 𝒯 P› ‹t2 ∈ 𝒟 Q›
      by (simp add: D-Seq) blast
    thus ‹s ∈ 𝒟 ((P after a ; Q) ⊓ ?rhs)›
    proof cases
      show ‹ev a # s ∈ 𝒟 P ⟹ s ∈ 𝒟 ((P after a ; Q) ⊓ ?rhs)›
        by (simp add: D-After D-Seq D-Ndet ‹P ≠ ⊥› assms(2))
    next
     fix t1 t2 r assume ∗ : ‹ev a # s = t1 @ t2› ‹t1 @ [✓(r)] ∈ 𝒯 P› ‹t2 ∈ 𝒟 Q›
      show ‹s ∈ 𝒟 ((P after a ; Q) ⊓ ?rhs)›
      proof (cases ‹t1 = []›)
        from ∗(1, 3) assms(1) show ‹t1 = [] ⟹ s ∈ 𝒟 ((P after a ; Q) ⊓ ?rhs)›
          by (simp add: D-After D-Ndet)
      next
        assume ‹t1 ≠ []›
        with ∗(1, 3) obtain t1' where ‹t1 = ev a # t1'› by (cases t1; simp)
        with ∗ show ‹s ∈ 𝒟 ((P after a ; Q) ⊓ ?rhs)›
          by (simp add: T-After D-Seq D-Ndet assms(2)) blast
      qed
    qed
  next
    fix s
    assume ‹s ∈ 𝒟 ((P after a ; Q) ⊓ ?rhs)›
    then consider ‹s ∈ 𝒟 (P after a ; Q)› | ‹s ∈ 𝒟 ?rhs›
      by (simp add: D-Ndet) blast
    thus ‹s ∈ 𝒟 ((P ; Q) after a)›
    proof cases
      show ‹s ∈ 𝒟 (P after a ; Q) ⟹ s ∈ 𝒟 ((P ; Q) after a)›
        by (simp add: After-projs D-Seq initials-Seq ‹P ≠ ⊥› assms(2) image-iff)
          (metis append-Cons)
    next
      from assms show ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ((P ; Q) after a)›
        by (simp add: D-After D-Seq initials-Seq ‹P ≠ ⊥› split: if-split-asm)
          (metis append-Nil initials-def mem-Collect-eq)
    qed
  next
    fix s X
    assume same-div : ‹𝒟 ((P ; Q) after a) = 𝒟 ((P after a ; Q) ⊓ ?rhs)›
```

assume ‹$(s, X) \in \mathcal{F}\ ((P\ ;;\ Q)\ after\ a)$›
hence ‹$(ev\ a\ \#\ s, X) \in \mathcal{F}\ (P\ ;;\ Q)$›
by (*simp add*: *F-After initials-Seq* ‹$P \neq \bot$› *assms*(*2*) *image-iff*)
then consider ‹$ev\ a\ \#\ s \in \mathcal{D}\ P$›
| ‹$(ev\ a\ \#\ s, X \cup range\ tick) \in \mathcal{F}\ P$› ‹$tF\ s$›
| *t1 t2 r* **where** ‹$ev\ a\ \#\ s = t1\ @\ t2$› ‹$t1\ @\ [\checkmark(r)] \in \mathcal{T}\ P$› ‹$(t2, X) \in \mathcal{F}\ Q$›
by (*simp add*: *F-Seq D-Seq*) *blast*
thus ‹$(s, X) \in \mathcal{F}\ ((P\ after\ a\ ;;\ Q) \sqcap ?rhs)$›
proof *cases*
assume ‹$ev\ a\ \#\ s \in \mathcal{D}\ P$›
hence ‹$s \in \mathcal{D}\ (P\ after\ a\ ;;\ Q)$›
by (*simp add*: *D-After D-Seq assms*(*2*))
with *same-div D-Ndet D-F* **show** ‹$(s, X) \in \mathcal{F}\ ((P\ after\ a\ ;;\ Q) \sqcap ?rhs)$› **by** *blast*
next
show ‹$(ev\ a\ \#\ s, X \cup range\ tick) \in \mathcal{F}\ P \Longrightarrow tF\ s \Longrightarrow$
$(s, X) \in \mathcal{F}\ ((P\ after\ a\ ;;\ Q) \sqcap ?rhs)$›
by (*simp add*: *F-Ndet F-Seq F-After assms*(*2*))
next
fix *t1 t2 r* **assume** $*$: ‹$ev\ a\ \#\ s = t1\ @\ t2$› ‹$t1\ @\ [\checkmark(r)] \in \mathcal{T}\ P$› ‹$(t2, X) \in \mathcal{F}\ Q$›
show ‹$(s, X) \in \mathcal{F}\ ((P\ after\ a\ ;;\ Q) \sqcap ?rhs)$›
proof (*cases* ‹$t1 = []$›)
from $*$(*1*, *3*) *assms* **show** ‹$t1 = [] \Longrightarrow (s, X) \in \mathcal{F}\ ((P\ after\ a\ ;;\ Q) \sqcap ?rhs)$›
by (*simp add*: *F-Ndet F-Seq F-After assms*(*2*))
next
assume ‹$t1 \neq []$›
with $*$(*1*, *3*) **obtain** *t1'* **where** ‹$t1 = ev\ a\ \#\ t1'$› **by** (*cases t1*; *simp*)
with $*$ **show** ‹$(s, X) \in \mathcal{F}\ ((P\ after\ a\ ;;\ Q) \sqcap ?rhs)$›
by (*simp add*: *F-Ndet F-Seq F-After T-After assms*(*2*)) *blast*
qed
qed
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}\ ((P\ ;;\ Q)\ after\ a) = \mathcal{D}\ ((P\ after\ a\ ;;\ Q) \sqcap ?rhs)$›
assume ‹$(s, X) \in \mathcal{F}\ ((P\ after\ a\ ;;\ Q) \sqcap ?rhs)$›
then consider ‹$(s, X) \in \mathcal{F}\ (P\ after\ a\ ;;\ Q)$› | ‹$(s, X) \in \mathcal{F}\ ?rhs$›
by (*simp add*: *F-Ndet*) *blast*
thus ‹$(s, X) \in \mathcal{F}\ ((P\ ;;\ Q)\ after\ a)$›
proof *cases*
assume ‹$(s, X) \in \mathcal{F}\ (P\ after\ a\ ;;\ Q)$›
then consider ‹$s \in \mathcal{D}\ (P\ after\ a\ ;;\ Q)$›
| ‹$(s, X \cup range\ tick) \in \mathcal{F}\ (P\ after\ a)$› ‹$tF\ s$›
| *t1 t2 r* **where** ‹$s = t1\ @\ t2$› ‹$t1\ @\ [\checkmark(r)] \in \mathcal{T}\ (P\ after\ a)$› ‹$(t2, X) \in \mathcal{F}\ Q$›
by (*simp add*: *F-Seq D-Seq*) *blast*
thus ‹$(s, X) \in \mathcal{F}\ ((P\ ;;\ Q)\ after\ a)$›
proof *cases*
show ‹$s \in \mathcal{D}\ (P\ after\ a\ ;;\ Q) \Longrightarrow (s, X) \in \mathcal{F}\ ((P\ ;;\ Q)\ after\ a)$›

using *same-div D-Ndet D-F* **by** *blast*
next
show ‹(*s*, *X* ∪ *range tick*) ∈ $\mathcal{F}$ (*P after a*) ⟹ *tF s* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ ((*P* ; *Q*) *after a*)›
by (*simp add*: *F-After F-Seq initials-Seq assms*(*2*) *image-iff*)
next
show ‹*s* = *t1* @ *t2* ⟹ *t1* @ [✓(*r*)] ∈ $\mathcal{T}$ (*P after a*) ⟹ (*t2*, *X*) ∈ $\mathcal{F}$ *Q*
⟹ (*s*, *X*) ∈ $\mathcal{F}$ ((*P* ; *Q*) *after a*)› **for** *t1 t2 r*
by (*simp add*: *F-After T-After F-Seq initials-Seq*
assms(*2*) *image-iff* ‹*P* ≠ ⊥›) (*metis Cons-eq-appendI*)
qed
next
from *assms*(*1*) ‹*P* ≠ ⊥› **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ ((*P* ; *Q*) *after a*)›
by (*simp add*: *F-GlobalNdet F-After F-Seq initials-Seq image-iff split*: *if-split-asm*)
(*simp add*: *initials-def*, *metis append-Nil*)
qed
qed
qed

lemma *not-initialL-not-skippable-or-not-initialR-After-Seq*:
‹*ev a* ∉ P^0 ⟹ *range tick* ∩ P^0 = {} ∨ (∀ *r*. *tick r* ∈ P^0 ⟶ *ev a* ∉ Q^0) ⟹
(*P* ; *Q*) *after a* = Ψ (*P* ; *Q*) *a*›
by (*simp add*: *not-skippable-or-not-initialR-After-Seq not-initial-After*)

lemma *After-Seq*:
‹(*P* ; *Q*) *after a* =
(*if range tick* ∩ P^0 = {} ∨ (∀ *r*. ✓(*r*) ∈ P^0 ⟶ *ev a* ∉ Q^0)
then *if ev a* ∈ P^0 *then P after a* ; *Q else* Ψ (*P* ; *Q*) *a*
else *if ev a* ∈ P^0
then (*P after a* ; *Q*) ⊓ *Q after a*
else Q after a)›
by (*simp add*: *not-skippable-or-not-initialR-After-Seq*
skippable-initialL-initialR-After-Seq skippable-not-initialL-After-Seq)

3.5.3 *After* Synchronization

Now let's focus on *Sync*. We want to obtain an equivalent of
Mprefix ?A ?P ⟦*?S*⟧ *Mprefix ?B ?Q* = (□*a*∈(*?A* − *?S*) → (*?P a* ⟦*?S*⟧ *Mprefix ?B ?Q*)) □ (□*b*∈(*?B* − *?S*) → (*Mprefix ?A ?P* ⟦*?S*⟧ *?Q b*)) □ (□*x*∈(*?A* ∩ *?B* ∩ *?S*) → (*?P x* ⟦*?S*⟧ *?Q x*))
We will also divide the task.

After version of
⟦*a* ∉ *?S*; *?B* ⊆ *?S*⟧ ⟹ *a* → *P* ⟦*?S*⟧ *Mprefix ?B ?Q* = *a* → (*P* ⟦*?S*⟧ *Mprefix*

?B ?Q).

lemma *initialL-not-initialR-not-in-After-Sync*:
‹(*P* ⟦*S*⟧ *Q*) *after a* = *P after a* ⟦*S*⟧ *Q*›
if *initial-hyps*: ‹*ev a* ∈ *P*⁰› ‹*ev a* ∉ *Q*⁰› **and** *notin*: ‹*a* ∉ *S*›
proof (*subst Process-eq-spec-optimized*, *safe*)
{ fix *s X*
assume *assms* : ‹*P* ≠ ⊥› ‹*Q* ≠ ⊥› ‹(*ev a* # *s*, *X*) ∈ ℱ (*P* ⟦*S*⟧ *Q*)›
and *same-div* : ‹𝒟 ((*P* ⟦*S*⟧ *Q*) *after a*) = 𝒟 (*P after a* ⟦*S*⟧ *Q*)›
have ‹(*s*, *X*) ∈ ℱ (*P after a* ⟦*S*⟧ *Q*)›
proof (*cases* ‹*ev a* # *s* ∈ 𝒟 (*P* ⟦*S*⟧ *Q*)›)
case *True*
hence ‹*s* ∈ 𝒟 ((*P* ⟦*S*⟧ *Q*) *after a*)›
by (*force simp add*: *D-After initials-Sync initial-hyps*(*1*) *assms*(*1*, *2*) *notin*)
with *D-F same-div* **show** ‹(*s*, *X*) ∈ ℱ (*P after a* ⟦*S*⟧ *Q*)› **by** *blast*
next
case *False*
with *assms*(*3*) **obtain** *s-P s-Q X-P X-Q*
where ∗ : ‹(*s-P*, *X-P*) ∈ ℱ *P*› ‹(*s-Q*, *X-Q*) ∈ ℱ *Q*›
‹(*ev a* # *s*) *setinterleaves* ((*s-P*, *s-Q*), *range tick* ∪ *ev* ‘ *S*)›
‹*X* = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
by (*simp add*: *F-Sync D-Sync*) *blast*
have ∗∗ : ‹*s-P* ≠ [] ∧ *hd s-P* = *ev a* ∧ *s setinterleaves* ((*tl s-P*, *s-Q*), *range tick* ∪ *ev* ‘ *S*)›
using ∗(*3*) **apply** (*cases s-P*; *cases s-Q*; *simp add*: *notin image-iff split*: *if-split-asm*)
using ∗(*2*) *initials-memI F-T initial-hyps*(*2*) **by** *blast*+
hence ‹(*tl s-P*, *X-P*) ∈ ℱ (*P after a*) ∧ (*s-Q*, *X-Q*) ∈ ℱ *Q* ∧
s setinterleaves ((*tl s-P*, *s-Q*), *range tick* ∪ *ev* ‘ *S*) ∧
X = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
by (*simp add*: *F-After* ∗∗ *initial-hyps*(*1*))
(*metis* ∗(*1*) ∗(*2*) ∗(*4*) ∗∗ *list.exhaust-sel*)
thus ‹(*s*, *X*) ∈ ℱ (*P after a* ⟦*S*⟧ *Q*)›
by (*simp add*: *F-Sync*) *blast*
qed
} note ∗ = *this*

show ‹𝒟 ((*P* ⟦*S*⟧ *Q*) *after a*) = 𝒟 (*P after a* ⟦*S*⟧ *Q*) ⟹
(*s*, *X*) ∈ ℱ ((*P* ⟦*S*⟧ *Q*) *after a*) ⟹ (*s*, *X*) ∈ ℱ (*P after a* ⟦*S*⟧ *Q*)› **for** *s X*
by (*simp add*: ∗ *F-After initials-Sync initial-hyps notin image-iff split*: *if-split-asm*)
(*metis After-BOT BOT-Sync CollectI D-BOT F-imp-front-tickFree Sync-BOT*
front-tickFree-Cons-iff front-tickFree-Nil is-processT8)
next

fix *s X*
assume *same-div* : ‹𝒟 ((*P* ⟦*S*⟧ *Q*) *after a*) = 𝒟 (*P after a* ⟦*S*⟧ *Q*)›
{ assume *assms* : ‹*P* ≠ ⊥› ‹*Q* ≠ ⊥› ‹(*s*, *X*) ∈ ℱ (*P after a* ⟦*S*⟧ *Q*)›
from *assms*(*3*) **consider**
‹∃ *s-P s-Q X-P X-Q*. (*s-P*, *X-P*) ∈ ℱ (*P after a*) ∧ (*s-Q*, *X-Q*) ∈ ℱ *Q* ∧
s setinterleaves ((*s-P*, *s-Q*), *range tick* ∪ *ev* ‘ *S*) ∧

```
                    X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ S) ∪ X-P ∩ X-Q› |
    ‹s ∈ 𝒟 (P after a ⟦S⟧ Q)›
    by (simp add: F-Sync D-Sync) blast
  hence ‹(ev a # s, X) ∈ ℱ (P ⟦S⟧ Q)›
  proof cases
    case 1
    then obtain s-P s-Q X-P X-Q
      where ∗ : ‹(ev a # s-P, X-P) ∈ ℱ P› ‹(s-Q, X-Q) ∈ ℱ Q›
        ‹s setinterleaves ((s-P, s-Q), range tick ∪ ev ‘ S)›
        ‹X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ S) ∪ X-P ∩ X-Q›
      by (simp add: F-After initial-hyps(1)) blast
    have ‹(s-Q, X-Q) ∈ ℱ Q ∧
          (ev a # s) setinterleaves ((ev a # s-P, s-Q), range tick ∪ ev ‘ S) ∧
           X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ S) ∪ X-P ∩ X-Q›
      apply (simp add: ∗(2, 4))
      using ∗(3) by (cases s-Q; simp add: notin image-iff)
    with ∗(1) show ‹(ev a # s, X) ∈ ℱ (P ⟦S⟧ Q)›
      by (simp add: F-Sync) blast
  next
    case 2
    from 2[simplified same-div[symmetric]]
    have ‹ev a # s ∈ 𝒟 (P ⟦S⟧ Q)›
      by (simp add: D-After initial-hyps(1) initials-Sync assms(1, 2) image-iff
notin)
    with D-F show ‹(ev a # s, X) ∈ ℱ (P ⟦S⟧ Q)› by blast
  qed
 }
 thus ‹(s, X) ∈ ℱ (P after a ⟦S⟧ Q) ⟹ (s, X) ∈ ℱ ((P ⟦S⟧ Q) after a)›
  by (simp add: F-After initials-Sync initial-hyps After-BOT F-BOT image-iff
     notin F-imp-front-tickFree front-tickFree-Cons-iff)
next

 { fix s
  assume assms : ‹P ≠ ⊥› ‹Q ≠ ⊥› ‹ev a # s ∈ 𝒟 (P ⟦S⟧ Q)›
  from assms(3) obtain t u r v
    where ∗ : ‹ftF v› ‹tF  r ∨ v = []› ‹ev a # s = r @ v›
      ‹r setinterleaves ((t, u), range tick ∪ ev ‘ S)›
      ‹t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 P› by (simp add: D-Sync) blast
  have ∗∗ : ‹r ≠ [] ∧ hd r = ev a›
  by (metis ∗(3, 4, 5) BOT-iff-Nil-D assms(1, 2) empty-setinterleaving hd-append
list.sel(1))
  hence ∗∗∗ : ‹(t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ⟶ t ≠ [] ∧ hd t = ev a ∧
                      tl r setinterleaves ((tl t, u), range tick ∪ ev ‘ S)) ∧
            (t ∈ 𝒟 Q ∧ u ∈ 𝒯 P ⟶ u ≠ [] ∧ hd u = ev a ∧
                      tl r setinterleaves ((t, tl u), range tick ∪ ev ‘ S))›
    using ∗(4) assms(1, 2)[simplified BOT-iff-Nil-D] initial-hyps[simplified ini-
tials-def] notin
    by (cases t; cases u; simp split: if-split-asm)
      (metis [[metis-verbose = false]] initials-memI D-T
```

```
      list.sel(1) list.sel(3) initial-hyps(2))+
  from ∗(5) have ‹s ∈ 𝒟 (P after a ⟦S⟧ Q)›
  proof (elim disjE)
    assume ‹t ∈ 𝒟 P ∧ u ∈ 𝒯 Q›
    hence ‹ftF v ∧ (tF (tl r) ∨ v = []) ∧ s = tl r @ v ∧
          tl r setinterleaves ((tl t, u), range tick ∪ ev ‘ S) ∧
          tl t ∈ 𝒟 (P after a) ∧ u ∈ 𝒯 Q›
      by (simp add: D-After initial-hyps(1))
        (metis ∗(1, 2, 3) ∗∗ ∗∗∗ list.collapse list.sel(3)
          tickFree-tl tl-append2)
    thus ‹s ∈ 𝒟 (P after a ⟦S⟧ Q)› by (simp add: D-Sync) blast
  next
    assume ‹t ∈ 𝒟 Q ∧ u ∈ 𝒯 P›
    hence ‹ftF v ∧ (tF (tl r) ∨ v = []) ∧ s = tl r @ v ∧
          tl r setinterleaves ((t, tl u), range tick ∪ ev ‘ S) ∧
          t ∈ 𝒟 Q ∧ tl u ∈ 𝒯 (P after a)›
      by (simp add: T-After initial-hyps(1))
        (metis ∗(1, 2, 3) ∗∗ ∗∗∗ list.collapse
          list.sel(3) tickFree-tl tl-append2)
    thus ‹s ∈ 𝒟 (P after a ⟦S⟧ Q)›
      by (simp add: D-Sync) blast
  qed
} note ∗ = this

show ‹s ∈ 𝒟 ((P ⟦S⟧ Q) after a) ⟹ s ∈ 𝒟 (P after a ⟦S⟧ Q)› for s
  by (simp add: D-After initials-Sync initial-hyps notin image-iff ∗
      split: if-split-asm)
    (elim disjE;
    simp add: After-BOT D-BOT, metis front-tickFree-Cons-iff front-tickFree-Nil)
next

fix s
{ assume assms : ‹P ≠ ⊥› ‹Q ≠ ⊥› ‹s ∈ 𝒟 (P after a ⟦S⟧ Q)›
  from assms(3) obtain t u r v
    where ∗ : ‹ftF v› ‹tF r ∨ v = []› ‹s = r @ v›
      ‹r setinterleaves ((t, u), range tick ∪ ev ‘ S)›
      ‹t ∈ 𝒟 (P after a) ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 (P after a)›
    by (simp add: D-Sync) blast
  from ∗(5) have ‹ev a # s ∈ 𝒟 (P ⟦S⟧ Q)›
  proof (elim disjE)
    assume ‹t ∈ 𝒟 (P after a) ∧ u ∈ 𝒯 Q›
    with ∗(1, 2, 3, 4) initial-hyps(1)
    have ∗∗ : ‹ftF v ∧ (tF (ev a # r) ∨ v = []) ∧ s = r @ v ∧
          (ev a # r) setinterleaves ((ev a # t, u), range tick ∪ ev ‘ S) ∧
          ev a # t ∈ 𝒟 P ∧ u ∈ 𝒯 Q›
      by (cases u; simp add: D-After notin image-iff)
    show ‹ev a # s ∈ 𝒟 (P ⟦S⟧ Q)›
      by (simp add: D-Sync) (metis ∗∗ Cons-eq-appendI)
  next
```

assume ‹$t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ (P\ after\ a)$›
with $*(1, 2, 3, 4)$ *initial-hyps*(1)
have $**$: ‹*ftF* $v \wedge$ (*tF* (*ev a* # *r*) $\vee$ $v = []$) $\wedge$ $s = r\ @\ v \wedge$
$(ev\ a \#\ r)$ *setinterleaves* $((t, ev\ a \#\ u)$, *range tick* $\cup$ *ev* ‘ *S*$) \wedge$
$t \in \mathcal{D}\ Q \wedge ev\ a \#\ u \in \mathcal{T}\ P$›
by (*cases t*; *simp add*: *T-After notin image-iff*)
show ‹*ev a* # $s \in \mathcal{D}\ (P\ [\![S]\!]\ Q)$›
by (*simp add*: *D-Sync*) (*metis* $**$ *Cons-eq-appendI*)
qed
}
thus ‹$s \in \mathcal{D}\ (P\ after\ a\ [\![S]\!]\ Q) \Longrightarrow s \in \mathcal{D}\ ((P\ [\![S]\!]\ Q)\ after\ a)$›
by (*simp add*: *D-After initials-Sync initial-hyps After-BOT D-BOT*
notin image-iff D-imp-front-tickFree front-tickFree-Cons-iff)
qed

lemma *not-initialL-in-After-Sync*:
‹*ev a* $\notin P^0 \Longrightarrow a \in S \Longrightarrow$
$(P\ [\![S]\!]\ Q)\ after\ a = (if\ Q = \bot\ then\ \bot\ else\ \Psi\ (P\ [\![S]\!]\ Q)\ a)$›
by (*simp add*: *After-BOT*, *intro impI*)
(*subst not-initial-After*, *auto simp add*: *After-BOT initials-Sync*)

After version of $[\![a \in ?S;\ a \in ?S]\!] \Longrightarrow a \rightarrow P\ [\![?S]\!]\ a \rightarrow Q = a \rightarrow (P\ [\![?S]\!]\ Q)$.

lemma *initialL-initialR-in-After-Sync*:
‹$(P\ [\![S]\!]\ Q)\ after\ a = P\ after\ a\ [\![S]\!]\ Q\ after\ a$›
if *initial-hyps*: ‹*ev a* $\in P^0$› ‹*ev a* $\in Q^0$› **and** *inside*: ‹$a \in S$›
proof (*subst Process-eq-spec-optimized*, *safe*)

{ fix *s X*
assume *assms* : ‹$P \neq \bot$› ‹$Q \neq \bot$› ‹$(ev\ a \#\ s, X) \in \mathcal{F}\ (P\ [\![S]\!]\ Q)$›
and *same-div* : ‹$\mathcal{D}\ ((P\ [\![S]\!]\ Q)\ after\ a) = \mathcal{D}\ (P\ after\ a\ [\![S]\!]\ Q\ after\ a)$›
from *assms*(3) **consider**
‹$\exists$ *s-P s-Q X-P X-Q*. $(s\text{-}P, X\text{-}P) \in \mathcal{F}\ P \wedge (s\text{-}Q, X\text{-}Q) \in \mathcal{F}\ Q \wedge$
$(ev\ a \#\ s)$ *setinterleaves* $((s\text{-}P, s\text{-}Q)$, *range tick* $\cup$ *ev* ‘ *S*$) \wedge$
$X = (X\text{-}P \cup X\text{-}Q) \cap (range\ tick \cup ev\ `\ S) \cup X\text{-}P \cap X\text{-}Q$› |
‹*ev a* # $s \in \mathcal{D}\ (P\ [\![S]\!]\ Q)$›
by (*simp add*: *F-Sync D-Sync*) *blast*
hence ‹$(s, X) \in \mathcal{F}\ (P\ after\ a\ [\![S]\!]\ Q\ after\ a)$›
proof *cases*
case *1*
then obtain *s-P s-Q X-P X-Q*
where $*$: ‹$(s\text{-}P, X\text{-}P) \in \mathcal{F}\ P$› ‹$(s\text{-}Q, X\text{-}Q) \in \mathcal{F}\ Q$›
‹$(ev\ a \#\ s)$ *setinterleaves* $((s\text{-}P, s\text{-}Q)$, *range tick* $\cup$ *ev* ‘ *S*$)$›
‹$X = (X\text{-}P \cup X\text{-}Q) \cap (range\ tick \cup ev\ `\ S) \cup X\text{-}P \cap X\text{-}Q$› **by** *blast*
from $*(3)$ **have** ‹$s\text{-}P \neq [] \wedge hd\ s\text{-}P = ev\ a \wedge s\text{-}Q \neq [] \wedge hd\ s\text{-}Q = ev\ a \wedge$
s setinterleaves $((tl\ s\text{-}P, tl\ s\text{-}Q)$, *range tick* $\cup$ *ev* ‘ *S*$)$›
using *inside* **by** (*cases s-P*; *cases s-Q*, *auto simp add*: *split*: *if-split-asm*)

```
    hence ‹(tl s-P, X-P) ∈ ℱ (P after a) ∧ (tl s-Q, X-Q) ∈ ℱ (Q after a) ∧
          s setinterleaves ((tl s-P, tl s-Q), range tick ∪ ev ‘ S) ∧
          X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ S) ∪ X-P ∩ X-Q›
     using ∗(1, 2, 4) initial-hyps by (simp add: F-After) (metis list.exhaust-sel)
    thus ‹(s, X) ∈ ℱ (P after a ⟦S⟧ Q after a)›
     by (simp add: F-Sync) blast
  next
    assume ‹ev a # s ∈ 𝒟 (P ⟦S⟧ Q)›
    hence ‹s ∈ 𝒟 ((P ⟦S⟧ Q) after a)›
     by (force simp add: D-After initials-Sync initial-hyps assms(1, 2))
    from this[simplified same-div] D-F
    show ‹(s, X) ∈ ℱ (P after a ⟦S⟧ Q after a)› by blast
  qed
 } note ∗ = this

 show ‹𝒟 ((P ⟦S⟧ Q) after a) = 𝒟 (P after a ⟦S⟧ Q after a) ⟹
       (s, X) ∈ ℱ ((P ⟦S⟧ Q) after a) ⟹ (s, X) ∈ ℱ (P after a ⟦S⟧ Q after a)›
for s X
   by (simp add: ∗ F-After initials-Sync initial-hyps image-iff split: if-split-asm)
   (metis After-BOT BOT-Sync BOT-iff-Nil-D CollectI D-BOT F-imp-front-tickFree
       Sync-BOT front-tickFree-Cons-iff is-processT8)
next

 fix s X
 assume same-div : ‹𝒟 ((P ⟦S⟧ Q) after a) = 𝒟 (P after a ⟦S⟧ Q after a)›
 { assume assms : ‹P ≠ ⊥› ‹Q ≠ ⊥› ‹(s, X) ∈ ℱ (P after a ⟦S⟧ Q after a)›
   from assms(3) consider
    ‹∃ s-P s-Q X-P X-Q. (ev a # s-P, X-P) ∈ ℱ P ∧ (ev a # s-Q, X-Q) ∈ ℱ Q
∧
                    s setinterleaves ((s-P, s-Q), range tick ∪ ev ‘ S) ∧
                    X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ S) ∪ X-P ∩ X-Q› |
    ‹s ∈ 𝒟 (P after a ⟦S⟧ Q after a)›
    by (simp add: F-Sync D-Sync F-After initial-hyps) blast
   hence ‹(ev a # s, X) ∈ ℱ (P ⟦S⟧ Q)›
   proof cases
    case 1
    then obtain s-P s-Q X-P X-Q
     where ∗ : ‹(ev a # s-P, X-P) ∈ ℱ P› ‹(ev a # s-Q, X-Q) ∈ ℱ Q›
       ‹s setinterleaves ((s-P, s-Q), range tick ∪ ev ‘ S)›
       ‹X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ S) ∪ X-P ∩ X-Q› by blast
    have ∗∗ : ‹(ev a # s) setinterleaves ((ev a # s-P, ev a # s-Q), range tick ∪
ev ‘ S)›
     by (simp add: inside ∗(3))
    show ‹(ev a # s, X) ∈ ℱ (P ⟦S⟧ Q)›
     apply (simp add: F-Sync)
     using ∗(1, 2, 4) ∗∗ by blast
   next
    assume ‹s ∈ 𝒟 (P after a ⟦S⟧ Q after a)›
    with same-div[symmetric] have ‹ev a # s ∈ 𝒟 (P ⟦S⟧ Q)›
```

```
      by (simp add: D-After initial-hyps initials-Sync assms(1, 2) image-iff)
    with D-F show ‹(ev a # s, X) ∈ ℱ (P ⟦S⟧ Q)› by blast
  qed
 }
 thus ‹(s, X) ∈ ℱ (P after a ⟦S⟧ Q after a) ⟹ (s, X) ∈ ℱ ((P ⟦S⟧ Q) after a)›
  by (simp add: F-After initials-Sync initial-hyps F-BOT image-iff
      F-imp-front-tickFree front-tickFree-Cons-iff)
next

 { fix s
  assume assms : ‹P ≠ ⊥› ‹Q ≠ ⊥› ‹ev a # s ∈ 𝒟 (P ⟦S⟧ Q)›
  from assms(3) obtain t u r v
   where ∗ : ‹ftF v› ‹tF  r ∨ v = []› ‹ev a # s = r @ v›
     ‹r setinterleaves ((t, u), range tick ∪ ev ‘ S)›
     ‹t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 P› by (simp add: D-Sync) blast
  have ∗∗ : ‹r ≠ [] ∧ hd r = ev a ∧ t ≠ [] ∧ hd t = ev a ∧ u ≠ [] ∧ hd u = ev a
∧
          tl r setinterleaves ((tl t, tl u), range tick ∪ ev ‘ S)›
   using ∗(3, 4, 5) inside assms(1, 2)[simplified BOT-iff-Nil-D]
   by (cases t; cases u) (auto simp add: image-iff split: if-split-asm)
  have ‹(tF  (tl r) ∨ v = []) ∧ s = tl r @ v ∧
      (tl t ∈ 𝒟 (P after a) ∧ tl u ∈ 𝒯 (Q after a) ∨
       tl t ∈ 𝒟 (Q after a) ∧ tl u ∈ 𝒯 (P after a))›
   using ∗(2, 3, 5) ∗∗ apply (simp add: D-After initial-hyps T-After)
   by (metis list.collapse list.sel(3) tickFree-tl tl-append2)
  with ∗(1) ∗∗ have ‹s ∈ 𝒟 (P after a ⟦S⟧ Q after a)› by (simp add: D-Sync)
blast
 } note ∗ = this

 show ‹s ∈ 𝒟 ((P ⟦S⟧ Q) after a) ⟹ s ∈ 𝒟 (P after a ⟦S⟧ Q after a)› for s
  by (simp add: ∗ D-After initials-Sync initial-hyps image-iff split: if-split-asm)
  (metis After-BOT BOT-Sync BOT-iff-Nil-D D-BOT Sync-BOT front-tickFree-Cons-iff
mem-Collect-eq)
next

 fix s
 { assume ‹s ∈ 𝒟 (P after a ⟦S⟧ Q after a)›
  from this obtain t u r v
   where ∗ : ‹ftF v› ‹tF  r ∨ v = []› ‹s = r @ v›
     ‹r setinterleaves ((t, u), range tick ∪ ev ‘ S)›
     ‹ev a # t ∈ 𝒟 P ∧ ev a # u ∈ 𝒯 Q ∨ ev a # t ∈ 𝒟 Q ∧ ev a # u ∈ 𝒯 P›
   by (simp add: D-Sync After-projs initial-hyps) blast
  have ∗∗ : ‹(ev a # r) setinterleaves ((ev a # t, ev a # u), range tick ∪ ev ‘
S)›
   by (simp add: inside ∗(4))
  have ‹ev a # s ∈ 𝒟 (P ⟦S⟧ Q)›
   by (simp add: D-Sync inside)
    (metis ∗(1, 2, 3, 5) ∗∗ Cons-eq-appendI event_ptick.disc(1) tickFree-Cons-iff)
 }
```

thus ‹$s \in \mathcal{D}\ (P\ after\ a\ [\![S]\!]\ Q\ after\ a) \Longrightarrow s \in \mathcal{D}\ ((P\ [\![S]\!]\ Q)\ after\ a)$›
by (*simp add*: *D-After initials-Sync initial-hyps After-BOT D-BOT inside*)
qed

After version of
$[\![e \notin ?S;\ e \notin ?S]\!] \Longrightarrow e \rightarrow P\ [\![?S]\!]\ e \rightarrow Q = (e \rightarrow (P\ [\![?S]\!]\ e \rightarrow Q)) \Box (e \rightarrow (e \rightarrow P\ [\![?S]\!]\ Q))$.

lemma *initialL-initialR-not-in-After-Sync*:
‹$(P\ [\![S]\!]\ Q)\ after\ a = (P\ after\ a\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ a)$›
if *initial-hyps*: ‹$ev\ a \in P^0$› ‹$ev\ a \in Q^0$› **and** *notin*: ‹$a \notin S$› **for** $P\ Q :: ('a, 'r)\ process_{ptick}$›
proof (*subst Process-eq-spec-optimized*, *safe*)

{ fix $P\ Q :: ('a, 'r)\ process_{ptick}$› **and** *s X s-P s-Q X-P X-Q*
assume *assms* : ‹$(s\text{-}P,\ X\text{-}P) \in \mathcal{F}\ P$› ‹$(s\text{-}Q,\ X\text{-}Q) \in \mathcal{F}\ Q$›
‹$X = (X\text{-}P \cup X\text{-}Q) \cap (range\ tick \cup ev\ `\ S) \cup X\text{-}P \cap X\text{-}Q$›
‹$s\text{-}P \neq []$› ‹$hd\ s\text{-}P = ev\ a$›
‹$s\ setinterleaves\ ((tl\ s\text{-}P,\ s\text{-}Q),\ range\ tick \cup ev\ `\ S)$›
‹$ev\ a \in P^0$›
from *assms*(*1*, *4*, *5*, *7*) **have** ‹$(tl\ s\text{-}P,\ X\text{-}P) \in \mathcal{F}\ (P\ after\ a)$›
by (*simp add*: *F-After*) (*metis list.exhaust-sel*)
with *assms*(*2*, *3*, *6*) **have** ‹$(s,\ X) \in \mathcal{F}\ (P\ after\ a\ [\![S]\!]\ Q)$›
by (*simp add*: *F-Sync*) *blast*
} note $* = this$

{ fix *s X*
assume *assms* : ‹$P \neq \bot$› ‹$Q \neq \bot$› ‹$(ev\ a\ \#\ s,\ X) \in \mathcal{F}\ (P\ [\![S]\!]\ Q)$›
and *same-div* : ‹$\mathcal{D}\ ((P\ [\![S]\!]\ Q)\ after\ a) = \mathcal{D}\ ((P\ after\ a\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ a))$›
from *assms*(*3*) **consider**
‹$\exists\, s\text{-}P\ s\text{-}Q\ X\text{-}P\ X\text{-}Q.\ (s\text{-}P,\ X\text{-}P) \in \mathcal{F}\ P \wedge (s\text{-}Q,\ X\text{-}Q) \in \mathcal{F}\ Q \wedge$
$(ev\ a\ \#\ s)\ setinterleaves\ ((s\text{-}P,\ s\text{-}Q),\ range\ tick \cup ev\ `\ S) \wedge$
$X = (X\text{-}P \cup X\text{-}Q) \cap (range\ tick \cup ev\ `\ S) \cup X\text{-}P \cap X\text{-}Q$› |
‹$s \in \mathcal{D}\ ((P\ [\![S]\!]\ Q)\ after\ a)$›
by (*simp add*: *F-Sync D-After D-Sync initials-Sync assms*(*1*, *2*) *initial-hyps image-iff*) *blast*
hence ‹$(s,\ X) \in \mathcal{F}\ ((P\ after\ a\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ a))$›
proof *cases*
case *1*
then obtain *s-P s-Q X-P X-Q*
where $**$: ‹$(s\text{-}P,\ X\text{-}P) \in \mathcal{F}\ P$› ‹$(s\text{-}Q,\ X\text{-}Q) \in \mathcal{F}\ Q$›
‹$(ev\ a\ \#\ s)\ setinterleaves\ ((s\text{-}P,\ s\text{-}Q),\ range\ tick \cup ev\ `\ S)$›
‹$X = (X\text{-}P \cup X\text{-}Q) \cap (range\ tick \cup ev\ `\ S) \cup X\text{-}P \cap X\text{-}Q$› **by** *blast*
have ‹$s\text{-}P \neq [] \wedge hd\ s\text{-}P = ev\ a \wedge s\ setinterleaves\ ((tl\ s\text{-}P,\ s\text{-}Q),\ range\ tick \cup ev\ `\ S) \vee$
$s\text{-}Q \neq [] \wedge hd\ s\text{-}Q = ev\ a \wedge s\ setinterleaves\ ((s\text{-}P,\ tl\ s\text{-}Q),\ range\ tick \cup ev\ `\ S)$›
using $**$(*3*) **by** (*cases s-P*; *cases s-Q*; *simp add*: *notin image-iff split*: *if-split-asm*) *blast*

```
    thus ‹(s, X) ∈ ℱ ((P after a ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q after a))›
      apply (elim disjE; simp add: F-Ndet)
      using ∗ ∗∗(1, 2, 4) initial-hyps(1) apply blast
      apply (rule disjI2, subst Sync-commute, rule ∗[OF ∗∗(2, 1)])
      by (simp-all add: ∗∗(4) Int-commute Un-commute Sync-commute
          setinterleaving-sym initial-hyps(2))
  next
    assume ‹s ∈ 𝒟 ((P ⟦S⟧ Q) after a)›
    from this[simplified same-div] D-F
    show ‹(s, X) ∈ ℱ ((P after a ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q after a))› by blast
  qed
 } note ∗∗ = this

 show ‹𝒟 ((P ⟦S⟧ Q) after a) = 𝒟 ((P after a ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q after a)) ⟹
     (s, X) ∈ ℱ ((P ⟦S⟧ Q) after a) ⟹ (s, X) ∈ ℱ ((P after a ⟦S⟧ Q) ⊓ (P ⟦S⟧
Q after a))› for s X
  by (simp add: ∗∗ F-After initials-Sync initial-hyps image-iff split: if-split-asm)
     (metis After-BOT BOT-Sync BOT-iff-Nil-D D-BOT F-imp-front-tickFree
Sync-commute
     front-tickFree-Cons-iff is-processT8 mem-Collect-eq)
next

 { fix s X P Q
  assume assms : ‹P ≠ ⊥› ‹Q ≠ ⊥› ‹(s, X) ∈ ℱ (P after a ⟦S⟧ Q)› ‹ev a ∈ P⁰›
    and same-div : ‹𝒟 ((P ⟦S⟧ Q) after a) = 𝒟 ((P after a ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q
after a))›
  from assms(3)[simplified F-Sync, simplified] consider
    s-P s-Q X-P X-Q where ‹(ev a # s-P, X-P) ∈ ℱ P› ‹(s-Q, X-Q) ∈ ℱ Q›
    ‹s setinterleaves ((s-P, s-Q), range tick ∪ ev ‘ S)›
    ‹X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ S) ∪ X-P ∩ X-Q›
  | ‹s ∈ 𝒟 (P after a ⟦S⟧ Q)›
    by (simp add: F-Sync F-After assms(4) D-Sync) blast
  hence ‹(ev a # s, X) ∈ ℱ (P ⟦S⟧ Q)›
  proof cases
    fix s-P s-Q X-P X-Q
    assume ∗ : ‹(ev a # s-P, X-P) ∈ ℱ P› ‹(s-Q, X-Q) ∈ ℱ Q›
      ‹s setinterleaves ((s-P, s-Q), range tick ∪ ev ‘ S)›
      ‹X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ S) ∪ X-P ∩ X-Q›
    have ‹(ev a # s) setinterleaves ((ev a # s-P, s-Q), range tick ∪ ev ‘ S)›
      using ∗(3) by (cases s-Q; simp add: notin image-iff)
    with ∗(1, 2, 4) show ‹(ev a # s, X) ∈ ℱ (P ⟦S⟧ Q)›
      by (simp add: F-Sync) blast
  next
    assume ‹s ∈ 𝒟 (P after a ⟦S⟧ Q)›
   hence ‹s ∈ 𝒟 ((P ⟦S⟧ Q) after a)› using same-div[simplified D-Ndet] by fast
    hence ‹(s, X) ∈ ℱ ((P ⟦S⟧ Q) after a)› by (simp add: is-processT8)
    thus ‹(ev a # s, X) ∈ ℱ (P ⟦S⟧ Q)›
      by (simp add: F-After initials-Sync assms(1, 2, 4) notin image-iff)
  qed
```

} **note** $* = this$

show ‹$\mathcal{D}$ ((P ⟦S⟧ Q) *after* a) = $\mathcal{D}$ ((P *after* a ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q *after* a)) ⟹ (s, X) ∈ $\mathcal{F}$ ((P *after* a ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q *after* a)) ⟹ (s, X) ∈ $\mathcal{F}$ ((P ⟦S⟧ Q) *after* a)› **for** s X
apply (*simp add*: *F-After initials-Sync initial-hyps F-BOT image-iff F-imp-front-tickFree front-tickFree-Cons-iff*)
apply (*rule impI*, *simp add*: *F-Ndet*, *elim disjE conjE*)
by (*simp add*: $*$ *initial-hyps*(*1*))
(*metis* $*$ *Ndet-commute Sync-commute initial-hyps*(*2*))
next

{ **fix** s
assume *assms* : ‹$P \neq \bot$› ‹$Q \neq \bot$› ‹*ev a* # $s \in \mathcal{D}$ (P ⟦S⟧ Q)›
from *assms*(*3*) **obtain** t u r v
where $**$: ‹*ftF* v› ‹*tF* $r \lor v = []$› ‹*ev a* # $s = r$ @ v›
‹r *setinterleaves* ((t, u), *range tick* ∪ *ev* ‘ S)›
‹$t \in \mathcal{D}$ $P \land u \in \mathcal{T}$ $Q \lor t \in \mathcal{D}$ $Q \land u \in \mathcal{T}$ P› **by** (*simp add*: *D-Sync*) *blast*
have $***$: ‹$r \neq []$› **using** $**$(*4*, *5*) *BOT-iff-Nil-D assms*(*1*, *2*) *empty-setinterleaving* **by** *blast*
have ‹$t \neq [] \land$ *hd* t = *ev a* $\land$ *tl* r *setinterleaves* ((*tl* t, u), *range tick* ∪ *ev* ‘ S) ∨ $u \neq [] \land$ *hd* u = *ev a* $\land$ *tl* r *setinterleaves* ((t, *tl* u), *range tick* ∪ *ev* ‘ S)›
using $**$(*3*, *4*) **by** (*cases t*; *cases u*, *auto simp add*: $***$ *notin split*: *if-split-asm*)
with $**$(*5*) **have** ‹$s \in \mathcal{D}$ ((P *after a* ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q *after a*))›
proof (*elim disjE*)
assume $*$: ‹$t \in \mathcal{D}$ $P \land u \in \mathcal{T}$ Q›
‹$t \neq [] \land$ *hd* t = *ev a* $\land$ *tl* r *setinterleaves* ((*tl* t, u), *range tick* ∪ *ev* ‘ S)›
hence ‹s = *tl* r @ $v \land$ *tl* $t \in \mathcal{D}$ (P *after a*) $\land$ $u \in \mathcal{T}$ Q›
using $**$(*3*) $***$ *initial-hyps*(*1*) **apply** (*simp add*: *D-After*, *intro conjI*)
by (*metis list.sel*(*3*) *tl-append2*) (*metis list.collapse*)
thus ‹$s \in \mathcal{D}$ ((P *after a* ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q *after a*))›
using $*$(*2*) $**$(*1*, *2*) *tickFree-tl* **by** (*simp add*: *D-Ndet D-Sync*) *blast*
next
assume $*$: ‹$t \in \mathcal{D}$ $P \land u \in \mathcal{T}$ Q›
‹$u \neq [] \land$ *hd* u = *ev a* $\land$ *tl* r *setinterleaves* ((t, *tl* u), *range tick* ∪ *ev* ‘ S)›
hence ‹s = *tl* r @ $v \land t \in \mathcal{D}$ $P \land$ *tl* $u \in \mathcal{T}$ (Q *after a*)›
using $**$(*3*) *initial-hyps*(*2*) $***$ **apply** (*simp add*: *T-After*, *intro conjI*)
by (*metis list.sel*(*3*) *tl-append2*) (*metis list.collapse*)
thus ‹$s \in \mathcal{D}$ ((P *after a* ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q *after a*))›
using $*$(*2*) $**$(*1*, *2*) *tickFree-tl* **by** (*simp add*: *D-Ndet D-Sync*) *blast*
next
assume $*$: ‹$t \in \mathcal{D}$ $Q \land u \in \mathcal{T}$ P›
‹$t \neq [] \land$ *hd* t = *ev a* $\land$ *tl* r *setinterleaves* ((*tl* t, u), *range tick* ∪ *ev* ‘ S)›
hence ‹s = *tl* r @ $v \land$ *tl* $t \in \mathcal{D}$ (Q *after a*) $\land$ $u \in \mathcal{T}$ P›
using $**$(*1*, *2*, *3*) *initial-hyps* **apply** (*simp add*: *D-After*, *intro conjI*)
by (*metis* $***$ *list.sel*(*3*) *tl-append2*) (*metis list.collapse*)
thus ‹$s \in \mathcal{D}$ ((P *after a* ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q *after a*))›
using $*$(*2*) $**$(*1*, *2*) *tickFree-tl* **by** (*simp add*: *D-Ndet D-Sync*) *blast*
next

assume $*$: ‹$t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P$›
‹$u \neq [] \wedge hd\ u = ev\ a \wedge tl\ r$ *setinterleaves* $((t,\ tl\ u),$ *range tick* $\cup$ *ev* ‘ $S)$›
hence ‹$s = tl\ r\ @\ v \wedge t \in \mathcal{D}\ Q \wedge tl\ u \in \mathcal{T}\ (P$ *after* $a)$›
using $**(1,\ 2,\ 3)$ *initial-hyps* **apply** (*simp add*: *T-After*, *intro conjI*)
by (*metis* $***$ *list.sel*(3) *tl-append2*) (*metis list.collapse*)
thus ‹$s \in \mathcal{D}\ ((P$ *after* $a\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q$ *after* $a))$›
using $*(2)$ $**(1,\ 2)$ *tickFree-tl* **by** (*simp add*: *D-Ndet D-Sync*) *blast*
qed
} **note** $* =$ *this*

show ‹$s \in \mathcal{D}\ ((P\ [\![S]\!]\ Q)$ *after* $a) \Longrightarrow s \in \mathcal{D}\ ((P$ *after* $a\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q$ *after* $a))$› **for** s
by (*simp add*: $*$ *D-After initials-Sync initial-hyps image-iff split*: *if-split-asm*)
(*metis D-BOT Ndet-is-BOT-iff Sync-is-BOT-iff*
front-tickFree-Cons-iff front-tickFree-Nil mem-Collect-eq)
next

{ **fix** $s\ P\ Q$
assume *assms* : ‹$P \neq \bot$› ‹$Q \neq \bot$› ‹$s \in \mathcal{D}\ ((P$ *after* $a\ [\![S]\!]\ Q))$› ‹$ev\ a \in P^0$›
from *assms*(3)[*simplified D-Sync*, *simplified*] **obtain** $t\ u\ r\ v$
where $*$: ‹*ftF* v› ‹$tF\ \ r \vee v = []$› ‹$s = r\ @\ v$›
‹r *setinterleaves* $((t,\ u),$ *range tick* $\cup$ *ev* ‘ $S)$›
‹$ev\ a \#\ t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge ev\ a \#\ u \in \mathcal{T}\ P$›
by (*simp add*: *D-Sync After-projs assms*(4)) *blast*
from $*(5)$ **have** ‹$ev\ a \#\ s \in \mathcal{D}\ (P\ [\![S]\!]\ Q)$›
proof (*elim disjE*)
assume $**$: ‹$ev\ a \#\ t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q$›
have $***$: ‹$(ev\ a \#\ r)$ *setinterleaves* $((ev\ a \#\ t,\ u),$ *range tick* $\cup$ *ev* ‘ $S)$›
using $*(4)$ **by** (*cases u*; *simp add*: *notin image-iff* $*(1,\ 2,\ 3)$)
show ‹$ev\ a \#\ s \in \mathcal{D}\ (P\ [\![S]\!]\ Q)$›
by (*simp add*: *D-Sync*)
(*metis* $*(1,\ 2,\ 3)$ $**$ $***$ *append-Cons* $event_{ptick}$.*disc*(1) *tickFree-Cons-iff*)
next
assume $**$: ‹$t \in \mathcal{D}\ Q \wedge ev\ a \#\ u \in \mathcal{T}\ P$›
have $***$: ‹$(ev\ a \#\ r)$ *setinterleaves* $((t,\ ev\ a \#\ u),$ *range tick* $\cup$ *ev* ‘ $S)$›
using $*(4)$ **by** (*cases t*; *simp add*: *notin image-iff* $*(1,\ 2,\ 3)$)
show ‹$ev\ a \#\ s \in \mathcal{D}\ (P\ [\![S]\!]\ Q)$›
by (*simp add*: *D-Sync*)
(*metis* $*(1,\ 2,\ 3)$ $**$ $***$ *append-Cons* $event_{ptick}$.*disc*(1) *tickFree-Cons-iff*)
qed
} **note** $* =$ *this*

show ‹$s \in \mathcal{D}\ ((P$ *after* $a\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q$ *after* $a)) \Longrightarrow s \in \mathcal{D}\ ((P\ [\![S]\!]\ Q)$ *after* $a)$› **for** s
by (*simp add*: *D-After initials-Sync D-BOT image-iff notin*)
(*metis* $*$ *D-Ndet D-imp-front-tickFree Sync-commute*
UnE $event_{ptick}$.*disc*(1) *front-tickFree-Cons-iff initial-hyps*)

qed

lemma *not-initialL-not-initialR-After-Sync*: ‹(*P* ⟦*S*⟧ *Q*) *after a* = Ψ (*P* ⟦*S*⟧ *Q*) *a*›

if *initial-hyps*: ‹*ev a* ∉ P^0› ‹*ev a* ∉ Q^0›
apply (*subst not-initial-After*, *simp add*: *initials-Sync*)
using *initials-BOT initial-hyps* **by** *auto*

Finally, the monster theorem !

theorem *After-Sync*:
‹(*P* ⟦*S*⟧ *Q*) *after a* =
(*if P* = ⊥ ∨ *Q* = ⊥ *then* ⊥
else if ev a ∈ P^0 ∩ Q^0
then if a ∈ *S then P after a* ⟦*S*⟧ *Q after a*
else (*P after a* ⟦*S*⟧ *Q*) ⊓ (*P* ⟦*S*⟧ *Q after a*)
else if ev a ∈ P^0 ∧ *a* ∉ *S then P after a* ⟦*S*⟧ *Q*
else if ev a ∈ Q^0 ∧ *a* ∉ *S then P* ⟦*S*⟧ *Q after a*
else Ψ (*P* ⟦*S*⟧ *Q*) *a*)›
by (*simp add*: *After-BOT initialL-initialR-in-After-Sync initialL-initialR-not-in-After-Sync*)
(*metis initialL-not-initialR-not-in-After-Sync Sync-commute*
not-initialL-in-After-Sync not-initialL-not-initialR-After-Sync)

3.5.4 *After* Hiding Operator

P \ *A* is harder to deal with, we will only obtain refinements results.

lemma *Hiding-FD-Hiding-After-if-initial-inside*:
‹*a* ∈ *A* ⟹ *P* \ *A* ⊑$_{FD}$ *P after a* \ *A*›
and *After-Hiding-FD-Hiding-After-if-initial-notin*:
‹*a* ∉ *A* ⟹ (*P* \ *A*) *after a* ⊑$_{FD}$ *P after a* \ *A*›
if *initial*: ‹*ev a* ∈ P^0›
supply *initial′* = *initial-notin-imp-initial-Hiding*[*OF initial*]
proof −
{ fix *s*
assume ‹*s* ∈ $\mathcal{D}$ (*P after a* \ *A*)›
with *D-Hiding* **obtain** *t u*
where ∗ : ‹*ftF u*› ‹*tF t*› ‹*s* = *trace-hide t* (*ev* ‘ *A*) @ *u*›
‹*t* ∈ $\mathcal{D}$ (*P after a*) ∨ (∃ *f*. *isInfHiddenRun f* (*P after a*) *A* ∧ *t* ∈ *range f*)›
by *blast*
from ∗(*4*) **have** ‹*s* ∈ (*if a* ∈ *A then* $\mathcal{D}$ (*P* \ *A*) *else* $\mathcal{D}$ ((*P* \ *A*) *after a*))›
proof (*elim disjE*)
assume ‹*t* ∈ $\mathcal{D}$ (*P after a*)›
hence ∗∗ : ‹*ev a* # *t* ∈ $\mathcal{D}$ *P*› **by** (*simp add*: *D-After initial*)
show ‹*s* ∈ (*if a* ∈ *A then* $\mathcal{D}$ (*P* \ *A*) *else* $\mathcal{D}$ ((*P* \ *A*) *after a*))›
proof (*split if-split*, *intro conjI impI*)
assume ‹*a* ∈ *A*›
with ∗(*3*) **have** ∗∗∗ : ‹*s* = *trace-hide* (*ev a* # *t*) (*ev* ‘ *A*) @ *u*› **by** *simp*
show ‹*s* ∈ $\mathcal{D}$ (*P* \ *A*)›
by (*simp add*: *D-Hiding*)
(*metis* ∗(*1*, *2*) ∗∗ ∗∗∗ *event*$_{ptick}$.*disc*(*1*) *tickFree-Cons-iff*)

```
    next
      assume ‹a ∉ A›
      with ∗(3) have ∗∗∗ : ‹ev a # s = trace-hide (ev a # t) (ev ‘ A) @ u›
        by (simp add: image-iff)
      have ‹ev a # s ∈ 𝒟 (P \ A)›
        by (simp add: D-Hiding)
          (metis ∗(1, 2) ∗∗ ∗∗∗ event_ptick.disc(1) tickFree-Cons-iff)
      thus ‹s ∈ 𝒟 ((P \ A) after a)› by (simp add: D-After initial' ‹a ∉ A›)
    qed
  next
    assume ‹∃f. isInfHiddenRun f (P after a) A ∧ t ∈ range f›
     then obtain f where ‹isInfHiddenRun f (P after a) A› ‹t ∈ range f› by
blast
    hence ∗∗ : ‹isInfHiddenRun (λi. ev a # f i) P A ∧
                ev a # t ∈ range (λi. ev a # f i)›
      by (simp add: monotone-on-def T-After initial) blast
    show ‹s ∈ (if a ∈ A then 𝒟 (P \ A) else 𝒟 ((P \ A) after a))›
    proof (split if-split, intro conjI impI)
      assume ‹a ∈ A›
      with ∗(3) have ∗∗∗ : ‹s = trace-hide (ev a # t) (ev ‘ A) @ u› by simp
      show ‹s ∈ 𝒟 (P \ A)›
        by (simp add: D-Hiding)
          (metis ∗(1, 2) ∗∗ ∗∗∗ event_ptick.disc(1) tickFree-Cons-iff)
    next
      assume ‹a ∉ A›
      with ∗(3) have ∗∗∗ : ‹ev a # s = trace-hide (ev a # t) (ev ‘ A) @ u›
        by (simp add: image-iff)
      have ‹ev a # s ∈ 𝒟 (P \ A)›
        by (simp add: D-Hiding)
          (metis ∗(1, 2) ∗∗ ∗∗∗ event_ptick.disc(1) tickFree-Cons-iff)
      thus ‹s ∈ 𝒟 ((P \ A) after a)› by (simp add: D-After initial' ‹a ∉ A›)
    qed
  qed
 } note div-ref = this

 { fix s X
  assume ‹(s, X) ∈ ℱ (P after a \ A)›
  with F-Hiding D-Hiding consider
    ‹∃t. s = trace-hide t (ev ‘ A) ∧ (t, X ∪ ev ‘ A) ∈ ℱ (P after a)›
    | ‹s ∈ 𝒟 (P after a \ A)› by blast
  hence ‹(s, X) ∈ (if a ∈ A then ℱ (P \ A) else ℱ ((P \ A) after a))›
  proof cases
    assume ‹∃t. s = trace-hide t (ev ‘ A) ∧ (t, X ∪ ev ‘ A) ∈ ℱ (P after a)›
    then obtain t where ∗ : ‹s = trace-hide t (ev ‘ A)› ‹(ev a # t, X ∪ ev ‘ A)
∈ ℱ P›
      by (simp add: F-After initial) blast
    show ‹(s, X) ∈ (if a ∈ A then ℱ (P \ A) else ℱ ((P \ A) after a))›
    proof (split if-split, intro conjI impI)
      from ∗ show ‹a ∈ A ⟹ (s, X) ∈ ℱ (P \ A)›
```

```
      by (simp add: F-Hiding) (metis filter.simps(2) image-eqI)
    next
      assume ‹a ∉ A›
      with ∗(1) have ∗∗ : ‹ev a # s = trace-hide (ev a # t) (ev ‘ A)›
        by (simp add: image-iff)
      show ‹(s, X) ∈ F ((P \ A) after a)›
        by (simp add: F-After initial' ‹a ∉ A› F-Hiding) (blast intro: ∗(2) ∗∗)
    qed
  next
    show ‹s ∈ D (P after a \ A) ⟹ (s, X) ∈ (if a ∈ A then F (P \ A) else F
((P \ A) after a))›
      by (drule div-ref, simp split: if-split-asm; use D-F in blast)
  qed
 } note fail-ref = this

 show ‹a ∈ A ⟹ P \ A ⊑FD P after a \ A›
  and ‹a ∉ A ⟹ (P \ A) after a ⊑FD P after a \ A›
  unfolding failure-divergence-refine-def failure-refine-def divergence-refine-def
  using div-ref fail-ref by auto
qed


lemmas Hiding-F-Hiding-After-if-initial-inside =
 Hiding-FD-Hiding-After-if-initial-inside[THEN leFD-imp-leF]
 and After-Hiding-F-Hiding-After-if-initial-notin =
 After-Hiding-FD-Hiding-After-if-initial-notin[THEN leFD-imp-leF]
 and Hiding-D-Hiding-After-if-initial-inside =
 Hiding-FD-Hiding-After-if-initial-inside[THEN leFD-imp-leD]
 and After-Hiding-D-Hiding-After-if-initial-notin =
 After-Hiding-FD-Hiding-After-if-initial-notin[THEN leFD-imp-leD]
 and Hiding-T-Hiding-After-if-initial-inside =
Hiding-FD-Hiding-After-if-initial-inside[THEN leFD-imp-leF, THEN leF-imp-leT]

 and After-Hiding-T-Hiding-After-if-initial-notin =
After-Hiding-FD-Hiding-After-if-initial-notin[THEN leFD-imp-leF, THEN leF-imp-leT]

corollary Hiding-DT-Hiding-After-if-initial-inside:
 ‹ev a ∈ P0 ⟹ a ∈ A ⟹ P \ A ⊑DT P after a \ A›
 and After-Hiding-DT-Hiding-After-if-initial-notin:
 ‹ev a ∈ P0 ⟹ a ∉ A ⟹ (P \ A) after a ⊑DT P after a \ A›
 by (simp add: Hiding-D-Hiding-After-if-initial-inside
     Hiding-T-Hiding-After-if-initial-inside leD-leT-imp-leDT)
   (simp add: After-Hiding-D-Hiding-After-if-initial-notin
     After-Hiding-T-Hiding-After-if-initial-notin leD-leT-imp-leDT)

end
```

3.5.5 Renaming is tricky

In all generality, *Renaming* takes a process $P :: ('a, 'r)\ process_{ptick}$, a function $f :: 'a \Rightarrow 'b$, a function $g :: 'r \Rightarrow 's$, and returns $Renaming\ P\ f\ g :: ('b, 's)\ process_{ptick}$. But if we try to write and prove a lemma *After-Renaming* like we did for the other operators, the mechanism of the locale *After* would constrain $f :: 'a \Rightarrow 'a$ and $g :: 'r \Rightarrow 'r$.

We solve this issue with a trick: we duplicate the locale, instantiating each one with a different free type.

locale *AfterDuplicated* = $After_\alpha : After\ \Psi_\alpha$ + $After_\beta : After\ \Psi_\beta$
  for $\Psi_\alpha :: \langle[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}\rangle$
    and $\Psi_\beta :: \langle[('b, 's)\ process_{ptick}, 'b] \Rightarrow ('b, 's)\ process_{ptick}\rangle$
begin

notation $After_\alpha.After$ (**infixl** $\langle after_\alpha \rangle$ *86*)
notation $After_\beta.After$ (**infixl** $\langle after_\beta \rangle$ *86*)

lemma *After-Renaming*:
  $\langle Renaming\ P\ f\ g\ after_\beta\ b =$
  — We highlight the fact that $f :: 'a \Rightarrow 'b$
  (*if* $P = \bot$ *then* $\bot$
    else if $\exists\, a.\ ev\ a \in P^0 \wedge f\ a = b$
      then $\sqcap\ a \in \{a.\ ev\ a \in P^0 \wedge f\ a = b\}.\ Renaming\ (P\ after_\alpha\ a)\ f\ g$
      else $\Psi_\beta\ (Renaming\ P\ f\ g)\ b)\rangle$
  (**is** $\langle ?lhs = (if\ P = \bot\ then\ \bot$
      else if $\exists\, a.\ ev\ a \in P^0 \wedge f\ a = b\ then\ ?rhs\ else\ -)\rangle$)

proof (*split if-split*, *intro conjI impI*)
  show $\langle P = \bot \Longrightarrow ?lhs = \bot\rangle$ **by** (*simp add*: $After_\beta.After\text{-}BOT$)
next
  assume *non-BOT*: $\langle P \neq \bot\rangle$
  show $\langle ?lhs = (if\ \exists\, a.\ ev\ a \in P^0 \wedge f\ a = b\ then\ ?rhs\ else\ \Psi_\beta\ (Renaming\ P\ f\ g)\ b)\rangle$
  proof (*split if-split*, *intro conjI impI*)
    show $\langle \nexists\, a.\ ev\ a \in P^0 \wedge f\ a = b \Longrightarrow ?lhs = \Psi_\beta\ (Renaming\ P\ f\ g)\ b\rangle$
      by (*subst* $After_\beta.not\text{-}initial\text{-}After$)
      (*auto simp add*: *initials-Renaming non-BOT image-iff* $ev\text{-}eq\text{-}map\text{-}event_{ptick}\text{-}iff$)
  next
    assume *assm* : $\langle\exists\, a.\ ev\ a \in P^0 \wedge f\ a = b\rangle$
    hence *initial*: $\langle ev\ b \in (Renaming\ P\ f\ g)^0\rangle$
      by (*auto simp add*: *initials-Renaming image-iff* $ev\text{-}eq\text{-}map\text{-}event_{ptick}\text{-}iff$)
    show $\langle ?lhs = ?rhs\rangle$
    proof (*subst Process-eq-spec-optimized*, *safe*)
      fix *s*
      assume $\langle s \in \mathcal{D}\ ?lhs\rangle$
      hence $*$: $\langle ev\ b\ \#\ s \in \mathcal{D}\ (Renaming\ P\ f\ g)\rangle$
        by (*auto simp add*: *initial* $After_\beta.D\text{-}After$ *split*: *if-split-asm*)
      from $*$ **obtain** *t1 t2*

where $**$: ‹*tF t1*› ‹*ftF t2*›
‹*ev b* # *s* = *map* (*map-event*$_{ptick}$ *f g*) *t1* @ *t2*› ‹*t1* ∈ $\mathcal{D}$ *P*›
by (*simp add*: *D-Renaming*) *blast*
from $**$(*1*, *3*, *4*) *non-BOT* **obtain** *a t1'*
where $***$: ‹*t1* = *ev a* # *t1'*› ‹*f a* = *b*›
by (*cases t1*) (*auto simp add*: *BOT-iff-Nil-D ev-eq-map-event*$_{ptick}$*-iff*)
have ‹*ev a* ∈ P^0›
using $**$(*4*) $***$(*1*) *initials-memI D-T* **by** *blast*
also have ‹*s* ∈ $\mathcal{D}$ (*Renaming* (*P after*$_\alpha$ *a*) *f g*)›
using $**$ $***$(*1*) **by** (*simp add*: *D-Renaming After*$_\alpha$*.D-After calculation*) *blast*
ultimately show ‹*s* ∈ $\mathcal{D}$ *?rhs*›
using $***$(*2*) **by** (*simp add*: *D-GlobalNdet*) *blast*
next
fix *s*
assume ‹*s* ∈ $\mathcal{D}$ *?rhs*›
then obtain *a* **where** $*$: ‹*ev a* ∈ P^0› ‹*f a* = *b*›
‹*s* ∈ $\mathcal{D}$ (*Renaming* (*P after*$_\alpha$ *a*) *f g*)›
by (*simp add*: *D-GlobalNdet split*: *if-split-asm*) *blast*
from $*$(*3*) **obtain** *s1 s2*
where $**$: ‹*tF s1*› ‹*ftF s2*›
‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *s2*› ‹*ev a* # *s1* ∈ $\mathcal{D}$ *P*›
by (*simp add*: *D-Renaming After*$_\alpha$*.D-After* $*$(*1*)) *blast*
have $***$: ‹*tF* (*ev a* # *s1*) ∧ *ev b* # *s* = *map* (*map-event*$_{ptick}$ *f g*) (*ev a* # *s1*) @ *s2*›
by (*simp add*: $**$(*1*, *3*) $*$(*2*))
from $**$(*2*, *4*) **show** ‹*s* ∈ $\mathcal{D}$ *?lhs*›
by (*simp add*: *After*$_\beta$*.D-After initial D-Renaming*) (*use* $***$ **in** *blast*)
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
then consider ‹*ev b* ∉ (*Renaming P f g*)^0› ‹*s* = []›
| ‹*ev b* ∈ (*Renaming P f g*)^0› ‹(*ev b* # *s*, *X*) ∈ $\mathcal{F}$ (*Renaming P f g*)›
by (*simp add*: *initial After*$_\beta$*.F-After split*: *if-split-asm*)
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
proof *cases*
from *initial* **show** ‹*ev b* ∉ (*Renaming P f g*)^0 ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*› **by** *simp*
next
assume *assms* : ‹*ev b* ∈ (*Renaming P f g*)^0›
‹(*ev b* # *s*, *X*) ∈ $\mathcal{F}$ (*Renaming P f g*)›
from *assms*(*2*) **consider** ‹*ev b* # *s* ∈ $\mathcal{D}$ (*Renaming P f g*)›
| *s1* **where** ‹(*s1*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ *P*› ‹*ev b* # *s* = *map* (*map-event*$_{ptick}$ *f g*) *s1*›
by (*simp add*: *F-Renaming D-Renaming*) *meson*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
proof *cases*
assume ‹*ev b* # *s* ∈ $\mathcal{D}$ (*Renaming P f g*)›
hence ‹*s* ∈ $\mathcal{D}$ *?lhs*› **by** (*force simp add*: *After*$_\beta$*.D-After assms*(*1*))

with *D-F same-div* **show** ‹$(s,\ X) \in \mathcal{F}$ *?rhs*› **by** *blast*
next
fix *s1* **assume** ∗ : ‹$(s1,\ map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ P$›
‹$ev\ b\ \#\ s = map\ (map\text{-}event_{ptick}\ f\ g)\ s1$›
from ∗(*2*) **obtain** *a s1′*
where ∗∗ : ‹$s1 = ev\ a\ \#\ s1'$› ‹$f\ a = b$›
by (*cases s1*; *simp*) (*metis* $map\text{-}event_{ptick}$*-eq-ev-iff*)
have ‹$ev\ a \in P^0$›
using ∗(*1*) ∗∗(*1*) *initials-memI F-T* **by** *blast*
also have ‹$(s,\ X) \in \mathcal{F}\ (Renaming\ (P\ after_\alpha\ a)\ f\ g)$›
using ∗(*1*, *2*) ∗∗(*1*) **by** (*simp add*: *F-Renaming* $After_\alpha$*.F-After calculation*) *blast*
ultimately show ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
using ∗∗(*2*) **by** (*simp add*: *F-GlobalNdet*) *blast*
qed
qed
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
then consider ‹$\forall\ a.\ ev\ a \in P^0 \longrightarrow f\ a \neq b$› ‹$s = []$›
| *a* **where** ‹$f\ a = b$› ‹$ev\ a \in P^0$› ‹$(s,\ X) \in \mathcal{F}\ (Renaming\ (P\ after_\alpha\ a)\ f\ g)$›
by (*auto simp add*: *F-GlobalNdet split*: *if-split-asm*)
thus ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
proof *cases*
from *assm* **show** ‹$\forall\ a.\ ev\ a \in P^0 \longrightarrow f\ a \neq b \Longrightarrow s = [] \Longrightarrow (s,\ X) \in \mathcal{F}$ *?lhs*›
by (*auto simp add*: $After_\beta$*.F-After F-Renaming initials-Renaming non-BOT*)
next
fix *a* **assume** ∗ : ‹$f\ a = b$› ‹$ev\ a \in P^0$› ‹$(s,\ X) \in \mathcal{F}\ (Renaming\ (P\ after_\alpha\ a)\ f\ g)$›
from ∗(*3*) **consider** ‹$s \in \mathcal{D}\ (Renaming\ (P\ after_\alpha\ a)\ f\ g)$›
| *s1* **where** ‹$(s1,\ map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ (P\ after_\alpha\ a)$› ‹$s = map\ (map\text{-}event_{ptick}\ f\ g)\ s1$›
by (*simp add*: *F-Renaming D-Renaming*) *blast*
thus ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
proof *cases*
assume ‹$s \in \mathcal{D}\ (Renaming\ (P\ after_\alpha\ a)\ f\ g)$›
with ∗(*1*, *2*) **have** ‹$s \in \mathcal{D}$ *?rhs*› **by** (*simp add*: *D-GlobalNdet*) *blast*
with *D-F same-div* **show** ‹$(s,\ X) \in \mathcal{F}$ *?lhs*› **by** *blast*
next
fix *s1* **assume** ∗∗ : ‹$(s1,\ map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ (P\ after_\alpha\ a)$›
‹$s = map\ (map\text{-}event_{ptick}\ f\ g)\ s1$›
from *initial* ∗(*1*, *2*) ∗∗
have ‹$(ev\ a\ \#\ s1,\ map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ P \wedge ev\ b\ \#\ s = map\ (map\text{-}event_{ptick}\ f\ g)\ (ev\ a\ \#\ s1)$›
by (*simp add*: $After_\alpha$*.F-After* ∗(*2*))
hence ‹$(ev\ b\ \#\ s,\ X) \in \mathcal{F}\ (Renaming\ P\ f\ g)$›
by (*auto simp add*: *F-Renaming*)

```
        thus ‹(s, X) ∈ ℱ ?lhs›
          by (simp add: After_β.F-After initial)
      qed
    qed
  qed
 qed
qed

no-notation After_α.After (infixl ‹after_α› 86)
no-notation After_β.After (infixl ‹after_β› 86)

end
```

Now we can get back to *After*.

```
context After
begin
```

3.6 Behaviour of *After* with Operators of `HOL-CSPM`

```
lemma After-GlobalDet-is-After-GlobalNdet:
 ‹ev a ∈ (⋃ a ∈ A. (P a)⁰) ⟹ (□ a ∈ A. P a) after a = (⊓ a ∈ A. P a) after a›
 by (simp add: Process-eq-spec After-projs initials-GlobalDet
     initials-GlobalNdet GlobalNdet-projs GlobalDet-projs)


lemma After-GlobalNdet:
 ‹(⊓ a ∈ A. P a) after a =  (  if ev a ∈ (⋃ a ∈ A. (P a)⁰)
                          then ⊓ x ∈ {x ∈ A. ev a ∈ (P x)⁰}. P x after a
                          else Ψ (⊓ a ∈ A. P a) a)›
 (is ‹?lhs = (if ?prem then ?rhs else -)›)
proof (split if-split, intro conjI impI)
 show ‹¬ ?prem ⟹ ?lhs = Ψ (⊓ a ∈ A. P a) a›
  by (simp add: not-initial-After initials-GlobalNdet)
next
 assume ?prem
 then obtain x where ‹x ∈ A› ‹ev a ∈ (P x)⁰› by blast
 show ‹?lhs = ?rhs›
 proof (subst Process-eq-spec, safe)
  from ‹x ∈ A› ‹ev a ∈ (P x)⁰› show ‹t ∈ 𝒟 ?lhs ⟹ t ∈ 𝒟 ?rhs› for t
   by (auto simp add: D-After initials-GlobalNdet D-GlobalNdet
       split: if-split-asm intro: D-T initials-memI)
 next
  show ‹t ∈ 𝒟 ?rhs ⟹ t ∈ 𝒟 ?lhs› for t
   by (auto simp add: D-GlobalNdet D-After initials-GlobalNdet)
 next
  from ‹x ∈ A› ‹ev a ∈ (P x)⁰› show ‹(t, X) ∈ ℱ ?lhs ⟹ (t, X) ∈ ℱ ?rhs›
for t X
   by (auto simp add: F-After initials-GlobalNdet F-GlobalNdet
       split: if-split-asm intro: F-T initials-memI)
```

next
from ‹$x \in A$› ‹$ev\ a \in (P\ x)^0$›› **show** ‹$(t,\ X) \in \mathcal{F}\ ?rhs \Longrightarrow (t,\ X) \in \mathcal{F}\ ?lhs$›
for $t\ X$
by (*auto simp add*: *F-GlobalNdet F-After initials-GlobalNdet split*: *if-split-asm*)
qed
qed

lemma *After-GlobalDet*:
‹$(\Box\ a \in A.\ P\ a)\ after\ a = (\ \ if\ ev\ a \in (\bigcup a \in A.\ (P\ a)^0)$
$then\ \sqcap\ x \in \{x \in A.\ ev\ a \in (P\ x)^0\}.\ P\ x\ after\ a$
$else\ \Psi\ (\Box\ a \in A.\ P\ a)\ a)$›
by (*simp add*: *After-GlobalDet-is-After-GlobalNdet After-GlobalNdet initials-GlobalDet not-initial-After*)

3.6.1 *After* Throwing

lemma *After-Throw*:
‹$(P\ \Theta\ a \in A.\ Q\ a)\ after\ a =$
$(\ \ if\ P = \bot\ then\ \bot$
$else\ \ \ if\ ev\ a \in P^0\ then\ \ \ if\ a \in A\ then\ Q\ a$
$else\ P\ after\ a\ \Theta\ a \in A.\ Q\ a$
$else\ \Psi\ (P\ \Theta\ a \in A.\ Q\ a)\ a)$›
(**is** ‹$?lhs = ?rhs$›)
proof −
have ‹$?lhs = Q\ a$› **if** ‹$P \neq \bot$› **and** ‹$ev\ a \in P^0$› **and** ‹$a \in A$›
proof (*rule Process-eq-optimizedI*)
fix t
assume ‹$t \in \mathcal{D}\ ?lhs$›
with ‹$ev\ a \in P^0$› **have** ‹$ev\ a\ \#\ t \in \mathcal{D}\ (P\ \Theta\ a \in A.\ Q\ a)$›
by (*simp add*: *D-After initials-Throw*)
with ‹$P \neq \bot$› **show** ‹$t \in \mathcal{D}\ (Q\ a)$›
apply (*simp add*: *D-Throw disjoint-iff BOT-iff-Nil-D*, *elim disjE*)
by (*metis hd-append2 hd-in-set image-eqI list.sel(1)* ‹$a \in A$›)
(*metis append-self-conv2 event$_{ptick}$.inject(1) hd-append2 hd-in-set image-eqI list.sel(1, 3)* ‹$a \in A$›)
next
have ‹$[ev\ a] \in \mathcal{T}\ P \wedge a \in A$› **using** *initials-def* ‹$ev\ a \in P^0$› ‹$a \in A$› **by** *blast*
thus ‹$t \in \mathcal{D}\ (Q\ a) \Longrightarrow t \in \mathcal{D}\ ((P\ \Theta\ a \in A.\ Q\ a)\ after\ a)$› **for** t
by (*simp add*: *D-After initials-Throw* ‹$ev\ a \in P^0$› *D-Throw*)
(*metis append-Nil empty-set inf-bot-left*)
next
fix $t\ X$
assume ‹$(t,\ X) \in \mathcal{F}\ ?lhs$›
with ‹$ev\ a \in P^0$› **have** ‹$(ev\ a\ \#\ t,\ X) \in \mathcal{F}\ (P\ \Theta\ a \in A.\ Q\ a)$›
by (*simp add*: *F-After initials-Throw*)
with ‹$P \neq \bot$› ‹$a \in A$› **show** ‹$(t,\ X) \in \mathcal{F}\ (Q\ a)$›
apply (*simp add*: *F-Throw image-iff BOT-iff-Nil-D*, *elim disjE*)
by (*metis disjoint-iff hd-append2 hd-in-set image-eqI list.sel(1)*)

```
    (metis append-Nil event_ptick.inject(1) hd-append2 imageI insert-disjoint(2)
      list.exhaust-sel list.sel(1, 3) list.simps(15))
next
  have ‹[ev a] ∈ 𝒯 P ∧ a ∈ A› using initials-def ‹ev a ∈ P⁰› ‹a ∈ A› by blast
  thus ‹(t, X) ∈ ℱ (Q a) ⟹ (t, X) ∈ ℱ ?lhs› for t X
    by (simp add: F-After initials-Throw ‹ev a ∈ P⁰› F-Throw)
      (metis append-Nil empty-set inf-bot-left)
qed

also have ‹?lhs = (P after a) Θ a ∈ A. Q a›
  if ‹P ≠ ⊥› and ‹ev a ∈ P⁰› and ‹a ∉ A›
proof (rule Process-eq-optimizedI)
  fix t
  assume ‹t ∈ 𝒟 ?lhs›
  with ‹ev a ∈ P⁰› have ‹ev a # t ∈ 𝒟 (P Θ a ∈ A. Q a)›
    by (simp add: D-After initials-Throw)
  then consider (divL) t1 t2 where ‹ev a # t = t1 @ t2› ‹t1 ∈ 𝒟 P› ‹tF t1›
    ‹set t1 ∩ ev ‘ A = {}› ‹ftF t2›
  | (divR) t1 t2 a' where ‹ev a # t = t1 @ ev a' # t2› ‹t1 @ [ev a'] ∈ 𝒯 P›
    ‹set t1 ∩ ev ‘ A = {}› ‹a' ∈ A› ‹t2 ∈ 𝒟 (Q a')›
    by (simp add: D-Throw) blast
  thus ‹t ∈ 𝒟 ((P after a) Θ a ∈ A. Q a)›
  proof cases
    case divL
    from ‹P ≠ ⊥› divL(1, 2) BOT-iff-Nil-D obtain t1'
      where ‹t1 = ev a # t1'› by (cases t1) auto
    with divL(1–4) have ‹t = t1' @ t2 ∧ t1' ∈ 𝒟 (P after a) ∧
                          tF t1' ∧ set t1' ∩ ev ‘ A = {}›
      by (simp add: D-After ‹ev a ∈ P⁰›)
    with divL(5) show ‹t ∈ 𝒟 ((P after a) Θ a ∈ A. Q a)›
      by (auto simp add: D-Throw)
  next
    case divR
    have ‹a ≠ a'› using divR(4) ‹a ∉ A› by blast
    with divR(1) obtain t1' where ** : ‹t1 = ev a # t1'› by (cases t1) auto
    with divR(2) ‹ev a ∈ P⁰› have ‹t1' @ [ev a'] ∈ 𝒯 (P after a)›
      by (simp add: T-After)
    with divR(1, 3–5) ** show ‹t ∈ 𝒟 ((P after a) Θ a ∈ A. Q a)›
      by (simp add: D-Throw) blast
  qed
next
  fix t
  assume ‹t ∈ 𝒟 ((P after a) Θ a ∈ A. Q a)›
  then consider (divL) t1 t2 where ‹t = t1 @ t2› ‹t1 ∈ 𝒟 (P after a)›
    ‹tF t1› ‹set t1 ∩ ev ‘ A = {}› ‹ftF t2›
  | (divR) t1 a' t2 where ‹t = t1 @ ev a' # t2› ‹t1 @ [ev a'] ∈ 𝒯 (P after a)›
    ‹set t1 ∩ ev ‘ A = {}› ‹a' ∈ A› ‹t2 ∈ 𝒟 (Q a')›
    unfolding D-Throw by blast
  thus ‹t ∈ 𝒟 ?lhs›
```

proof *cases*
case *divL*
from *divL(2)* ‹*ev a* ∈ P^0› **have** ** : ‹*ev a* # *t1* ∈ $\mathcal{D}$ *P*› **by** (*simp add*: *D-After*)
have *** : ‹*tF* (*ev a* # *t1*) ∧ *set* (*ev a* # *t1*) ∩ *ev* ‘ *A* = {}›
by (*simp add*: *image-iff divL*(*3*, *4*) ‹*a* ∉ *A*›)
show ‹*t* ∈ $\mathcal{D}$ *?lhs*›
by (*simp add*: *D-After D-Throw initials-Throw* ‹*ev a* ∈ P^0›)
(*metis divL*(*1*, *5*) ** *** *Cons-eq-appendI*)
next
case *divR*
from *divR(2)* ‹*ev a* ∈ P^0› **have** ** : ‹*ev a* # *t1* @ [*ev a'*] ∈ $\mathcal{T}$ *P*›
by (*simp add*: *T-After*)
have *** : ‹*set* (*ev a* # *t1*) ∩ *ev* ‘ *A* = {}›
by (*simp add*: *image-iff divR*(*3*) ‹*a* ∉ *A*›)
show ‹*t* ∈ $\mathcal{D}$ *?lhs*›
by (*simp add*: *D-After D-Throw initials-Throw* ‹*ev a* ∈ P^0›)
(*metis divR*(*1*, *4*, *5*) ** *** *Cons-eq-appendI*)
qed
next
fix *t X* **assume** ‹*t* ∉ $\mathcal{D}$ *?lhs*›
assume ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*›
with ‹*ev a* ∈ P^0› **have** ‹(*ev a* # *t*, *X*) ∈ $\mathcal{F}$ (*P* Θ *a* ∈ *A*. *Q a*)›
by (*simp add*: *F-After initials-Throw*)
with ‹*t* ∉ $\mathcal{D}$ *?lhs*› ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*› **consider** ‹(*ev a* # *t*, *X*) ∈ $\mathcal{F}$ *P*› ‹*set t* ∩ *ev* ‘ *A* = {}›
| (*failR*) *t1 a' t2* **where** ‹*ev a* # *t* = *t1* @ *ev a'* # *t2*› ‹*t1* @ [*ev a'*] ∈ $\mathcal{T}$ *P*›
‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*a'* ∈ *A*› ‹(*t2*, *X*) ∈ $\mathcal{F}$ (*Q a'*)›
by (*auto simp add*: *D-After Throw-projs initials-Throw split*: *if-split-asm*)
(*metis D-T initials-memI is-processT7*)
thus ‹(*t*, *X*) ∈ $\mathcal{F}$ (*P after a* Θ *a*∈*A*. *Q a*)›
proof *cases*
show ‹(*ev a* # *t*, *X*) ∈ $\mathcal{F}$ *P* ⟹ *set t* ∩ *ev* ‘ *A* = {} ⟹
(*t*, *X*) ∈ $\mathcal{F}$ (*P after a* Θ *a*∈*A*. *Q a*)›
by (*simp add*: *F-Throw F-After* ‹*ev a* ∈ P^0›)
next
case *failR*
have ‹*a* ≠ *a'*› **using** *failR*(*4*) ‹*a* ∉ *A*› **by** *blast*
with *failR*(*1*) **obtain** *t1'* **where** ‹*t1* = *ev a* # *t1'*› **by** (*cases t1*) *auto*
also have ‹*t1'* @ [*ev a'*] ∈ $\mathcal{T}$ (*P after a*) ∧ *set t1'* ∩ *ev* ‘ *A* = {}›
using *failR*(*2*, *3*) **by** (*simp add*: *image-iff T-After* ‹*ev a* ∈ P^0› *calculation*)
ultimately show ‹(*t*, *X*) ∈ $\mathcal{F}$ (*P after a* Θ *a* ∈ *A*. *Q a*)›
using *failR*(*1*, *4*, *5*) **by** (*simp add*: *F-Throw*) *blast*
qed
next
fix *t X*
assume ‹*t* ∉ $\mathcal{D}$ (*P after a* Θ *a*∈*A*. *Q a*)› **and** ‹(*t*, *X*) ∈ $\mathcal{F}$ (*P after a* Θ *a*∈*A*. *Q a*)›
then consider (*failL*) ‹(*t*, *X*) ∈ $\mathcal{F}$ (*P after a*)› ‹*set t* ∩ *ev* ‘ *A* = {}›

| (*failR*) *t1* a' *t2* **where** ‹$t = t1$ @ *ev* $a' \# t2$› ‹*t1* @ [*ev* a'] $\in \mathcal{T}$ (*P after a*)›
‹*set t1* ∩ *ev* ‘ $A = \{\}$› ‹$a' \in A$› ‹($t2$, X) $\in \mathcal{F}$ (*Q* a')›
by (*auto simp add*: *Throw-projs D-After* ‹*ev a* $\in P^0$›)
thus ‹(t, X) $\in \mathcal{F}$ *?lhs*›
proof *cases*
case *failL*
from *failL*(*2*) **have** $*$: ‹*set* (*ev a* # *t*) ∩ *ev* ‘ $A = \{\}$›
by (*simp add*: *image-iff* ‹$a \notin A$› *failL*(*2*))
from *failL*(*1*) **have** ‹(*ev a* # *t*, X) $\in \mathcal{F}$ *P*›
by (*simp add*: *F-After* ‹*ev a* $\in P^0$›)
with $*$ **show** ‹(t, X) $\in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-After F-Throw initials-Throw* ‹*ev a* $\in P^0$›)
next
case *failR*
from *failR*(*2*) ‹*ev a* $\in P^0$› **have** $**$: ‹*ev a* # *t1* @ [*ev* a'] $\in \mathcal{T}$ *P*›
by (*simp add*: *T-After*)
have $***$: ‹*set* (*ev a* # *t1*) ∩ *ev* ‘ $A = \{\}$›
by (*simp add*: *image-iff failR*(*3*) ‹$a \notin A$›)
from *failR*(*1*, *4*, *5*) **show** ‹(t, X) $\in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-After F-Throw initials-Throw* ‹*ev a* $\in P^0$›)
(*metis* $**$ $***$ *append-Cons*)
qed
qed

ultimately show ‹*?lhs* = *?rhs*›
by (*simp add*: *After-BOT not-initial-After initials-Throw*)
qed

3.6.2 *After* Interrupting

theorem *After-Interrupt*:
‹(*P* △ *Q*) *after a* =
(*if ev a* $\in P^0 \cap Q^0$
then Q after a ⊓ (*P after a* △ *Q*)
else if ev a $\in P^0 \wedge$ *ev a* $\notin Q^0$
then P after a △ *Q*
else if ev a $\notin P^0 \wedge$ *ev a* $\in Q^0$
then Q after a
else Ψ (*P* △ *Q*) *a*)›
proof −
have ‹(*P* △ *Q*) *after a* $\sqsubseteq_{FD}$ *Q after a*› **if** *initial*: ‹*ev a* $\in Q^0$›
proof (*unfold failure-divergence-refine-def failure-refine-def divergence-refine-def*, *safe*)
fix t
assume ‹$t \in \mathcal{D}$ (*Q after a*)›
hence ‹*ev a* # $t \in \mathcal{D}$ *Q*›
by (*simp add*: *D-After initial*)
thus ‹$t \in \mathcal{D}$ ((*P* △ *Q*) *after a*)›
by (*simp add*: *D-After initial initials-Interrupt D-Interrupt*)

(*metis Nil-elem-T append-Nil tickFree-Nil*)
next
show ‹$(t, X) \in \mathcal{F}\ (Q\ after\ a) \Longrightarrow (t, X) \in \mathcal{F}\ ((P \triangle Q)\ after\ a)$› **for** $t\ X$
by (*simp add*: *F-After initials-Interrupt F-Interrupt initial*)
(*metis Nil-elem-T append-Nil list.distinct*(*1*) *tickFree-Nil*)
qed

moreover have ‹$(P \triangle Q)\ after\ a \sqsubseteq_{FD} P\ after\ a \triangle Q$› **if** *initial*: ‹$ev\ a \in P^0$›
proof (*unfold failure-divergence-refine-def failure-refine-def divergence-refine-def*, *safe*)
show ‹$t \in \mathcal{D}\ (P\ after\ a \triangle Q) \Longrightarrow t \in \mathcal{D}\ ((P \triangle Q)\ after\ a)$› **for** t
by (*simp add*: *D-Interrupt D-After initial T-After initials-Interrupt*)
(*metis append-Cons event*$_{ptick}$*.disc*(*1*) *tickFree-Cons-iff*)
next
show ‹$(t, X) \in \mathcal{F}\ (P\ after\ a \triangle Q) \Longrightarrow (t, X) \in \mathcal{F}\ ((P \triangle Q)\ after\ a)$› **for** $t\ X$
by (*simp add*: *F-Interrupt F-After initial initials-Interrupt T-After D-After*, *elim disjE*)
(*metis* [[*metis-verbose* = *false*]] *Cons-eq-appendI event*$_{ptick}$*.disc*(*1*) *tickFree-Cons-iff*)+
qed

moreover have ‹$Q\ after\ a \sqsubseteq_{FD} (P \triangle Q)\ after\ a$›
if *initial-hyps*: ‹$ev\ a \notin P^0$› ‹$ev\ a \in Q^0$›
proof (*unfold failure-divergence-refine-def failure-refine-def divergence-refine-def*, *safe*)
show ‹$t \in \mathcal{D}\ ((P \triangle Q)\ after\ a) \Longrightarrow t \in \mathcal{D}\ (Q\ after\ a)$› **for** t
by (*simp add*: *D-After initials-Interrupt D-Interrupt initial-hyps*)
(*metis initials-memI D-T append-Nil hd-append*
list.exhaust-sel list.sel(*1*) *initial-hyps*(*1*))
next
from *initials-memI*[*of* ‹$ev\ a$› - P, *simplified initial-hyps*(*1*)]
show ‹$(t, X) \in \mathcal{F}\ ((P \triangle Q)\ after\ a) \Longrightarrow (t, X) \in \mathcal{F}\ (Q\ after\ a)$› **for** $t\ X$
by (*auto simp add*: *F-After initials-Interrupt F-Interrupt initial-hyps intro*: *F-T D-T*)
(*metis*, *metis append-eq-Cons-conv*, *metis append-eq-Cons-conv is-processT8*)
qed

moreover have ‹$P\ after\ a \triangle Q \sqsubseteq_{FD} (P \triangle Q)\ after\ a$›
if *initial-hyps*: ‹$ev\ a \in P^0$› ‹$ev\ a \notin Q^0$›
proof (*unfold failure-divergence-refine-def failure-refine-def divergence-refine-def*, *safe*)
from *initials-memI*[*of* ‹$ev\ a$› - Q, *simplified initial-hyps*(*2*)]
show ‹$t \in \mathcal{D}\ ((P \triangle Q)\ after\ a) \Longrightarrow t \in \mathcal{D}\ (P\ after\ a \triangle Q)$› **for** t
by (*simp add*: *After-projs initials-Interrupt initial-hyps D-Interrupt*)
(*metis append-Nil hd-append2 list.exhaust-sel D-T*
list.sel(*1*, *3*) *tickFree-Cons-iff tl-append2*)
next
fix $t\ X$
assume ‹$(t, X) \in \mathcal{F}\ ((P \triangle Q)\ after\ a)$›

with *initial-hyps*(*1*) **have** ‹(*ev a # t*, *X*) ∈ $\mathcal{F}$ (*P* △ *Q*)›
by (*simp add*: *F-After initials-Interrupt*)
with *initials-memI*[*of* ‹*ev a*› - *Q*, *simplified initial-hyps*(*2*)]
show ‹(*t*, *X*) ∈ $\mathcal{F}$ (*P after a* △ *Q*)›
by (*simp add*: *F-Interrupt After-projs initial-hyps*(*1*), *elim disjE*)
(*metis* (*no-types*, *opaque-lifting*) [[*metis-verbose* = *false*]]
append-self-conv2 event$_{ptick}$*.simps*(*3*) *tl-append2 F-T list.exhaust-sel*
list.sel(*1*, *3*) *tickFree-Cons-iff D-T hd-append2*)+
qed

moreover have ‹(*Q after a*) ⊓ (*P after a* △ *Q*) ⊑$_{FD}$ (*P* △ *Q*) *after a*›
if *both-initial*: ‹*ev a* ∈ *P*^{0}› ‹*ev a* ∈ *Q*^{0}›
proof (*unfold failure-divergence-refine-def failure-refine-def divergence-refine-def*, *safe*)
fix *t* **assume** ‹*t* ∈ $\mathcal{D}$ ((*P* △ *Q*) *after a*)›
with *both-initial*(*1*) **have** ‹*ev a # t* ∈ $\mathcal{D}$ (*P* △ *Q*)›
by (*simp add*: *D-After initials-Interrupt*)
thus ‹*t* ∈ $\mathcal{D}$ ((*Q after a*) ⊓ (*P after a* △ *Q*))›
by (*simp add*: *D-Interrupt After-projs D-Ndet both-initial*)
(*metis tickFree-tl append-Cons append-Nil list.exhaust-sel list.sel*(*1*, *3*))
next
fix *t X* **assume** ‹(*t*, *X*) ∈ $\mathcal{F}$ ((*P* △ *Q*) *after a*)›
with *both-initial*(*1*) **have** ‹(*ev a # t*, *X*) ∈ $\mathcal{F}$ (*P* △ *Q*)›
by (*simp add*: *F-After initials-Interrupt*)
thus ‹(*t*, *X*) ∈ $\mathcal{F}$ ((*Q after a*) ⊓ (*P after a* △ *Q*))›
by (*simp add*: *F-Interrupt F-Ndet After-projs both-initial*, *elim disjE*)
(*metis* (*no-types*, *opaque-lifting*) [[*metis-verbose* = *false*]]
event$_{ptick}$*.distinct*(*1*) *list.sel*(*1*, *3*) *append-Nil hd-append2*
list.exhaust-sel process-charn tickFree-Cons-iff tl-append2)+
qed

ultimately show *?thesis*
by (*auto simp add*: *not-initial-After initials-Interrupt*
intro: *FD-antisym*)
(*metis mono-Ndet-FD FD-antisym Ndet-id*)
qed

3.7 Behaviour of *After* with Reference Processes

lemma *After-DF*:
‹*DF A after a* = (*if a* ∈ *A then DF A else* Ψ (*DF A*) *a*)›
by (*simp add*: *not-initial-After initials-DF image-iff*)
(*subst DF-unfold*, *simp add*: *After-Mndetprefix*)

lemma *After-DF*$_{SKIPS}$:
‹*DF*$_{SKIPS}$ *A R after a* = (*if a* ∈ *A then DF*$_{SKIPS}$ *A R else* Ψ (*DF*$_{SKIPS}$ *A R*) *a*)›
by (*simp add*: *not-initial-After initials-DF*$_{SKIPS}$ *image-iff*)
(*subst DF*$_{SKIPS}$*-unfold*,

simp add: *After-Ndet After-Mndetprefix initials-Mndetprefix image-iff*)

lemma *After-RUN*:
‹*RUN A after a* = (*if a* ∈ *A then RUN A else* Ψ (*RUN A*) *a*)›
by (*simp add*: *not-initial-After initials-RUN image-iff*)
(*subst RUN-unfold*, *subst After-Mprefix*, *simp*)

lemma *After-CHAOS*:
‹*CHAOS A after a* = (*if a* ∈ *A then CHAOS A else* Ψ (*CHAOS A*) *a*)›
by (*simp add*: *not-initial-After initials-CHAOS image-iff*)
(*subst CHAOS-unfold*,
simp add: *After-Ndet initials-Mprefix After-Mprefix*)

lemma *After-CHAOS*$_{SKIPS}$:
‹*CHAOS*$_{SKIPS}$ *A R after a* = (*if a* ∈ *A then CHAOS*$_{SKIPS}$ *A R else* Ψ (*CHAOS*$_{SKIPS}$ *A R*) *a*)›
by (*simp add*: *not-initial-After initials-CHAOS*$_{SKIPS}$ *image-iff*)
(*subst CHAOS*$_{SKIPS}$*-unfold*,
simp add: *initials-Ndet initials-Mprefix After-Ndet After-Mprefix image-iff*)

lemma *DF-FD-After*: ‹*DF A* ⊑$_{FD}$ *P after a*› **if** ‹*ev a* ∈ *P*^{0}› **and** ‹*DF A* ⊑$_{FD}$ *P*›
proof −
have ‹*DF A after a* ⊑$_{FD}$ *P after a*› **by** (*rule mono-After-FD*[*OF that*])
also have ‹*a* ∈ *A*›
using *anti-mono-initials-FD*[*OF that*(*2*), *THEN set-mp*, *OF that*(*1*)]
by (*simp add*: *initials-DF image-iff*)
ultimately show ‹*DF A* ⊑$_{FD}$ *P after a*› **by** (*subst* (*asm*) *After-DF*, *simp split*: *if-split-asm*)
qed

lemma *DF*$_{SKIPS}$*-FD-After*: ‹*DF*$_{SKIPS}$ *A R* ⊑$_{FD}$ *P after a*› **if** ‹*ev a* ∈ *P*^{0}› **and** ‹*DF*$_{SKIPS}$ *A R* ⊑$_{FD}$ *P*›
proof −
have ‹*DF*$_{SKIPS}$ *A R after a* ⊑$_{FD}$ *P after a*› **by** (*rule mono-After-FD*[*OF that*])

also have ‹*a* ∈ *A*›
using *anti-mono-initials-FD*[*OF that*(*2*), *THEN set-mp*, *OF that*(*1*)]
by (*simp add*: *initials-DF*$_{SKIPS}$ *image-iff*)
ultimately show ‹*DF*$_{SKIPS}$ *A R* ⊑$_{FD}$ *P after a*› **by** (*subst* (*asm*) *After-DF*$_{SKIPS}$, *simp split*: *if-split-asm*)
qed

We have corollaries on *deadlock-free* and *deadlock-free*$_{SKIPS}$.

corollary *deadlock-free-After*:
‹*deadlock-free P* ⟹
deadlock-free (*P after a*) ⟷

```
  (if ev a ∈ P⁰ then True else deadlock-free (Ψ P a))›
 by (simp add: not-initial-After deadlock-free-def)
  (intro impI DF-FD-After)

corollary deadlock-free_SKIPS-After:
 ‹deadlock-free_SKIPS P ⟹
  deadlock-free_SKIPS (P after a) ⟷
  (if ev a ∈ P⁰ then True else deadlock-free_SKIPS (Ψ P a))›
 by (simp add: not-initial-After deadlock-free_SKIPS-FD)
  (intro impI DF_SKIPS-FD-After)
```

3.8 Continuity

This is a new result whose main consequence will be the admissibility of the event transition that is defined later (property that paves the way for point-fixed induction)...

Of course this result will require an additional assumption of continuity on the placeholder Ψ.

```
context begin


private lemma mono-Ψ-imp-chain-After:
 ‹(⋀P Q. P ⊑ Q ⟹ Ψ P a ⊑ Ψ Q a) ⟹ chain Y ⟹ chain (λi. Y i after a)›
 by (simp add: mono-After chain-def)

private lemma cont-prem-After :
 ‹(⨆ i. Y i) after a = (⨆ i. Y i after a)› (is ‹?lhs = ?rhs›)
 if cont-Ψ: ‹cont (λP. Ψ P a)› and chain-Y : ‹chain Y›
proof −
 from chain-Y cont2monofunE cont-Ψ mono-Ψ-imp-chain-After
 have chain-After : ‹chain (λi. Y i after a)› by blast
 show ‹?lhs = ?rhs›
 proof (cases ‹ev a ∈ (⨆ i. Y i)⁰›)
  assume initial : ‹ev a ∈ (⨆ i. Y i)⁰›
  hence ∗ : ‹∀ i. ev a ∈ (Y i)⁰› by (simp add: initials-LUB chain-Y)
  show ‹?lhs = ?rhs›
  proof (rule Process-eq-optimizedI)
   show ‹t ∈ 𝒟 ?lhs ⟹ t ∈ 𝒟 ?rhs› for t
    by (simp add: D-After initial)
     (simp add: D-After ∗ limproc-is-thelub chain-After chain-Y D-LUB)
  next
   fix t
   define S
    where ‹S i ≡ {u. t = u ∧ ev a # u ∈ 𝒟 (Y i)}› for i
   assume ‹t ∈ 𝒟 ?rhs›
   hence ‹t ∈ 𝒟 (Y i after a)› for i
    by (simp add: limproc-is-thelub D-LUB chain-After)
```

```
    hence ‹S i ≠ {}› for i by (simp add: S-def D-After ∗)
  moreover have ‹finite (S 0)› unfolding S-def by (prove-finite-subset-of-prefixes
t)
    moreover have ‹S (Suc i) ⊆ S i› for i
      by (simp add: S-def subset-iff) (metis in-mono le-approx1 po-class.chainE
chain-Y)
  ultimately have ‹(⋂ i. S i) ≠ {}› by (rule Inter-nonempty-finite-chained-sets)
    then obtain t1 where ‹∀ i. t1 ∈ S i›
      by (meson INT-iff ex-in-conv iso-tuple-UNIV-I)
    thus ‹t ∈ 𝒟 ?lhs› by (simp add: S-def D-After initial ∗)
        (simp add: limproc-is-thelub chain-Y D-LUB)
  next
    show ‹(s, X) ∈ ℱ ?lhs ⟹ (s, X) ∈ ℱ ?rhs› for s X
      by (simp add: F-After initial)
        (simp add: F-After ∗ limproc-is-thelub chain-After chain-Y F-LUB)
  next
    fix t X assume ‹t ∉ 𝒟 ?rhs› and ‹(t, X) ∈ ℱ ?rhs›
    from ‹t ∉ 𝒟 ?rhs› obtain j where ‹t ∉ 𝒟 (Y j after a)›
      by (auto simp add: limproc-is-thelub chain-After ‹chain Y› D-LUB)
    moreover from ‹(t, X) ∈ ℱ ?rhs› have ‹(t, X) ∈ ℱ (Y j after a)›
      by (simp add: limproc-is-thelub chain-After ‹chain Y› F-LUB)
    ultimately show ‹(t, X) ∈ ℱ ?lhs›
      using chain-Y initial is-ub-thelub mono-After proc-ord2a by blast
  qed
 next
  assume ‹ev a ∉ (⨆ i. Y i)⁰›
  then obtain k where ∗ : ‹ev a ∉ (Y (i + k))⁰› for i
    by (simp add: initials-LUB chain-Y)
   (metis add.commute anti-mono-initials chain-Y in-mono le-add1 po-class.chain-mono)
   hence ∗∗ : ‹ev a ∉ (⨆ i. Y (i + k))⁰› by (simp add: initials-LUB chain-Y
chain-shift)
  have ‹?lhs = (⨆ i. Y (i + k)) after a› by (simp add: chain-Y lub-range-shift)
  also have ‹... = (⨆ i. Y (i + k) after a)›
    by (simp add: not-initial-After ∗ ∗∗)
      (fact cont2contlubE[OF cont-Ψ chain-shift[OF chain-Y]])
  also from chain-After lub-range-shift have ‹... = ?rhs› by blast
  finally show ‹?lhs = ?rhs› .
 qed
qed


lemma After-cont [simp] :
 ‹cont (λx. f x after a)› if cont-Ψ : ‹cont (λP. Ψ P a)› and cont-f : ‹cont f›
proof (rule contI2)
 show ‹monofun (λx. f x after a)›
  by (rule monofunI) (metis cont2monofunE cont-Ψ cont-f mono-After)
next
 show ‹chain Y ⟹ f (⨆ i. Y i) after a ⊑ (⨆ i. f (Y i) after a)› for Y
  by (simp add: ch2ch-cont cont2contlubE cont-Ψ cont-f cont-prem-After)
```

qed

end

end

Chapter 4

Extension of the After Operator

4.1 The After$_\checkmark$ Operator

locale *AfterExt* = *After* Ψ
  for Ψ :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*a*] ⇒ ('*a*, '*r*) *process*$_{ptick}$ ›
    — Just declaring the types '*a* and '*r*. +
  fixes Ω :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*r*] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
begin

4.1.1 Definition

We just defined *P after e* for *P*::('*a*, '*r*) *process*$_{ptick}$ and *e*::'*a*; in other words we cannot handle ✓(*r*). We now introduce a generalisation for *e*::('*a*, '*r*) *event*$_{ptick}$.

In the previous version, we agreed to get *STOP* after a termination, but only if *P* was not ⊥ since otherwise we kept ⊥. We were not really sure about this choice, and we even introduced a variation where the result after a termination was always *STOP*. In this new version we use a placeholder instead: Ω. We define *P* after ✓(*r*) being equal to Ω *P r*.

For the moment we have no additional assumption on Ω. This will be discussed later.

definition *After*$_{tick}$ :: ‹[('*a*, '*r*) *process*$_{ptick}$, ('*a*, '*r*) *event*$_{ptick}$] ⇒ ('*a*, '*r*) *process*$_{ptick}$› (**infixl** ‹*after*$_\checkmark$› *86*)
  where ‹*P after*$_\checkmark$ *e* ≡ *case e of ev x* ⇒ *P after x* | ✓(*r*) ⇒ Ω *P r*›

lemma *not-initial-After*$_{tick}$:
  ‹*e* ∉ *initials P* ⟹ *P after*$_\checkmark$ *e* = (*case e of ev x* ⇒ Ψ *P x* | ✓(*r*) ⇒ Ω *P r*)›
  by (*auto simp add*: *After*$_{tick}$*-def After-BOT intro*: *not-initial-After split*: *event*$_{ptick}$*.split*)

lemma *initials-After*$_{tick}$:
‹(*P after*$_{\checkmark}$ *e*)^0 =
(*case e of* ✓(*r*) ⇒ (Ω *P r*)^0
| *ev a* ⇒ *if ev a* ∈ *P*^{0} *then* {*e*. [*ev a*, *e*] ∈ $\mathcal{T}$ *P*} *else* (Ψ *P a*)^0)›
by (*simp add*: *After*$_{tick}$*-def initials-After split*: *event*$_{ptick}$*.split*)

4.1.2 Projections

lemma *F-After*$_{tick}$:
‹$\mathcal{F}$ (*P after*$_{\checkmark}$ *e*) =
(*case e of* ✓(*r*) ⇒ $\mathcal{F}$ (Ω *P r*)
| *ev a* ⇒ *if ev a* ∈ *P*^{0} *then* {(*s*, *X*). (*ev a* # *s*, *X*) ∈ $\mathcal{F}$ *P*} *else* $\mathcal{F}$ (Ψ *P a*)›
by (*simp add*: *After*$_{tick}$*-def F-After split*: *event*$_{ptick}$*.split*)

lemma *D-After*$_{tick}$:
‹$\mathcal{D}$ (*P after*$_{\checkmark}$ *e*) =
(*case e of* ✓(*r*) ⇒ $\mathcal{D}$ (Ω *P r*)
| *ev a* ⇒ *if ev a* ∈ *P*^{0} *then* {*s*. *ev a* # *s* ∈ $\mathcal{D}$ *P*} *else* $\mathcal{D}$ (Ψ *P a*))›
by (*simp add*: *After*$_{tick}$*-def D-After split*: *event*$_{ptick}$*.split*)

lemma *T-After*$_{tick}$:
‹$\mathcal{T}$ (*P after*$_{\checkmark}$ *e*) =
(*case e of* ✓(*r*) ⇒ $\mathcal{T}$ (Ω *P r*)
| *ev a* ⇒ *if ev a* ∈ *P*^{0} *then* {*s*. *ev a* # *s* ∈ $\mathcal{T}$ *P*} *else* $\mathcal{T}$ (Ψ *P a*))›
by (*simp add*: *After*$_{tick}$*-def T-After split*: *event*$_{ptick}$*.split*)

4.1.3 Monotony

lemma *mono-After*$_{tick}$*-T* :
‹*e* ∈ *Q*^{0} ⟹ (*case e of* ✓(*r*) ⇒ Ω *P r* ⊑$_T$ Ω *Q r*) ⟹ *P* ⊑$_T$ *Q* ⟹ *P after*$_{\checkmark}$ *e* ⊑$_T$ *Q after*$_{\checkmark}$ *e*›
and *mono-After*$_{tick}$*-F* :
‹*e* ∈ *Q*^{0} ⟹ (*case e of* ✓(*r*) ⇒ Ω *P r* ⊑$_F$ Ω *Q r*) ⟹ *P* ⊑$_F$ *Q* ⟹ *P after*$_{\checkmark}$ *e* ⊑$_F$ *Q after*$_{\checkmark}$ *e*›
and *mono-After*$_{tick}$*-D* :
‹*e* ∈ *Q*^{0} ⟹ (*case e of* ✓(*r*) ⇒ Ω *P r* ⊑$_D$ Ω *Q r*) ⟹ *P* ⊑$_D$ *Q* ⟹ *P after*$_{\checkmark}$ *e* ⊑$_D$ *Q after*$_{\checkmark}$ *e*›
and *mono-After*$_{tick}$*-FD* :
‹*e* ∈ *Q*^{0} ⟹ (*case e of* ✓(*r*) ⇒ Ω *P r* ⊑$_{FD}$ Ω *Q r*) ⟹ *P* ⊑$_{FD}$ *Q* ⟹ *P after*$_{\checkmark}$ *e* ⊑$_{FD}$ *Q after*$_{\checkmark}$ *e*›
and *mono-After*$_{tick}$*-DT* :
‹*e* ∈ *Q*^{0} ⟹ (*case e of* ✓(*r*) ⇒ Ω *P r* ⊑$_{DT}$ Ω *Q r*) ⟹ *P* ⊑$_{DT}$ *Q* ⟹ *P after*$_{\checkmark}$ *e* ⊑$_{DT}$ *Q after*$_{\checkmark}$ *e*›
using *mono-After-T mono-After-F mono-After-D mono-After-FD mono-After-DT*
by (*auto simp add*: *After*$_{tick}$*-def split*: *event*$_{ptick}$*.split*)

lemma *mono-After*$_{tick}$:
‹⟦*P* ⊑ *Q*;

$(case\ e\ of\ ev\ a \Rightarrow (ev\ a \in Q^0 \vee \Psi\ P\ a \sqsubseteq \Psi\ Q\ a));$
$(case\ e\ of\ \checkmark(r) \Rightarrow \Omega\ P\ r \sqsubseteq \Omega\ Q\ r)]\!] \Longrightarrow$
$P\ after_{\checkmark}\ e \sqsubseteq Q\ after_{\checkmark}\ e\rangle$
by (*force simp add*: $After_{tick}$*-def split*: $event_{ptick}$*.split intro*: *mono-After*)

4.1.4 Behaviour of $After_{tick}$ with $STOP$, $SKIP$ and $\bot$

lemma $After_{tick}$*-STOP*: $\langle STOP\ after_{\checkmark}\ e = (case\ e\ of\ ev\ a \Rightarrow \Psi\ STOP\ a \mid \checkmark(r) \Rightarrow \Omega\ STOP\ r)\rangle$
and $After_{tick}$*-SKIP*: $\langle SKIP\ r\ after_{\checkmark}\ e = (case\ e\ of\ ev\ a \Rightarrow \Psi\ (SKIP\ r)\ a \mid \checkmark(s) \Rightarrow \Omega\ (SKIP\ r)\ s)\rangle$
and $After_{tick}$*-BOT* : $\langle \bot\ after_{\checkmark}\ e = (case\ e\ of\ ev\ x \Rightarrow \bot \mid \checkmark(r) \Rightarrow \Omega\ \bot\ r)\rangle$
by (*simp-all add*: $After_{tick}$*-def After-STOP After-SKIP After-BOT split*: $event_{ptick}$*.split*)

lemma $After_{tick}$*-is-BOT-iff*:
$\langle P\ after_{\checkmark}\ e = \bot \longleftrightarrow$
$(case\ e\ of\ \checkmark(r) \Rightarrow \Omega\ P\ r = \bot$
$\mid ev\ a \Rightarrow if\ ev\ a \in P^0\ then\ [ev\ a] \in \mathcal{D}\ P\ else\ \Psi\ P\ a = \bot)\rangle$
by (*simp add*: $After_{tick}$*-def After-is-BOT-iff split*: $event_{ptick}$*.split*)

4.1.5 Behaviour of $After_{tick}$ with Operators of HOL-CSP

Here again, we lose determinism.

lemma $After_{tick}$*-Mprefix-is-After*$_{tick}$*-Mndetprefix*:
$\langle a \in A \Longrightarrow (\Box a \in A \rightarrow P\ a)\ after_{\checkmark}\ ev\ a = (\sqcap a \in A \rightarrow P\ a)\ after_{\checkmark}\ ev\ a\rangle$
by (*simp add*: $After_{tick}$*-def After-Mprefix-is-After-Mndetprefix*)

lemma $After_{tick}$*-Det-is-After*$_{tick}$*-Ndet*:
$\langle ev\ a \in P^0 \cup Q^0 \Longrightarrow (P \Box Q)\ after_{\checkmark}\ ev\ a = (P \sqcap Q)\ after_{\checkmark}\ ev\ a\rangle$
by (*simp add*: $After_{tick}$*-def After-Det-is-After-Ndet*)

lemma $After_{tick}$*-Sliding-is-After*$_{tick}$*-Ndet*:
$\langle ev\ a \in P^0 \cup Q^0 \Longrightarrow (P \rhd Q)\ after_{\checkmark}\ ev\ a = (P \sqcap Q)\ after_{\checkmark}\ ev\ a\rangle$
by (*simp add*: $After_{tick}$*-def After-Sliding-is-After-Ndet*)

lemma $After_{tick}$*-Ndet*:
$\langle (P \sqcap Q)\ after_{\checkmark}\ e =$
$(case\ e\ of\ \checkmark(r) \Rightarrow \Omega\ (P \sqcap Q)\ r$
$\mid ev\ a \Rightarrow\ \ if\ ev\ a \in P^0 \cap Q^0$
$then\ P\ after_{\checkmark}\ ev\ a \sqcap Q\ after_{\checkmark}\ ev\ a$
$else\ \ if\ ev\ a \in P^0$
$then\ P\ after_{\checkmark}\ ev\ a$
$else\ \ if\ ev\ a \in Q^0$
$then\ Q\ after_{\checkmark}\ ev\ a$
$else\ \Psi\ (P \sqcap Q)\ a)\rangle$
by (*simp add*: $After_{tick}$*-def After-Ndet split*: $event_{ptick}$*.split*)

lemma *After$_{tick}$-Det*:
‹(*P* □ *Q*) *after*$_{\checkmark}$ *e* =
(*case e of* ✓(*r*) ⇒ Ω (*P* □ *Q*) *r*
| *ev a* ⇒ *if ev a* ∈ P^0 ∩ Q^0
then P after$_{\checkmark}$ *ev a* ⊓ *Q after*$_{\checkmark}$ *ev a*
else if ev a ∈ P^0
then P after$_{\checkmark}$ *ev a*
else if ev a ∈ Q^0
then Q after$_{\checkmark}$ *ev a*
else Ψ (*P* □ *Q*) *a*)›
by (*simp add*: *After$_{tick}$-def After-Det split*: *event$_{ptick}$.split*)

lemma *After$_{tick}$-Sliding*:
‹(*P* ▷ *Q*) *after*$_{\checkmark}$ *e* =
(*case e of* ✓(*r*) ⇒ Ω (*P* ▷ *Q*) *r*
| *ev a* ⇒ *if ev a* ∈ P^0 ∩ Q^0
then P after$_{\checkmark}$ *ev a* ⊓ *Q after*$_{\checkmark}$ *ev a*
else if ev a ∈ P^0
then P after$_{\checkmark}$ *ev a*
else if ev a ∈ Q^0
then Q after$_{\checkmark}$ *ev a*
else Ψ (*P* ▷ *Q*) *a*)›
by (*simp add*: *After$_{tick}$-def After-Sliding split*: *event$_{ptick}$.split*)

lemma *After$_{tick}$-Mprefix*:
‹(□ *a* ∈ *A* → *P a*) *after*$_{\checkmark}$ *e* =
(*case e of* ✓(*r*) ⇒ Ω (□ *a* ∈ *A* → *P a*) *r*
| *ev a* ⇒ *if a* ∈ *A then P a else* Ψ (□ *a* ∈ *A* → *P a*) *a*)›
by (*simp add*: *After$_{tick}$-def After-Mprefix split*: *event$_{ptick}$.split*)

lemma *After$_{tick}$-Mndetprefix*:
‹(⊓ *a* ∈ *A* → *P a*) *after*$_{\checkmark}$ *e* =
(*case e of* ✓(*r*) ⇒ Ω (⊓ *a* ∈ *A* → *P a*) *r*
| *ev a* ⇒ *if a* ∈ *A then P a else* Ψ (⊓ *a* ∈ *A* → *P a*) *a*)›
by (*simp add*: *After$_{tick}$-def After-Mndetprefix split*: *event$_{ptick}$.split*)

corollary *After$_{tick}$-write0*:
‹(*a* → *P*) *after*$_{\checkmark}$ *e* =
(*case e of* ✓(*r*) ⇒ Ω (*a* → *P*) *r*
| *ev b* ⇒ *if b* = *a then P else* Ψ (*a* → *P*) *b*)›
by (*simp add*: *After$_{tick}$-def After-write0 split*: *event$_{ptick}$.split*)

corollary ‹(*a* → *P*) *after*$_{\checkmark}$ *ev a* = *P*› **by** (*simp add*: *After$_{tick}$-write0*)

corollary *After$_{tick}$-read*:
‹(*c*?*a*∈*A* → *P a*) *after*$_{\checkmark}$ *e* =
(*case e of* ✓(*r*) ⇒ Ω (*c*?*a*∈*A* → *P a*) *r*
| *ev b* ⇒ *if b* ∈ *c* ‘ *A then P* (*inv-into A c b*) *else* Ψ (*c*?*a*∈*A* → *P a*) *b*)›

by (*simp add*: $After_{tick}$*-def After-read split*: $event_{ptick}$*.split*)

corollary $After_{tick}$*-ndet-write*:
‹(*c*!!*a*∈*A* → *P a*) *after*✓ *e* =
(*case e of* ✓(*r*) ⇒ Ω (*c*!!*a*∈*A* → *P a*) *r*
| *ev b* ⇒ *if b* ∈ *c* ‘ *A then P* (*inv-into A c b*) *else* Ψ (*c*!!*a*∈*A* → *P a*) *b*)›
by (*simp add*: $After_{tick}$*-def After-ndet-write split*: $event_{ptick}$*.split*)

lemma $After_{tick}$*-Seq*:
‹(*P* ; *Q*) *after*✓ *e* =
(*case e of* ✓(*r*) ⇒ Ω (*P* ; *Q*) *r*
| *ev a* ⇒ *if range tick* ∩ P^0 = {} ∨ (∀ *r*. ✓(*r*) ∈ P^0 ⟶ *ev a* ∉ Q^0)
then if ev a ∈ P^0 *then P after*✓ *ev a* ; *Q else* Ψ (*P* ; *Q*) *a*
else if ev a ∈ P^0 *then* (*P after*✓ *ev a* ; *Q*) ⊓ *Q after*✓ *ev a*
else Q after✓ *ev a*)›
by (*simp add*: $After_{tick}$*-def After-Seq split*: $event_{ptick}$*.split*)

lemma $After_{tick}$*-Sync*:
‹(*P* ⟦*S*⟧ *Q*) *after*✓ *e* =
(*case e of* ✓(*r*) ⇒ Ω (*P* ⟦*S*⟧ *Q*) *r*
| *ev a* ⇒ *if P* = ⊥ ∨ *Q* = ⊥ *then* ⊥
else if ev a ∈ P^0 ∩ Q^0
then if a ∈ *S then P after*✓ *ev a* ⟦*S*⟧ *Q after*✓ *ev a*
else (*P after*✓ *ev a* ⟦*S*⟧ *Q*) ⊓ (*P* ⟦*S*⟧ *Q after*✓ *ev a*)
else if ev a ∈ P^0 ∧ *a* ∉ *S then P after*✓ *ev a* ⟦*S*⟧ *Q*
else if ev a ∈ Q^0 ∧ *a* ∉ *S then P* ⟦*S*⟧ *Q after*✓ *ev a*
else Ψ (*P* ⟦*S*⟧ *Q*) *a*)›
by (*simp add*: $After_{tick}$*-def After-Sync split*: $event_{ptick}$*.split*)

lemma *Hiding-FD-Hiding-*$After_{tick}$*-if-initial-inside*:
‹*ev a* ∈ P^0 ⟹ *a* ∈ *A* ⟹ *P* \ *A* ⊑$_{FD}$ *P after*✓ *ev a* \ *A*›
and $After_{tick}$*-Hiding-FD-Hiding-*$After_{tick}$*-if-initial-notin*:
‹*ev a* ∈ P^0 ⟹ *a* ∉ *A* ⟹ (*P* \ *A*) *after*✓ *ev a* ⊑$_{FD}$ *P after*✓ *ev a* \ *A*›
by (*simp add*: $After_{tick}$*-def Hiding-FD-Hiding-After-if-initial-inside*)
(*simp add*: $After_{tick}$*-def After-Hiding-FD-Hiding-After-if-initial-notin*)

lemmas *Hiding-F-Hiding-*$After_{tick}$*-if-initial-inside* =
Hiding-FD-Hiding-$After_{tick}$*-if-initial-inside*[*THEN leFD-imp-leF*]
and $After_{tick}$*-Hiding-F-Hiding-*$After_{tick}$*-if-initial-notin* =
$After_{tick}$*-Hiding-FD-Hiding-*$After_{tick}$*-if-initial-notin*[*THEN leFD-imp-leF*]
and *Hiding-D-Hiding-*$After_{tick}$*-if-initial-inside* =
Hiding-FD-Hiding-$After_{tick}$*-if-initial-inside*[*THEN leFD-imp-leD*]
and $After_{tick}$*-Hiding-D-Hiding-*$After_{tick}$*-if-initial-notin* =
$After_{tick}$*-Hiding-FD-Hiding-*$After_{tick}$*-if-initial-notin*[*THEN leFD-imp-leD*]
and *Hiding-T-Hiding-*$After_{tick}$*-if-initial-inside* =

Hiding-FD-Hiding-After$_{tick}$-if-initial-inside[*THEN leFD-imp-leF*, *THEN leF-imp-leT*]

and *After$_{tick}$-Hiding-T-Hiding-After$_{tick}$-if-initial-notin* = *After$_{tick}$-Hiding-FD-Hiding-After$_{tick}$-if-initial-notin*[*THEN leFD-imp-leF*, *THEN leF-imp-leT*]

corollary *Hiding-DT-Hiding-After$_{tick}$-if-initial-inside*:
‹*ev a* ∈ P^0 ⟹ *a* ∈ *A* ⟹ *P* \ *A* ⊑$_{DT}$ *P after*$_{✓}$ *ev a* \ *A*›
and *After$_{tick}$-Hiding-DT-Hiding-After$_{tick}$-if-initial-notin*:
‹*ev a* ∈ P^0 ⟹ *a* ∉ *A* ⟹ (*P* \ *A*) *after*$_{✓}$ *ev a* ⊑$_{DT}$ *P after*$_{✓}$ *ev a* \ *A*›
by (*simp add*: *Hiding-D-Hiding-After$_{tick}$-if-initial-inside Hiding-T-Hiding-After$_{tick}$-if-initial-inside leD-leT-imp-leDT*)
(*simp add*: *After$_{tick}$-Hiding-D-Hiding-After$_{tick}$-if-initial-notin After$_{tick}$-Hiding-T-Hiding-After$_{tick}$-if-initial-notin leD-leT-imp-leDT*)

end

As with *After*, we need to "duplicate" the locale to formalize the result for the *Renaming* operator.

locale *AfterExtDuplicated* = *After$_{tick\alpha}$*: *AfterExt* Ψ_α Ω_α + *After$_{tick\beta}$*: *AfterExt* Ψ_β Ω_β
for Ψ_α :: ‹[('*a*, '*r*) *process$_{ptick}$*, '*a*] ⇒ ('*a*, '*r*) *process$_{ptick}$*›
and Ω_α :: ‹[('*a*, '*r*) *process$_{ptick}$*, '*r*] ⇒ ('*a*, '*r*) *process$_{ptick}$*›
and Ψ_β :: ‹[('*b*, '*s*) *process$_{ptick}$*, '*b*] ⇒ ('*b*, '*s*) *process$_{ptick}$*›
and Ω_β :: ‹[('*b*, '*s*) *process$_{ptick}$*, '*s*] ⇒ ('*b*, '*s*) *process$_{ptick}$*›

sublocale *AfterExtDuplicated* ⊆ *AfterDuplicated* **.**
— Recovering *AfterDuplicated.After-Renaming*

context *AfterExtDuplicated*
begin

notation *After$_{tick\alpha}$.After* (**infixl** ‹*after*$_\alpha$› *86*)
notation *After$_{tick\beta}$.After* (**infixl** ‹*after*$_\beta$› *86*)
notation *After$_{tick\alpha}$.After$_{tick}$* (**infixl** ‹*after*$_{✓\alpha}$› *86*)
notation *After$_{tick\beta}$.After$_{tick}$* (**infixl** ‹*after*$_{✓\beta}$› *86*)

lemma *After$_{tick}$-Renaming*:
‹*Renaming P f g after*$_{✓\beta}$ *e* =
(*case e of* ✓(*s*) ⇒ Ω_β (*Renaming P f g*) *s*
| *ev b* ⇒ *if P* = ⊥ *then* ⊥
else if ∃ *a*. *ev a* ∈ P^0 ∧ *f a* = *b*
then ⊓ *a*∈{*a*. *ev a* ∈ P^0 ∧ *f a* = *b*}. *Renaming* (*P after*$_\alpha$ *a*) *f g*
else Ψ_β (*Renaming P f g*) *b*)›
by (*auto simp add*: *After$_{tick\alpha}$.After$_{tick}$-def After$_{tick\beta}$.After$_{tick}$-def After-Renaming split*: *event$_{ptick}$.split*)

end

context *AfterExt* — Back to *AfterExt*.
begin

4.1.6 Behaviour of $After_{tick}$ with Operators of HOL-CSPM

lemma *$After_{tick}$-GlobalDet-is-$After_{tick}$-GlobalNdet*:
‹*ev a* ∈ (⋃ *a* ∈ *A*. (*P a*)^0) ⟹
(□ *a* ∈ *A*. *P a*) *after*✓ *ev a* = (⊓ *a* ∈ *A*. *P a*) *after*✓ *ev a*›
by (*simp add*: *$After_{tick}$-def After-GlobalDet-is-After-GlobalNdet*)

lemma *$After_{tick}$-GlobalNdet*:
‹(⊓ *a* ∈ *A*. *P a*) *after*✓ *e* =
(*case e of* ✓(*r*) ⇒ Ω (⊓ *a* ∈ *A*. *P a*) *r*
| *ev a* ⇒ *if ev a* ∈ (⋃ *a* ∈ *A*. (*P a*)^0)
then ⊓ *x* ∈ {*x* ∈ *A*. *ev a* ∈ (*P x*)^0}. *P x after*✓ *ev a*
else Ψ (⊓ *a* ∈ *A*. *P a*) *a*)›
and *$After_{tick}$-GlobalDet*:
‹(□ *a* ∈ *A*. *P a*) *after*✓ *e* =
(*case e of* ✓(*r*) ⇒ Ω (□ *a* ∈ *A*. *P a*) *r*
| *ev a* ⇒ *if ev a* ∈ (⋃ *a* ∈ *A*. (*P a*)^0)
then ⊓ *x* ∈ {*x* ∈ *A*. *ev a* ∈ (*P x*)^0}. *P x after*✓ *ev a*
else Ψ (□ *a* ∈ *A*. *P a*) *a*)›
by (*simp-all add*: *$After_{tick}$-def After-GlobalNdet After-GlobalDet split*: *$event_{ptick}$.split*)

lemma *$After_{tick}$-Throw*:
‹(*P* Θ *a* ∈ *A*. *Q a*) *after*✓ *e* =
(*case e of* ✓(*r*) ⇒ Ω (*P* Θ *a* ∈ *A*. *Q a*) *r*
| *ev a* ⇒ *if P* = ⊥ *then* ⊥
else if ev a ∈ *P*^{0}
then if a ∈ *A*
then Q a
else P after✓ *ev a* Θ *a* ∈ *A*. *Q a*
else Ψ (*P* Θ *a* ∈ *A*. *Q a*) *a*)›
by (*simp add*: *$After_{tick}$-def After-Throw split*: *$event_{ptick}$.split*)

lemma *$After_{tick}$-Interrupt*:
‹(*P* △ *Q*) *after*✓ *e* =
(*case e of* ✓(*r*) ⇒ Ω (*P* △ *Q*) *r*
| *ev a* ⇒ *if ev a* ∈ *P*^{0} ∩ *Q*^{0}
then Q after✓ *ev a* ⊓ (*P after*✓ *ev a* △ *Q*)
else if ev a ∈ *P*^{0} ∧ *ev a* ∉ *Q*^{0}
then P after✓ *ev a* △ *Q*

else if ev a ∉ P⁰ ∧ ev a ∈ Q⁰
then Q after✓ ev a
else Ψ (P △ Q) a)›
by (*simp add*: *After$_{tick}$-def After-Interrupt split*: *event$_{ptick}$.split*)

4.1.7 Behaviour of $After_{tick}$ with Reference Processes

lemma *After$_{tick}$-DF*:
‹DF A after✓ e =
(case e of ✓(r) ⇒ Ω (DF A) r
| ev a ⇒ if a ∈ A then DF A else Ψ (DF A) a)›
by (*simp add*: *After$_{tick}$-def After-DF split*: *event$_{ptick}$.split*)

lemma *After$_{tick}$-DF$_{SKIPS}$*:
‹DF$_{SKIPS}$ A R after✓ e =
(case e of ✓(r) ⇒ Ω (DF$_{SKIPS}$ A R) r
| ev a ⇒ if a ∈ A then DF$_{SKIPS}$ A R else Ψ (DF$_{SKIPS}$ A R) a)›
by (*simp add*: *After$_{tick}$-def After-DF$_{SKIPS}$ split*: *event$_{ptick}$.split*)

lemma *After$_{tick}$-RUN*:
‹RUN A after✓ e =
(case e of ✓(r) ⇒ Ω (RUN A) r
| ev a ⇒ if a ∈ A then RUN A else Ψ (RUN A) a)›
by (*simp add*: *After$_{tick}$-def After-RUN split*: *event$_{ptick}$.split*)

lemma *After$_{tick}$-CHAOS*:
‹CHAOS A after✓ e =
(case e of ✓(r) ⇒ Ω (CHAOS A) r
| ev a ⇒ if a ∈ A then CHAOS A else Ψ (CHAOS A) a)›
by (*simp add*: *After$_{tick}$-def After-CHAOS split*: *event$_{ptick}$.split*)

lemma *After$_{tick}$-CHAOS$_{SKIPS}$*:
‹CHAOS$_{SKIPS}$ A R after✓ e =
(case e of ✓(r) ⇒ Ω (CHAOS$_{SKIPS}$ A R) r
| ev a ⇒ if a ∈ A then CHAOS$_{SKIPS}$ A R else Ψ (CHAOS$_{SKIPS}$ A R) a)›
by (*simp add*: *After$_{tick}$-def After-CHAOS$_{SKIPS}$ split*: *event$_{ptick}$.split*)

lemma *DF-FD-After$_{tick}$*:
‹DF A ⊑$_{FD}$ P ⟹ e ∈ P⁰ ⟹ DF A ⊑$_{FD}$ P after✓ e›
by (*cases e*, *simp add*: *After$_{tick}$-def DF-FD-After*)
(*metis anti-mono-initials-FD event$_{ptick}$.distinct(1) image-iff initials-DF subsetD*)

lemma *DF$_{SKIPS}$-FD-After$_{tick}$*:
‹DF$_{SKIPS}$ A R ⊑$_{FD}$ P ⟹ ev a ∈ P⁰ ⟹ DF$_{SKIPS}$ A R ⊑$_{FD}$ P after✓ ev a›
by (*simp add*: *After$_{tick}$-def DF$_{SKIPS}$-FD-After*)

lemma *deadlock-free-After*$_{tick}$:
‹$e \in P^0 \Longrightarrow$ *deadlock-free* $P \Longrightarrow$
deadlock-free (P *after*$_{\checkmark}$ e) $\longleftrightarrow$
(*case* e *of ev a* $\Rightarrow$ *True* | $\checkmark(r) \Rightarrow$ *deadlock-free* (Ω P r))›
using *deadlock-free-After* **by** (*auto simp add*: *After*$_{tick}$*-def split*: *event*$_{ptick}$*.split*)

lemma *deadlock-free*$_{SKIPS}$*-After*$_{tick}$:
‹$e \in P^0 \Longrightarrow$ *deadlock-free*$_{SKIPS}$ $P \Longrightarrow$
deadlock-free$_{SKIPS}$ (P *after*$_{\checkmark}$ e) $\longleftrightarrow$
(*case* e *of ev a* $\Rightarrow$ *True* | $\checkmark(r) \Rightarrow$ *deadlock-free*$_{SKIPS}$ (Ω P r))›
using *deadlock-free*$_{SKIPS}$*-After* **by** (*auto simp add*: *After*$_{tick}$*-def split*: *event*$_{ptick}$*.split*)

4.1.8 Characterizations for Deadlock Freeness

lemma *deadlock-free-imp-not-initial-tick*: ‹*deadlock-free* $P \Longrightarrow$ *range tick* $\cap$ $P^0 = \{\}$›
by (*rule ccontr*, *simp add*: *disjoint-iff*)
(*meson anti-mono-initials-FD deadlock-free-def initial-tick-iff-FD-SKIP non-deadlock-free-SKIP subsetD*)

lemma *initial-tick-imp-range-ev-in-refusals*: ‹$\checkmark(r) \in P^0 \Longrightarrow$ *range ev* $\in \mathcal{R}$ P›
unfolding *initials-def image-def*
by (*simp add*: *Refusals-iff is-processT6-TR-notin*)

lemma *deadlock-free-After*$_{tick}$*-characterization*:
‹*deadlock-free* $P \longleftrightarrow$ *range ev* $\notin \mathcal{R}$ $P \wedge (\forall e.\ ev\ e \in P^0 \longrightarrow$ *deadlock-free* (P *after*$_{\checkmark}$ *ev e*))›
(**is** ‹*deadlock-free* $P \longleftrightarrow$ *?rhs*›)
proof (*intro iffI*)
from *event*$_{ptick}$*.exhaust-sel* **have** ‹*range ev* = *UNIV* − *range tick*› **by** *blast*
hence ‹*range ev* $\notin \mathcal{R}$ $P \longleftrightarrow$ *UNIV* − *range tick* $\notin \mathcal{R}$ P› **by** *metis*
thus ‹*deadlock-free* $P \Longrightarrow$ *?rhs*›
by (*intro conjI*)
(*solves* ‹*simp add*: *deadlock-free-F failure-refine-def*,
subst (*asm*) *F-DF*, *auto simp add*: *Refusals-iff*›,
use DF-FD-After$_{tick}$ *deadlock-free-def* **in** *blast*)
next
assume ‹*?rhs*›
hence $*$: ‹*range ev* $\notin \mathcal{R}$ P›
‹*ev e* $\in P^0 \Longrightarrow \{(s, X)\ |s\ X.\ (ev\ e\ \#\ s, X) \in \mathcal{F}\ P\} \subseteq \mathcal{F}$ (*DF UNIV*)› **for** e
by (*simp-all add*: *deadlock-free-F failure-refine-def F-After*$_{tick}$ *subset-iff*)
show ‹*deadlock-free* P›
proof (*unfold deadlock-free-F failure-refine-def*, *safe*)
fix s X
assume $**$: ‹$(s, X) \in \mathcal{F}$ P›
show ‹$(s, X) \in \mathcal{F}$ (*DF UNIV*)›

```
    proof (cases s)
      show ‹s = [] ⟹ (s, X) ∈ ℱ (DF UNIV)›
        by (subst F-DF, simp)
          (metis ∗(1) ∗∗ Refusals-iff image-subset-iff is-processT4)
    next
      fix e s′
      assume ∗∗∗ : ‹s = e # s′›
      from ∗∗[THEN F-T, simplified this, THEN initials-memI]
        initial-tick-imp-range-ev-in-refusals ∗(1)
      obtain x where ‹e = ev x› ‹ev x ∈ P⁰›
        by (cases e) (simp-all add: initial-tick-imp-range-ev-in-refusals)
      with ∗(2)[OF this(2)] ∗∗ ∗∗∗ show ‹(s, X) ∈ ℱ (DF UNIV)›
        by (subst F-DF) (simp add: subset-iff)
    qed
  qed
qed



lemma deadlock-free_SKIPS-After_tick-characterization:
  ‹deadlock-free_SKIPS P ⟷
   UNIV ∉ ℛ P ∧ (∀ e ∈ P⁰ − range tick. deadlock-free_SKIPS (P after✓ e))›
  (is ‹deadlock-free_SKIPS P ⟷ ?rhs›)
proof (intro iffI)
  show ?rhs if ‹deadlock-free_SKIPS P›
  proof (intro conjI)
    from ‹deadlock-free_SKIPS P› show ‹UNIV ∉ ℛ P›
      by (simp add: deadlock-free_SKIPS-def failure-refine-def)
        (subst (asm) F-DF_SKIPS, auto simp add: Refusals-iff)
  next
    from ‹deadlock-free_SKIPS P› show ‹∀ e∈P⁰ − range tick. deadlock-free_SKIPS
(P after✓ e)›
      by (auto simp add: deadlock-free_SKIPS-After_tick split: event_ptick.split)
  qed
next
  assume assm : ‹?rhs›
  have ∗ : ‹ev e ∈ P⁰ ⟹ {(s, X) |s X. (ev e # s, X) ∈ ℱ P} ⊆ ℱ (DF_SKIPS
UNIV UNIV)› for e
    using assm[THEN conjunct2, rule-format, of ‹ev e›, simplified]
    by (simp add: deadlock-free_SKIPS-def failure-refine-def F-After_tick image-iff)
  show ‹deadlock-free_SKIPS P›
  proof (unfold deadlock-free_SKIPS-def failure-refine-def, safe)
    fix s X
    assume ∗∗ : ‹(s, X) ∈ ℱ P›
    show ‹(s, X) ∈ ℱ (DF_SKIPS UNIV UNIV)›
    proof (cases s)
      from assm[THEN conjunct1] ∗∗ show ‹s = [] ⟹ (s, X) ∈ ℱ (DF_SKIPS
UNIV UNIV)›
        by (subst F-DF_SKIPS, simp add: Refusals-iff)
```

(*metis UNIV-eq-I* $event_{ptick}$*.exhaust*)
next
fix *e s′*
assume ∗∗∗ : ‹*s* = *e* # *s′*›
show ‹(*s*, *X*) ∈ $\mathcal{F}$ (DF_{SKIPS} *UNIV UNIV*)›
proof (*cases e*)
fix *a* **assume** ‹*e* = *ev a*›
with ∗∗ ∗∗∗ *initials-memI F-T* **have** ‹*ev a* ∈ P^0› **by** *blast*
with ∗[*OF this*] ∗∗ ∗∗∗ ‹*e* = *ev a*› **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ (DF_{SKIPS} *UNIV UNIV*)›
by (*subst F-*DF_{SKIPS}) (*simp add*: *subset-iff*)
next
fix *r* **assume** ‹*e* = ✓(*r*)›
hence ‹*s* = [✓(*r*)]›
by (*metis ∗∗ ∗∗∗* $event_{ptick}$*.disc*(*2*) *front-tickFree-Cons-iff is-processT2*)
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ (DF_{SKIPS} *UNIV UNIV*)›
by (*subst F-*DF_{SKIPS}, *simp*)
qed
qed
qed
qed

4.1.9 Continuity

lemma $After_{tick}$*-cont* [*simp*] :
assumes *cont-*Ψ*Ω* : ‹*case e of ev a* ⇒ *cont* (λ*P*. Ψ *P a*) | ✓(*r*) ⇒ *cont* (λ*P*. Ω *P r*)›
and *cont-f* : ‹*cont f*›
shows ‹*cont* (λ*x*. *f x* $after_{✓}$ *e*)›
by (*cases e*, *simp-all add*: $After_{tick}$*-def*)
(*use After-cont cont-*Ψ*Ω cont-f* **in** ‹*auto intro*: *cont-compose*›)

end

4.2 The After trace Operator

context *AfterExt*
begin

4.2.1 Definition

We just defined *P* $after_{✓}$ *e* for *P*::(′*a*, ′*r*) $process_{ptick}$ and *e*::(′*a*, ′*r*) $event_{ptick}$. Since a trace of a *P* is just an (′*a*, ′*r*) $event_{ptick}$ *list*, the following inductive

definition is natural.

fun $After_{trace}$ :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $trace_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$›
(**infixl** ‹$after_{\mathcal{T}}$› *86*)
where ‹*P* $after_{\mathcal{T}}$ [] = *P*›
| ‹*P* $after_{\mathcal{T}}$ (*e* # *t*) = *P* $after_{\checkmark}$ *e* $after_{\mathcal{T}}$ *t*›

We can also induct backward.

lemma $After_{trace}$-*append*: ‹*P* $after_{\mathcal{T}}$ (*t* @ *u*) = *P* $after_{\mathcal{T}}$ *t* $after_{\mathcal{T}}$ *u*›
apply (*induct u rule*: *rev-induct*, *solves* ‹*simp*›)
apply (*induct t rule*: *rev-induct*, *solves* ‹*simp*›)
by (*metis* $After_{trace}$.*simps append.assoc append.right-neutral append-Cons append-Nil*)

lemma $After_{trace}$-*snoc* : ‹*P* $after_{\mathcal{T}}$ (*t* @ [*e*]) = *P* $after_{\mathcal{T}}$ *t* $after_{\checkmark}$ *e*›
by (*simp add*: $After_{trace}$-*append*)

4.2.2 Projections

lemma *F*-$After_{trace}$:
‹*tF t* ⟹ (*t* @ *u*, *X*) ∈ $\mathcal{F}$ *P* ⟹ (*u*, *X*) ∈ $\mathcal{F}$ (*P* $after_{\mathcal{T}}$ *t*)›
proof (*induct t arbitrary*: *u rule*: *rev-induct*)
show ‹([] @ *u*, *X*) ∈ $\mathcal{F}$ *P* ⟹ (*u*, *X*) ∈ $\mathcal{F}$ (*P* $after_{\mathcal{T}}$ [])› **for** *u* **by** *simp*
next
fix *e t u*
assume *hyp* : ‹*tF t* ⟹ (*t* @ *u*, *X*) ∈ $\mathcal{F}$ *P* ⟹ (*u*, *X*) ∈ $\mathcal{F}$ (*P* $after_{\mathcal{T}}$ *t*)› **for** *u*
assume *prems* : ‹*tF* (*t* @ [*e*])› ‹((*t* @ [*e*]) @ *u*, *X*) ∈ $\mathcal{F}$ *P*›
have ∗ : ‹(*e* # *u*, *X*) ∈ $\mathcal{F}$ (*P* $after_{\mathcal{T}}$ *t*)› **by** (*rule hyp*; *use prems* **in** *simp*)
thus ‹(*u*, *X*) ∈ $\mathcal{F}$ (*P* $after_{\mathcal{T}}$ (*t* @ [*e*]))›
by (*simp add*: $After_{trace}$-*snoc F*-$After_{tick}$ *split*: $event_{ptick}$.*split*)
(*metis initials-memI F-T non-tickFree-tick prems*(*1*) *tickFree-append-iff*)
qed

lemma *D*-$After_{trace}$:
‹*tF t* ⟹ *t* @ *u* ∈ $\mathcal{D}$ *P* ⟹ *u* ∈ $\mathcal{D}$ (*P* $after_{\mathcal{T}}$ *t*)›
proof (*induct t arbitrary*: *u rule*: *rev-induct*)
show ‹[] @ *u* ∈ $\mathcal{D}$ *P* ⟹ *u* ∈ $\mathcal{D}$ (*P* $after_{\mathcal{T}}$ [])› **for** *u* **by** *simp*
next
fix *e t u*
assume *hyp* : ‹*tF t* ⟹ *t* @ *u* ∈ $\mathcal{D}$ *P* ⟹ *u* ∈ $\mathcal{D}$ (*P* $after_{\mathcal{T}}$ *t*)› **for** *u*
assume *prems* : ‹*tF* ((*t* @ [*e*]))› ‹(*t* @ [*e*]) @ *u* ∈ $\mathcal{D}$ *P*›
have ∗ : ‹*e* # *u* ∈ $\mathcal{D}$ (*P* $after_{\mathcal{T}}$ *t*)› **by** (*rule hyp*; *use prems* **in** *simp*)
thus ‹*u* ∈ $\mathcal{D}$ (*P* $after_{\mathcal{T}}$ (*t* @ [*e*]))›
by (*simp add*: $After_{trace}$-*snoc D*-$After_{tick}$ *split*: $event_{ptick}$.*split*)
(*metis initials-memI D-T non-tickFree-tick prems*(*1*) *tickFree-append-iff*)
qed

lemma *T*-$After_{trace}$: ‹*t* @ *u* ∈ $\mathcal{T}$ *P* ⟹ *u* ∈ $\mathcal{T}$ (*P* $after_{\mathcal{T}}$ *t*)›

proof (*induct t arbitrary*: *u rule*: *rev-induct*)
show ‹[] @ *u* ∈ $\mathcal{T}$ *P* ⟹ *u* ∈ $\mathcal{T}$ (*P after*$_{\mathcal{T}}$ [])› **for** *u* **by** *simp*
next
fix *e t u*
assume *hyp* : ‹*t* @ *u* ∈ $\mathcal{T}$ *P* ⟹ *u* ∈ $\mathcal{T}$ (*P after*$_{\mathcal{T}}$ *t*)› **for** *u*
assume *prem* : ‹(*t* @ [*e*]) @ *u* ∈ $\mathcal{T}$ *P*›
have ∗ : ‹*e* # *u* ∈ $\mathcal{T}$ (*P after*$_{\mathcal{T}}$ *t*)› **by** (*rule hyp*, *use prem* **in** *simp*)
thus ‹*u* ∈ $\mathcal{T}$ (*P after*$_{\mathcal{T}}$ (*t* @ [*e*]))›
by (*simp add*: *After*$_{trace}$*-snoc T-After*$_{tick}$ *split*: *event*$_{ptick}$*.split*)
(*metis initials-memI append-T-imp-tickFree*
is-processT1-TR non-tickFree-tick prem tickFree-append-iff)
qed

corollary *initials-After*$_{trace}$:
‹*t* @ *e* # *u* ∈ $\mathcal{T}$ *P* ⟹ *e* ∈ (*P after*$_{\mathcal{T}}$ *t*)^0›
by (*metis initials-memI T-After*$_{trace}$)

corollary *F-imp-R-After*$_{trace}$: ‹*tF t* ⟹ (*t*, *X*) ∈ $\mathcal{F}$ *P* ⟹ *X* ∈ $\mathcal{R}$ (*P after*$_{\mathcal{T}}$ *t*)›
by (*simp add*: *F-After*$_{trace}$ *Refusals-iff*)

corollary *D-imp-After*$_{trace}$*-is-BOT*: ‹*tF t* ⟹ *t* ∈ $\mathcal{D}$ *P* ⟹ *P after*$_{\mathcal{T}}$ *t* = ⊥›
by (*simp add*: *BOT-iff-Nil-D D-After*$_{trace}$)

lemma *F-After*$_{trace}$*-eq*:
‹*t* ∈ $\mathcal{T}$ *P* ⟹ *tF t* ⟹ $\mathcal{F}$ (*P after*$_{\mathcal{T}}$ *t*) = {(*u*, *X*). (*t* @ *u*, *X*) ∈ $\mathcal{F}$ *P*}›
proof *safe*
show ‹*t* ∈ $\mathcal{T}$ *P* ⟹ *tF t* ⟹ (*u*, *X*) ∈ $\mathcal{F}$ (*P after*$_{\mathcal{T}}$ *t*) ⟹ (*t* @ *u*, *X*) ∈ $\mathcal{F}$ *P*›
for *u X*
proof (*induct t arbitrary*: *u rule*: *rev-induct*)
show ‹(*u*, *X*) ∈ $\mathcal{F}$ (*P after*$_{\mathcal{T}}$ []) ⟹ ([] @ *u*, *X*) ∈ $\mathcal{F}$ *P*› **for** *u* **by** *simp*
next
case (*snoc e t*)
from *initials-After*$_{trace}$ *snoc.prems*(*1*) **have** ∗ : ‹*e* ∈ (*P after*$_{\mathcal{T}}$ *t*)^0› **by** *blast*
obtain *a* **where** ‹*e* = *ev a*›
by (*metis event*$_{ptick}$*.exhaust non-tickFree-tick snoc.prems*(*2*) *tickFree-append-iff*)
show *?case*
apply (*simp*, *rule snoc.hyps*)
apply (*metis prefixI is-processT3-TR snoc.prems*(*1*))
apply (*use snoc.prems*(*2*) *tickFree-append-iff* **in** *blast*)
using *snoc.prems*(*3*) ∗ **by** (*simp add*: *After*$_{trace}$*-snoc F-After*$_{tick}$ ‹*e* = *ev a*›)
qed
next
show ‹*tF t* ⟹ (*t* @ *u*, *X*) ∈ $\mathcal{F}$ *P* ⟹ (*u*, *X*) ∈ $\mathcal{F}$ (*P after*$_{\mathcal{T}}$ *t*)› **for** *u X*
by (*fact F-After*$_{trace}$)
qed

lemma *D-After*$_{trace}$*-eq*:
‹$t \in \mathcal{T}\ P \Longrightarrow tF\ t \Longrightarrow \mathcal{D}\ (P\ after_{\mathcal{T}}\ t) = \{u.\ t\ @\ u \in \mathcal{D}\ P\}$›
proof *safe*
show ‹$t \in \mathcal{T}\ P \Longrightarrow tF\ t \Longrightarrow u \in \mathcal{D}\ (P\ after_{\mathcal{T}}\ t) \Longrightarrow t\ @\ u \in \mathcal{D}\ P$› **for** *u*
proof (*induct t arbitrary*: *u rule*: *rev-induct*)
show ‹$u \in \mathcal{D}\ (P\ after_{\mathcal{T}}\ []) \Longrightarrow []\ @\ u \in \mathcal{D}\ P$› **for** *u* **by** *simp*
next
case (*snoc e t*)
from *initials-After*$_{trace}$ *snoc.prems*(*1*) **have** ∗ : ‹$e \in (P\ after_{\mathcal{T}}\ t)^0$› **by** *blast*
obtain *a* **where** ‹$e = ev\ a$›
by (*metis event*$_{ptick}$*.exhaust non-tickFree-tick snoc.prems*(*2*) *tickFree-append-iff*)
show *?case*
apply (*simp*, *rule snoc.hyps*)
apply (*metis prefixI is-processT3-TR snoc.prems*(*1*))
using ∗ *After*$_{tick}$*-def After*$_{trace}$*-snoc D-After* ‹$e = ev\ a$› *snoc.prems*(*2*, *3*)
by *auto*
qed
next
show ‹$tF\ t \Longrightarrow t\ @\ u \in \mathcal{D}\ P \Longrightarrow u \in \mathcal{D}\ (P\ after_{\mathcal{T}}\ t)$› **for** *u* **by** (*fact D-After*$_{trace}$)
qed

lemma *T-After*$_{trace}$*-eq*:
‹$t \in \mathcal{T}\ P \Longrightarrow tF\ t \Longrightarrow \mathcal{T}\ (P\ after_{\mathcal{T}}\ t) = \{u.\ t\ @\ u \in \mathcal{T}\ P\}$›
proof *safe*
show ‹$t \in \mathcal{T}\ P \Longrightarrow tF\ t \Longrightarrow u \in \mathcal{T}\ (P\ after_{\mathcal{T}}\ t) \Longrightarrow t\ @\ u \in \mathcal{T}\ P$› **for** *u*
proof (*induct t arbitrary*: *u rule*: *rev-induct*)
show ‹$u \in \mathcal{T}\ (P\ after_{\mathcal{T}}\ []) \Longrightarrow []\ @\ u \in \mathcal{T}\ P$› **for** *u* **by** *simp*
next
case (*snoc e t*)
from *initials-After*$_{trace}$ *snoc.prems*(*1*) **have** ∗ : ‹$e \in (P\ after_{\mathcal{T}}\ t)^0$› **by** *blast*
obtain *a* **where** ‹$e = ev\ a$›
by (*metis event*$_{ptick}$*.exhaust non-tickFree-tick snoc.prems*(*2*) *tickFree-append-iff*)
show *?case*
apply (*simp*, *rule snoc.hyps*)
apply (*meson prefixI is-processT3-TR snoc.prems*(*1*))
using ∗ *After*$_{tick}$*-def After*$_{trace}$*-snoc T-After* ‹$e = ev\ a$› *snoc.prems*(*2*, *3*)
by *auto*
qed
next
show ‹$t\ @\ u \in \mathcal{T}\ P \Longrightarrow u \in \mathcal{T}\ (P\ after_{\mathcal{T}}\ t)$› **for** *u* **by** (*fact T-After*$_{trace}$)
qed

4.2.3 Monotony

4.2.4 Four inductive Constructions with $After_{trace}$

Reachable Processes

inductive-set *reachable-processes* :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ *set*› (‹$\mathcal{R}_{proc}$›)
for P :: ‹('a, 'r) $process_{ptick}$›
where *reachable-self* : ‹$P \in \mathcal{R}_{proc}\ P$›
| *reachable-after*: ‹$Q \in \mathcal{R}_{proc}\ P \Longrightarrow ev\ e \in Q^0 \Longrightarrow Q\ after\ e \in \mathcal{R}_{proc}\ P$›

lemma *reachable-processes-is*: ‹$\mathcal{R}_{proc}\ P = \{Q.\ \exists t \in \mathcal{T}\ P.\ tF\ t \wedge Q = P\ after_{\mathcal{T}}\ t\}$›
proof *safe*
show ‹$Q \in \mathcal{R}_{proc}\ P \Longrightarrow \exists t \in \mathcal{T}\ P.\ tF\ t \wedge Q = P\ after_{\mathcal{T}}\ t$› **for** Q
proof (*induct rule*: *reachable-processes.induct*)
show ‹$\exists t \in \mathcal{T}\ P.\ tF\ t \wedge P = P\ after_{\mathcal{T}}\ t$›
proof (*intro bexI*)
show ‹$tF\ [] \wedge P = P\ after_{\mathcal{T}}\ []$› **by** *simp*
next
show ‹$[] \in \mathcal{T}\ P$› **by** *simp*
qed
next
fix Q e
assume *assms* : ‹$Q \in \mathcal{R}_{proc}\ P$› ‹$\exists t \in \mathcal{T}\ P.\ tF\ t \wedge Q = P\ after_{\mathcal{T}}\ t$› ‹$ev\ e \in Q^0$›
from *assms*(*2*) **obtain** t **where** $*$: ‹$tF\ t$› ‹$t \in \mathcal{T}\ P$› ‹$Q = P\ after_{\mathcal{T}}\ t$› **by** *blast*
show ‹$\exists t{\in}\mathcal{T}\ P.\ tF\ t \wedge Q\ after\ e = P\ after_{\mathcal{T}}\ t$›
proof (*intro bexI*)
show ‹$tF\ (t\ @\ [ev\ e]) \wedge Q\ after\ e = P\ after_{\mathcal{T}}\ (t\ @\ [ev\ e])$›
by (*simp add*: $After_{trace}$-*snoc* $After_{tick}$-*def* $*$(*1*, *3*))
next
from $*$ *T-*$After_{trace}$*-eq assms*(*3*) *initials-def*
show ‹$t\ @\ [ev\ e] \in \mathcal{T}\ P$› **by** *fastforce*
qed
qed
next
show ‹$t \in \mathcal{T}\ P \Longrightarrow tF\ t \Longrightarrow P\ after_{\mathcal{T}}\ t \in \mathcal{R}_{proc}\ P$› **for** t
proof (*induct t rule*: *rev-induct*)
show ‹$P\ after_{\mathcal{T}}\ [] \in \mathcal{R}_{proc}\ P$› **by** (*simp add*: *reachable-self*)
next
fix t e
assume *hyp* : ‹$t \in \mathcal{T}\ P \Longrightarrow tF\ t \Longrightarrow P\ after_{\mathcal{T}}\ t \in \mathcal{R}_{proc}\ P$›
and *prems* : ‹$t\ @\ [e] \in \mathcal{T}\ P$› ‹$tF\ (t\ @\ [e])$›
from *prems T-F-spec is-processT3 tickFree-append-iff*
have $*$: ‹$t \in \mathcal{T}\ P$› ‹$tF\ t$› **by** *blast*+
obtain a **where** ‹$e = ev\ a$›
by (*metis* $event_{ptick}$*.exhaust non-tickFree-tick prems*(*2*) *tickFree-append-iff*)

thus ‹P *after*$_{\mathcal{T}}$ (t @ [e]) ∈ $\mathcal{R}_{proc}$ P›
by (*simp add*: *After*$_{trace}$*-snoc After*$_{tick}$*-def*)
(*use initials-After*$_{trace}$ *prems*(*1*) **in**
‹*blast intro*: *initials-After*$_{trace}$ *prems*(*1*) *reachable-after*[*OF hyp*[*OF* ∗]]›)
qed
qed

lemma *reachable-processes-trans*: ‹$Q \in \mathcal{R}_{proc}\ P \Longrightarrow R \in \mathcal{R}_{proc}\ Q \Longrightarrow R \in \mathcal{R}_{proc}\ P$›
apply (*simp add*: *reachable-processes-is*, *elim bexE*)
apply (*simp add*: *After*$_{trace}$*-append*[*symmetric*] *T-After*$_{trace}$*-eq*)
using *tickFree-append-iff* **by** *blast*

Antecedent Processes

inductive-set *antecedent-processes* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ ('*a*, '*r*) *process*$_{ptick}$ *set*› (‹$\mathcal{A}_{proc}$›)
for P :: ‹('*a*, '*r*) *process*$_{ptick}$›
where *antecedent-self* : ‹$P \in \mathcal{A}_{proc}\ P$›
| *antecedent-after*: ‹Q *after* $e \in \mathcal{A}_{proc}\ P \Longrightarrow ev\ e \in Q^0 \Longrightarrow Q \in \mathcal{A}_{proc}\ P$›

lemma *antecedent-processes-is*: ‹$\mathcal{A}_{proc}\ P = \{Q.\ \exists t \in \mathcal{T}\ Q.\ tF\ t \wedge P = Q\ after_{\mathcal{T}}\ t\}$›
proof *safe*
have ‹$Q \in \mathcal{A}_{proc}\ P \Longrightarrow P \in \mathcal{R}_{proc}\ Q$› **for** Q
proof (*induct rule*: *antecedent-processes.induct*)
show ‹$P \in \mathcal{R}_{proc}\ P$› **by** (*fact reachable-self*)
next
show ‹Q *after* $e \in \mathcal{A}_{proc}\ P \Longrightarrow P \in \mathcal{R}_{proc}\ (Q\ after\ e) \Longrightarrow ev\ e \in Q^0 \Longrightarrow P \in \mathcal{R}_{proc}\ Q$› **for** Q e
by (*meson reachable-after reachable-self reachable-processes-trans*)
qed
thus ‹$Q \in \mathcal{A}_{proc}\ P \Longrightarrow \exists t \in \mathcal{T}\ Q.\ tF\ t \wedge P = Q\ after_{\mathcal{T}}\ t$› **for** Q
by (*simp add*: *reachable-processes-is*)
next
show ‹$t \in \mathcal{T}\ Q \Longrightarrow tF\ t \Longrightarrow Q \in \mathcal{A}_{proc}\ (Q\ after_{\mathcal{T}}\ t)$› **for** t Q
proof (*induct t arbitrary*: *Q*)
show ‹$Q \in \mathcal{A}_{proc}\ (Q\ after_{\mathcal{T}}\ [])$› **for** Q **by** (*simp add*: *antecedent-self*)
next
fix *e t Q*
assume *hyp* : ‹$\bigwedge Q.\ t \in \mathcal{T}\ Q \Longrightarrow tF\ t \Longrightarrow Q \in \mathcal{A}_{proc}\ (Q\ after_{\mathcal{T}}\ t)$›
and *prems* : ‹$e\ \#\ t \in \mathcal{T}\ Q$› ‹$tF\ (e\ \#\ t)$›
from *prems* **obtain** *a* **where** ‹$e = ev\ a$› ‹$ev\ a \in Q^0$›
by (*metis initials-memI is-ev-def tickFree-Cons-iff*)
with *prems* **have** ‹$t \in \mathcal{T}\ (Q\ after\ a)$› ‹$tF\ t$› **by** (*simp-all add*: *T-After*)
from *hyp*[*OF this*] **have** ‹$Q\ after\ a \in \mathcal{A}_{proc}\ (Q\ after_{\mathcal{T}}\ (e\ \#\ t))$›
by (*simp add*: *After*$_{tick}$*-def* ‹$e = ev\ a$›)
from *this* ‹$ev\ a \in Q^0$› **show** ‹$Q \in \mathcal{A}_{proc}\ (Q\ after_{\mathcal{T}}\ (e\ \#\ t))$›

by (*fact antecedent-after*)
qed
qed

lemma *antecedent-processes-trans*: ‹$Q \in \mathcal{A}_{proc}\ P \Longrightarrow R \in \mathcal{A}_{proc}\ Q \Longrightarrow R \in \mathcal{A}_{proc}\ P$›
apply (*simp add*: *antecedent-processes-is*, *elim bexE*)
apply (*simp add*: $After_{trace}$-*append*[*symmetric*] *T-*$After_{trace}$-*eq*)
using *tickFree-append-iff* **by** *blast*

corollary *antecedent-processes-iff-rev-reachable-processes*: ‹$P \in \mathcal{A}_{proc}\ Q \longleftrightarrow Q \in \mathcal{R}_{proc}\ P$›
by (*simp add*: *antecedent-processes-is reachable-processes-is*)

4.2.5 Nth initials Events

primrec *nth-initials* :: ‹$('a, 'r)\ process_{ptick} \Rightarrow nat \Rightarrow ('a, 'r)\ event_{ptick}\ set$› (‹-› [*1000*, *3*] *999*)
where ‹$P^0 = P^0$›
| ‹$P^{Suc\ n} = \bigcup \{(P\ after\ e)^n \mid e.\ ev\ e \in P^0\}$›

lemma ‹$P^0 = P^0$› **by** *simp*

lemma *first-initials-is* : ‹$P^1 = \bigcup \{(P\ after\ e)^0 \mid e.\ ev\ e \in P^0\}$› **by** *simp*

lemma *second-initials-is* :
‹$P^2 = \bigcup \{(P\ after\ e\ after\ f)^0 \mid e\ f.\ ev\ e \in P^0 \wedge ev\ f \in (P\ after\ e)^0\}$›
by (*auto simp add*: *numeral-eq-Suc*)

lemma *third-initials-is* :
‹$P^3 = \bigcup \{(P\ after\ e\ after\ f\ after\ g)^0 \mid e\ f\ g.$
$ev\ e \in P^0 \wedge ev\ f \in (P\ after\ e)^0 \wedge ev\ g \in (P\ after\ e\ after\ f)^0\}$›
by (*auto simp add*: *numeral-eq-Suc*)

More generally, we have the following result.

lemma *nth-initials-is*: ‹$P^n = \bigcup \{(P\ after_{\mathcal{T}}\ t)^0 \mid t.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge length\ t = n\}$›
proof (*induct n arbitrary*: *P*)
show ‹$P^0 = \bigcup \{(P\ after_{\mathcal{T}}\ t)^0 \mid t.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge length\ t = 0\}$› **for** *P* **by** *simp*
next
fix *n P*
assume *hyp* : ‹$P^n = \bigcup \{(P\ after_{\mathcal{T}}\ t)^0 \mid t.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge length\ t = n\}$› **for** *P*
show ‹$P^{Suc\ n} = \bigcup \{(P\ after_{\mathcal{T}}\ t)^0 \mid t.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge length\ t = Suc\ n\}$›
proof *safe*
show ‹$e \in P^{Suc\ n} \Longrightarrow e \in \bigcup \{(P\ after_{\mathcal{T}}\ t)^0 \mid t.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge length\ t =$

Suc n}› **for** *e*
by (*auto simp add*: *hyp T-After*)
(*metis* (*no-types*, *lifting*) *AfterExt.After*$_{tick}$*-def AfterExt.After*$_{trace}$*.simps*(*2*)
event$_{ptick}$*.simps*(*5*) *is-ev-def length-Cons tickFree-Cons-iff*)
next
fix *e t*
assume *assms* : ‹*e* ∈ (*P after*$_{\mathcal{T}}$ *t*)^0› ‹*t* ∈ $\mathcal{T}$ *P*› ‹*tF t*› ‹*length t* = *Suc n*›
from *assms*(*1*−*3*) **have** ∗ : ‹*t* @ [*e*] ∈ $\mathcal{T}$ *P*›
unfolding *initials-def* **by** (*simp add*: *T-After*$_{trace}$*-eq*)
from *assms*(*3*, *4*)**obtain** *a t*′ **where** ∗∗ : ‹*t* = *ev a* # *t*′› ‹*tF t*′› ‹*length t*′ = *n*›
by (*metis Suc-length-conv event*$_{ptick}$*.collapse*(*1*) *tickFree-Cons-iff*)
with *initials-memI assms*(*2*) **have** ‹*ev a* ∈ P^0› **by** *blast*
show ‹*e* ∈ $P^{Suc\ n}$›
proof (*unfold nth-initials.simps*(*2*), *rule UnionI*)
from ‹*ev a* ∈ P^0› **show** ‹(*P after a*)^n ∈ {(*P after e*)^n |*e*. *ev e* ∈ P^0}› **by** *blast*
next
from *assms*(*1*, *2*) ∗∗(*1*−*3*) ‹*ev a* ∈ P^0›
show ‹*e* ∈ (*P after a*)^n› **by** (*auto simp add*: *hyp* ∗∗(*1*) *After*$_{tick}$*-def T-After*)
qed
qed
qed

lemma *nth-initials-DF*: ‹(*DF A*)^n = *ev* ‘ *A*›
by (*induct n*) (*auto simp add*: *initials-DF After-DF*)

lemma *nth-initials-DF*$_{SKIPS}$:
‹(*DF*$_{SKIPS}$ *A R*)^n = (*if A* = {} *then if n* = *0 then tick* ‘ *R else* {} *else ev* ‘ *A* ∪ *tick* ‘ *R*)›
by (*induct n*) (*auto simp add*: *initials-DF*$_{SKIPS}$ *After-DF*$_{SKIPS}$)

lemma *nth-initials-RUN*: ‹(*RUN A*)^n = *ev* ‘ *A*›
by (*induct n*) (*auto simp add*: *initials-RUN After-RUN*)

lemma *nth-initials-CHAOS*: ‹(*CHAOS A*)^n = *ev* ‘ *A*›
by (*induct n*) (*auto simp add*: *initials-CHAOS After-CHAOS*)

lemma *nth-initials-CHAOS*$_{SKIPS}$:
‹(*CHAOS*$_{SKIPS}$ *A R*)^n = (*if A* = {} *then if n* = *0 then tick* ‘ *R else* {} *else ev* ‘ *A* ∪ *tick* ‘ *R*)›
by (*induct n*) (*auto simp add*: *initials-CHAOS*$_{SKIPS}$ *After-CHAOS*$_{SKIPS}$)

Reachable Ev

inductive *reachable-ev* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ '*a* ⇒ *bool*›
where *initial-ev-reachable*:
‹*ev a* ∈ P^0 ⟹ *reachable-ev P a*›
| *reachable-ev-after-reachable*:

```
    ‹ev b ∈ P⁰ ⟹ reachable-ev (P after b) a ⟹ reachable-ev P a›

definition reachable-ev-set :: ‹('a, 'r) process_ptick ⇒ 'a set› (‹ℛ_ev›)
  where ‹ℛ_ev P ≡ ⋃ Q ∈ ℛ_proc P. {a. ev a ∈ Q⁰}›


lemma reachable-ev-BOT : ‹reachable-ev ⊥ a›
  and not-reachable-ev-STOP : ‹¬ reachable-ev STOP a›
  and not-reachable-ev-SKIP : ‹¬ reachable-ev (SKIP r) a›
  by (subst reachable-ev.simps, simp)+


lemma events-of-iff-reachable-ev: ‹a ∈ α(P) ⟷ reachable-ev P a›
proof (intro iffI)
  show ‹reachable-ev P a ⟹ a ∈ α(P)›
    apply (induct rule: reachable-ev.induct;
        simp add: T-After events-of-def initials-def split: if-split-asm)
    by (meson list.set-intros(1)) (meson list.set-intros(2))
next
  assume ‹a ∈ α(P)›
  then obtain t where ∗ : ‹t ∈ 𝒯 P› ‹ev a ∈ set t› by (meson events-of-memD)
  thus ‹reachable-ev P a›
  proof (induct t arbitrary: P)
    show ‹⋀P a. ev a ∈ set [] ⟹ reachable-ev P a› by simp
  next
    case (Cons e t)
    from Cons.prems(1) initials-memI
    have ∗ : ‹e ∈ P⁰› unfolding initials-def by blast
    from Cons.prems(2) consider ‹e = ev a› | ‹ev a ∈ set t› by auto
    thus ‹reachable-ev P a›
    proof cases
      assume ‹e = ev a›
      from ∗[simplified this]
      show ‹reachable-ev P a› by (fact reachable-ev.intros(1))
    next
      show ‹reachable-ev P a› if ‹ev a ∈ set t›
      proof (cases e)
        fix b
        assume ‹e = ev b›
       with ∗ Cons.prems(1) have ‹t ∈ 𝒯 (P after b)› by (force simp add: T-After)
        show ‹reachable-ev P a›
        proof (rule reachable-ev.intros(2))
          from ∗ ‹e = ev b› show ‹ev b ∈ P⁰› by blast
        next
          show ‹reachable-ev (P after b) a›
            by (fact Cons.hyps[OF ‹t ∈ 𝒯 (P after b)› ‹ev a ∈ set t›])
        qed
      next
        from Cons.prems(1) have ‹e = ✓(r) ⟹ t = []› for r
```

```
      by simp (meson T-imp-front-tickFree event_ptick.disc(2) front-tickFree-Cons-iff)
       with ‹ev a ∈ set t› show ‹e = ✓(r) ⟹ reachable-ev P a› for r by simp
     qed
    qed
  qed
qed


lemma reachable-ev-iff-in-initials-After_trace-for-some-tickFree-T:
  ‹reachable-ev P a ⟷ (∃ t ∈ 𝒯 P. tF t ∧ ev a ∈ (P after_𝒯 t)⁰)›
proof (intro iffI)
  show ‹reachable-ev P a ⟹ ∃ t∈𝒯 P. tF t ∧ ev a ∈ (P after_𝒯 t)⁰›
  proof (induct rule: reachable-ev.induct)
    show ‹ev a ∈ P⁰ ⟹ ∃ t∈𝒯 P. tF t ∧ ev a ∈ (P after_𝒯 t)⁰› for a P
    proof (intro bexI)
      show ‹ev a ∈ P⁰ ⟹ tF [] ∧ ev a ∈ (P after_𝒯 [])⁰› by simp
    next
      show ‹[] ∈ 𝒯 P› by (fact Nil-elem-T)
    qed
  next
    fix b a P
    assume prems : ‹ev b ∈ P⁰› ‹reachable-ev (P after b) a›
      and   hyp : ‹∃ t ∈ 𝒯 (P after b). tF t ∧ ev a ∈ (P after b after_𝒯 t)⁰›
    from hyp obtain t
      where ∗ : ‹tF t› ‹t ∈ 𝒯 (P after b)›
        ‹ev a ∈ (P after b after_𝒯 t)⁰› by blast
    show ‹∃ t ∈ 𝒯 P. tF t ∧ ev a ∈ (P after_𝒯 t)⁰›
    proof (intro bexI)
      show ‹tF (ev b # t) ∧ ev a ∈ (P after_𝒯 (ev b # t))⁰›
        by (simp add: ∗(1, 3) After_tick-def)
    next
      from ∗(2) show ‹ev b # t ∈ 𝒯 P› by (simp add: T-After prems(1))
    qed
  qed
next
  show ‹∃ t ∈ 𝒯 P. tF t ∧ ev a ∈ (P after_𝒯 t)⁰ ⟹ reachable-ev P a›
  proof (elim bexE conjE)
    show ‹t ∈ 𝒯 P ⟹ tF t ⟹ ev a ∈ (P after_𝒯 t)⁰ ⟹ reachable-ev P a› for t
    proof (induct t arbitrary: P)
      show ‹ev a ∈ (P after_𝒯 [])⁰ ⟹ reachable-ev P a› for P
        by (simp add: reachable-ev.intros(1))
    next
      fix e t P
      assume   hyp : ‹t ∈ 𝒯 P ⟹ tF t ⟹ ev a ∈ (P after_𝒯 t)⁰ ⟹
                      reachable-ev P a› for P
      assume prems : ‹tF (e # t)› ‹e # t ∈ 𝒯 P› ‹ev a ∈ (P after_𝒯 (e # t))⁰›
      from prems(1) obtain b where ‹e = ev b› by (cases e; simp)
      with prems(2) have initial : ‹ev b ∈ P⁰› by (simp add: initials-memI)
      from reachable-ev-after-reachable[OF initial[simplified ‹e = ev b›]]
```

After$_{tick}$*-def T-After* ‹$e = ev\ b$› *hyp prems initial*
show ‹*reachable-ev P a*› **by** *auto*
qed
qed
qed

Properties

corollary *reachable-ev-set-is-mem-Collect-reachable-ev*:
‹$\mathcal{R}_{ev}\ P = \{a.\ reachable\text{-}ev\ P\ a\}$›
by (*auto simp add*: *reachable-ev-set-def reachable-processes-is*
reachable-ev-iff-in-initials-After$_{trace}$*-for-some-tickFree-T*)

corollary *events-of-is-reachable-ev-set*: ‹$\alpha(P) = \mathcal{R}_{ev}\ P$›
by (*simp add*: *reachable-ev-set-is-mem-Collect-reachable-ev*
set-eq-iff events-of-iff-reachable-ev)

lemma *events-of-reachable-processes-subset*: ‹$Q \in \mathcal{R}_{proc}\ P \Longrightarrow \alpha(Q) \subseteq \alpha(P)$›
by (*induct rule*: *reachable-processes.induct*)
(*simp-all add*: *subset-iff events-of-iff-reachable-ev reachable-ev-after-reachable*)

corollary *events-of-antecedent-processes-superset*: ‹$Q \in \mathcal{A}_{proc}\ P \Longrightarrow \alpha(P) \subseteq \alpha(Q)$›
by (*simp add*: *antecedent-processes-iff-rev-reachable-processes*
events-of-reachable-processes-subset)

lemma *events-of-is-Union-nth-initials*: ‹$\alpha(P) = (\bigcup n.\ \{a.\ ev\ a \in P^n\})$›
by (*auto simp add*: *nth-initials-is events-of-iff-reachable-ev*
reachable-ev-iff-in-initials-After$_{trace}$*-for-some-tickFree-T*)+

Reachable Tick

inductive *reachable-tick* :: ‹$('a, 'r)\ process_{ptick} \Rightarrow 'r \Rightarrow bool$›
where *initial-tick-reachable*:
‹$\checkmark(r) \in P^0 \Longrightarrow reachable\text{-}tick\ P\ r$›
| *reachable-tick-after-reachable*:
‹$ev\ a \in P^0 \Longrightarrow reachable\text{-}tick\ (P\ after\ a)\ r \Longrightarrow reachable\text{-}tick\ P\ r$›

definition *reachable-tick-set* :: ‹$('a, 'r)\ process_{ptick} \Rightarrow 'r\ set$› (‹$\mathcal{R}_{\checkmark}$›)
where ‹$\mathcal{R}_{\checkmark}\ P \equiv \bigcup Q \in \mathcal{R}_{proc}\ P.\ \{r.\ \checkmark(r) \in Q^0\}$›

lemma *reachable-tick-BOT* : ‹$reachable\text{-}tick\ \bot\ r$›
and *not-reachable-tick-STOP* : ‹$\neg\ reachable\text{-}tick\ STOP\ s$›
and *reachable-tick-SKIP-iff* : ‹$reachable\text{-}tick\ (SKIP\ r)\ s \longleftrightarrow s = r$›
by (*subst reachable-tick.simps*, *simp*)+

lemma *ticks-of-iff-reachable-tick* : ‹$r \in \checkmark s(P) \longleftrightarrow reachable\text{-}tick\ P\ r$›
proof (*intro iffI*)

```
  show ‹reachable-tick P r ⟹ r ∈ ✓s(P)›
    apply (induct rule: reachable-tick.induct;
        simp add: T-After ticks-of-def initials-def split: if-split-asm)
    by (metis append-Nil, metis append-Cons)
next
  assume ‹r ∈ ✓s(P)›
  then obtain t where ∗ : ‹t @ [✓(r)] ∈ 𝒯 P› by (meson ticks-of-memD)
  thus ‹reachable-tick P r›
  proof (induct t arbitrary: P)
    show ‹[] @ [✓(r)] ∈ 𝒯 P ⟹ reachable-tick P r› for P
      by (simp add: initial-tick-reachable initials-memI)
  next
    case (Cons e t)
    obtain a where ‹e = ev a›
      by (meson Cons.prems append-T-imp-tickFree is-ev-def not-Cons-self2 tick-
Free-Cons-iff)
    moreover from Cons.prems have ‹e ∈ P⁰› by (auto intro: initials-memI)
    ultimately have ‹t @ [✓(r)] ∈ 𝒯 (P after a)›
      using Cons.prems by (simp add: T-After ‹e = ev a›)
    with Cons.hyps have ‹reachable-tick (P after a) r› by blast
    with ‹e = ev a› ‹e ∈ P⁰› reachable-tick-after-reachable
    show ‹reachable-tick P r› by blast
  qed
qed


lemma reachable-tick-iff-in-initials-After_trace-for-some-tickFree-T:
  ‹reachable-tick P r ⟷ (∃ t ∈ 𝒯 P. tF t ∧ ✓(r) ∈ (P after_𝒯 t)⁰)›
proof (intro iffI)
  show ‹reachable-tick P r ⟹ ∃ t∈𝒯 P. tF t ∧ ✓(r) ∈ (P after_𝒯 t)⁰›
  proof (induct rule: reachable-tick.induct)
    show ‹✓(r) ∈ P⁰ ⟹ ∃ t∈𝒯 P. tF t ∧ ✓(r) ∈ (P after_𝒯 t)⁰› for r P
    proof (intro bexI)
      show ‹✓(r) ∈ P⁰ ⟹ tF [] ∧ ✓(r) ∈ (P after_𝒯 [])⁰› by simp
    next
      show ‹[] ∈ 𝒯 P› by (fact Nil-elem-T)
    qed
  next
    fix a P r
    assume prems : ‹ev a ∈ P⁰› ‹reachable-tick (P after a) r›
      and   hyp : ‹∃ t∈𝒯 (P after a). tF t ∧ ✓(r) ∈ (P after a after_𝒯 t)⁰›
    from hyp obtain t
      where ∗ : ‹tF t› ‹t ∈ 𝒯 (P after a)›
        ‹✓(r) ∈ (P after a after_𝒯 t)⁰› by blast
    show ‹∃ t∈𝒯 P. tF t ∧ ✓(r) ∈ (P after_𝒯 t)⁰›
    proof (intro bexI)
      show ‹tF (ev a # t) ∧ ✓(r) ∈ (P after_𝒯 (ev a # t))⁰›
        by (simp add: ∗(1, 3) After_tick-def)
    next
```

from $*(2)$ **show** ‹*ev a # t ∈ 𝒯 P*› **by** (*simp add*: *T-After prems*(*1*))
qed
qed
next
show ‹ ∃ *t*∈𝒯 *P*. *tF t* ∧ ✓(*r*) ∈ (*P after*$_{\mathcal{T}}$ *t*)^0 ⟹ *reachable-tick P r*›
proof (*elim bexE conjE*)
show ‹*t* ∈ 𝒯 *P* ⟹ *tF t* ⟹ ✓(*r*) ∈ (*P after*$_{\mathcal{T}}$ *t*)^0 ⟹ *reachable-tick P r*› **for** *t*
proof (*induct t arbitrary*: *P*)
show ‹✓(*r*) ∈ (*P after*$_{\mathcal{T}}$ []^0 ⟹ *reachable-tick P r*› **for** *P*
by (*simp add*: *initial-tick-reachable*)
next
fix *e t P*
assume *hyp* : ‹*t* ∈ 𝒯 *P* ⟹ *tF t* ⟹ ✓(*r*) ∈ (*P after*$_{\mathcal{T}}$ *t*)^0 ⟹ *reachable-tick P r*› **for** *P*
assume *prems* : ‹*tF* (*e # t*)› ‹*e # t* ∈ 𝒯 *P*› ‹✓(*r*) ∈ (*P after*$_{\mathcal{T}}$ (*e # t*))^0›
from *prems*(*1*) **obtain** *a* **where** ‹*e* = *ev a*› **by** (*cases e*; *simp*)
with *prems*(*2*) **have** *initial* : ‹*ev a* ∈ *P*^{0}› **by** (*simp add*: *initials-memI*)
from *reachable-tick-after-reachable*[*OF initial*[*simplified* ‹*e* = *ev a*›]]
After$_{tick}$*-def T-After* ‹*e* = *ev a*› *hyp prems initial*
show ‹*reachable-tick P r*› **by** *auto*
qed
qed
qed

Properties

corollary *reachable-tick-set-is-mem-Collect-reachable-tick* :
‹$\mathcal{R}_{\checkmark}$ *P* = {*a*. *reachable-tick P a*}›
by (*auto simp add*: *reachable-tick-set-def reachable-processes-is*
reachable-tick-iff-in-initials-After$_{trace}$*-for-some-tickFree-T*)

corollary *ticks-of-is-reachable-tick-set* : ‹✓*s*(*P*) = $\mathcal{R}_{\checkmark}$ *P*›
by (*simp add*: *reachable-tick-set-is-mem-Collect-reachable-tick*
set-eq-iff ticks-of-iff-reachable-tick)

lemma *ticks-of-reachable-processes-subset* : ‹*Q* ∈ $\mathcal{R}_{proc}$ *P* ⟹ ✓*s*(*Q*) ⊆ ✓*s*(*P*)›
by (*induct rule*: *reachable-processes.induct*)
(*simp-all add*: *subset-iff ticks-of-iff-reachable-tick reachable-tick-after-reachable*)

corollary *ticks-of-antecedent-processes-superset* : ‹*Q* ∈ $\mathcal{A}_{proc}$ *P* ⟹ ✓*s*(*P*) ⊆ ✓*s*(*Q*)›
by (*simp add*: *antecedent-processes-iff-rev-reachable-processes*
ticks-of-reachable-processes-subset)

lemma *ticks-of-is-Union-nth-initials*: ‹✓*s*(*P*) = (⋃ *n*. {*r*. ✓(*r*) ∈ *P*^{n}})›
by (*auto simp add*: *nth-initials-is ticks-of-iff-reachable-tick*
reachable-tick-iff-in-initials-After$_{trace}$*-for-some-tickFree-T*)+

4.2.6 Characterizations for Deadlock Freeness

Remember that we have characterized *deadlock-free P* with an equality involving ($after_{\checkmark}$): *deadlock-free P* = (*range ev* $\notin \mathcal{R}$ *P* $\wedge$ ($\forall$ *e. ev e* $\in P^0$ $\longrightarrow$ *deadlock-free* (*P after*$_{\checkmark}$ *ev e*))). This can of course be derived in a characterization involving ($after_{\mathcal{T}}$).

lemma *deadlock-free-After*$_{trace}$*-characterization*:
‹*deadlock-free P* $\longleftrightarrow$ ($\forall$ *t* $\in \mathcal{T}$ *P. range ev* $\notin \mathcal{R}_a$ *P t* $\wedge$ (*t* $\neq$ [] $\longrightarrow$ *deadlock-free* (*P after*$_{\mathcal{T}}$ *t*)))›
proof (*intro iffI*)
show ‹*deadlock-free P* $\Longrightarrow$ $\forall$ *t*$\in\mathcal{T}$ *P. range ev* $\notin \mathcal{R}_a$ *P t* $\wedge$ (*t* $\neq$ [] $\longrightarrow$ *deadlock-free* (*P after*$_{\mathcal{T}}$ *t*))›
proof (*intro ballI*)
show ‹*deadlock-free P* $\Longrightarrow$ *t* $\in \mathcal{T}$ *P* $\Longrightarrow$ *range ev* $\notin \mathcal{R}_a$ *P t* $\wedge$ (*t* $\neq$ [] $\longrightarrow$ *deadlock-free* (*P after*$_{\mathcal{T}}$ *t*))› **for** *t*
proof (*induct t rule*: *rev-induct*)
case *Nil*
thus *?case*
by (*subst* (*asm*) *deadlock-free-After*$_{tick}$*-characterization*)
(*simp add*: *Refusals-after-def Nil.prems*(*1*) *Refusals-iff*)
next
case (*snoc e t*)
thus *?case*
by (*simp add*: *Refusals-after-def After*$_{trace}$*-snoc*)
(*metis After*$_{trace}$*.simps*(*1*) *After*$_{trace}$*-snoc DF-FD-After*$_{tick}$ *F-imp-R-After*$_{trace}$ *T-F-spec*
deadlock-free-After$_{tick}$*-characterization deadlock-free-def*
deadlock-free-implies-non-terminating initials-After$_{trace}$ *is-processT3*)
qed
qed
show ‹$\forall$ *t*$\in\mathcal{T}$ *P. range ev* $\notin \mathcal{R}_a$ *P t* $\wedge$ (*t* $\neq$ [] $\longrightarrow$ *deadlock-free* (*P after*$_{\mathcal{T}}$ *t*)) $\Longrightarrow$ *deadlock-free P*›
by (*subst deadlock-free-After*$_{tick}$*-characterization*)
(*metis After*$_{trace}$*.simps Nil-elem-T Refusals-after-def*
Refusals-iff list.distinct(*1*) *mem-Collect-eq initials-def*)
qed

lemma *deadlock-free*$_{SKIPS}$*-After*$_{trace}$*-characterization*:
‹*deadlock-free*$_{SKIPS}$ *P* $\longleftrightarrow$
($\forall$ *t* $\in \mathcal{T}$ *P. tF t* $\longrightarrow$ *UNIV* $\notin \mathcal{R}_a$ *P t* $\wedge$ (*t* $\neq$ [] $\longrightarrow$ *deadlock-free*$_{SKIPS}$ (*P after*$_{\mathcal{T}}$ *t*)))›
by (*auto simp add*: *deadlock-free*$_{SKIPS}$*-is-right F-After*$_{trace}$*-eq T-After*$_{trace}$*-eq Refusals-after-def*)

Actually, with $After_{trace}$, we can obtain much more powerful results. This will be developed later.

4.2.7 Continuity

lemma $After_{trace}$-*cont* :
‹⟦∀ *a*. *ev a* ∈ *set t* ⟶ *cont* (λ*P*. Ψ *P a*);
∀ *r*. ✓(*r*) ∈ *set t* ⟶ *cont* (λ*P*. Ω *P r*); *cont f*⟧ ⟹
cont (λ*x*. *f x after*$_{\mathcal{T}}$ *t*)›
proof (*induct t arbitrary*: *f*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons e s*)
show *?case* **by** (*cases e*; *simp*, *rule Cons.hyps*) (*simp-all add*: *Cons.prems*)
qed

end

Chapter 5

Motivations for our Definitions

To construct our bridge between denotational and operational semantics, we want to define two kind of transitions:

- without external event: $P \rightsquigarrow_\tau P'$
- with the terminating event ✓(r): $P \rightsquigarrow_{\checkmark r} P'$
- with a non terminating external event *ev e*: $P \rightsquigarrow_e P'$.

  We will discuss in this theory some fundamental properties that we want

  $P \rightsquigarrow_\tau Q$, $P \rightsquigarrow_e P'$ and $P \rightsquigarrow_{\checkmark r} P'$ to verify, and the consequences that this will have.

Let's say we want to define the τ transition as an inductive predicate with three introduction rules:

- we allow a process to make a τ transition towards itself: $P \rightsquigarrow_\tau P$
- the non-deterministic choice ($\sqcap$) can make a τ transition to the left side $P \sqcap Q \rightsquigarrow_\tau P$
- the non-deterministic choice ($\sqcap$) can make a τ transition to the right side $P \sqcap Q \rightsquigarrow_\tau Q$.

inductive *τ-trans* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ ('*a*, '*r*) *process*$_{ptick}$ ⇒ *bool*› (**infixl** ‹⇝$_\tau$› *50*)
  where *τ-trans-eq* : ‹*P* ⇝$_\tau$ *P*›
  | *τ-trans-NdetL* : ‹*P* ⊓ *Q* ⇝$_\tau$ *P*›

| τ-trans-NdetR : $\langle P \sqcap Q \rightsquigarrow_\tau Q\rangle$

— We can obtain the same inductive predicate by removing τ-trans-eq and τ-trans-NdetR clauses (because of ($\sqcap$) properties).

With this definition, we immediately show that the τ transition is the FD-refinement ($\sqsubseteq_{FD}$).

lemma τ-trans-is-FD: $\langle(\rightsquigarrow_\tau) = (\sqsubseteq_{FD})\rangle$
apply (*intro ext iffI*)
by (*metis Ndet-FD-self-left Ndet-FD-self-right* τ*-trans.simps order-class.order-eq-iff*)
(*metis FD-antisym Ndet-FD-self-left Ndet-id* τ*-trans-NdetR mono-Ndet-FD*)

The definition of the event transition will be a little bit more complex.

First of all we want to prevent a process P::$('a, 'r)$ $process_{ptick}$ to make a transition with ev (e::$'a$) (resp. ✓(r::$'r$)) if P can not begin with ev e (resp. ✓(r)).

More formally, we want $P \rightsquigarrow_e P' \Longrightarrow ev\ e \in P^0$ (resp. $P \rightsquigarrow_{\checkmark r} P' \Longrightarrow \checkmark(r) \in P^0$).

Moreover, we want the event transitions to absorb the τ transitions.

Finally, when $e \in P^0$ (resp. ✓(r) $\in P^0$), we want to have $P \rightsquigarrow_e P$ $after_\checkmark$ ev e (resp. $P \rightsquigarrow_{\checkmark r} P$ $after_\checkmark$ ✓(r)).

This brings us to the following inductive definition:

inductive *event-trans-prem* :: $\langle('a, 'r)\ process_{ptick} \Rightarrow ('a, 'r)\ event_{ptick} \Rightarrow ('a, 'r)\ process_{ptick} \Rightarrow bool\rangle$
where
τ-left-absorb : $\langle[\![e \in initials\ P'; P \rightsquigarrow_\tau P'; event\text{-}trans\text{-}prem\ P'\ e\ P'']\!] \Longrightarrow event\text{-}trans\text{-}prem\ P\ e\ P''\rangle$
| τ-right-absorb : $\langle[\![e \in initials\ P; event\text{-}trans\text{-}prem\ P\ e\ P'; P' \rightsquigarrow_\tau P'']\!] \Longrightarrow event\text{-}trans\text{-}prem\ P\ e\ P''\rangle$
| $initial\text{-}trans\text{-}to\text{-}After_{tick}$: $\langle e \in initials\ P \Longrightarrow event\text{-}trans\text{-}prem\ P\ e\ (P\ after_\checkmark\ e)\rangle$

abbreviation *event-trans* :: $\langle('a, 'r)\ process_{ptick} \Rightarrow 'a \Rightarrow ('a, 'r)\ process_{ptick} \Rightarrow bool\rangle$
($\langle$- $\rightsquigarrow$_ -$\rangle$ [50, 3, 51] 50)
where $\langle P \rightsquigarrow_e P' \equiv ev\ e \in initials\ P \wedge event\text{-}trans\text{-}prem\ P\ (ev\ e)\ P'\rangle$

abbreviation *tick-trans* :: $\langle('a, 'r)\ process_{ptick} \Rightarrow 'r \Rightarrow ('a, 'r)\ process_{ptick} \Rightarrow bool\rangle$ ($\langle$- $\rightsquigarrow_\checkmark$ -$\rangle$ [50, 3, 51] 50)
where $\langle P \rightsquigarrow_{\checkmark r} P' \equiv \checkmark(r) \in P^0 \wedge event\text{-}trans\text{-}prem\ P\ \checkmark(r)\ P'\rangle$

We immediately show that, under the assumption of monotony of Ω, this event transition definition is equivalent to the following:

lemma *startable-imp-ev-trans-is-startable-and-FD-After*:

```
  ‹(case e of ev x ⇒ P ⇝x P' | ✓(r) ⇒ P ⇝✓r P') ⟷ e ∈ P⁰ ∧ P after✓ e ⇝τ
P'›
  if ‹⋀P Q. case e of ✓(r) ⇒ Ω P r ⇝τ Ω Q r›
proof −
  have ‹(case e of ev x ⇒ P ⇝x P' | ✓(r) ⇒ P ⇝✓r P') ⟷
        e ∈ initials P ∧ event-trans-prem P e P'› by (simp split: event_ptick.split)
  also have ‹... ⟷ e ∈ initials P ∧ P after✓ e ⇝τ P'›
  proof (insert that, safe)
    show ‹event-trans-prem P e P' ⟹ e ∈ P⁰ ⟹
          (⋀P Q. case e of ✓(r) ⇒ Ω P r ⇝τ Ω Q r) ⟹ P after✓ e ⇝τ P'›
    proof (induct rule: event-trans-prem.induct)
      case (τ-left-absorb e P' P P'')
      thus ?case
        apply (cases e)
       apply (metis (mono-tags, lifting) τ-trans-is-FD event_ptick.simps(5) mono-After_tick-FD
trans-FD)
        by (metis After_tick-def FD-antisym τ-trans-is-FD event_ptick.simps(6))
    next
      case (τ-right-absorb e P P' P'')
      thus ?case by (metis τ-trans-is-FD trans-FD)
    next
      case (initial-trans-to-After_tick e P)
      thus ?case by (simp add: τ-trans-eq)
    qed
  next
    show ‹e ∈ initials P ⟹ P after✓ e ⇝τ P' ⟹ event-trans-prem P e P'›
      by (rule τ-right-absorb[of e P ‹P after✓ e› P'])
        (simp-all add: initial-trans-to-After_tick τ-trans-is-FD)
  qed
  finally show ?thesis .
qed
```

With these two results, we are encouraged in the following theories to define:

- $P \rightsquigarrow_\tau Q \equiv P \sqsubseteq_{FD} Q$
- $P \rightsquigarrow_e P' \equiv ev\ e \in P^0 \wedge P\ after_{\checkmark}\ ev\ e \rightsquigarrow_\tau Q$
- $P \rightsquigarrow_{\checkmark r} P' \equiv \checkmark(r) \in P^0 \wedge P\ after_{\checkmark}\ \checkmark(r) \rightsquigarrow_\tau Q$

and possible variations with other refinements.

But we want to make the construction as general as possible. Therefore we will continue with the locale mechanism, eventually adding additional required assumptions for each operator, and we will instantiate with refinements at the end.

Chapter 6

Generic Operational Semantics as a Locale

Some Properties of Monotony **lemma** *FD-iff-eq-Ndet*: ‹$P \sqsubseteq_{FD} Q \longleftrightarrow P = P \sqcap Q$›
by (*auto simp add*: *Process-eq-spec failure-divergence-refine-def failure-refine-def divergence-refine-def F-Ndet D-Ndet*)

lemma *non-BOT-mono-Det-F* :
‹$P = \bot \vee P' \neq \bot \Longrightarrow Q = \bot \vee Q' \neq \bot \Longrightarrow P \sqsubseteq_F P' \Longrightarrow Q \sqsubseteq_F Q' \Longrightarrow P \Box Q \sqsubseteq_F P' \Box Q'$›
using *F-subset-imp-T-subset* **by** (*simp add*: *failure-refine-def F-Det BOT-iff-Nil-D*) *blast*

lemma *non-BOT-mono-Det-left-F* : ‹$P = \bot \vee P' \neq \bot \vee Q = \bot \Longrightarrow P \sqsubseteq_F P' \Longrightarrow P \Box Q \sqsubseteq_F P' \Box Q$ ›
and *non-BOT-mono-Det-right-F* : ‹$Q = \bot \vee Q' \neq \bot \vee P = \bot \Longrightarrow Q \sqsubseteq_F Q' \Longrightarrow P \Box Q \sqsubseteq_F P \Box Q'$›
by (*metis Det-is-BOT-iff idem-F non-BOT-mono-Det-F*)+

lemma *non-BOT-mono-Sliding-F* :
‹$P = \bot \vee P' \neq \bot \vee Q = \bot \Longrightarrow P \sqsubseteq_F P' \Longrightarrow Q \sqsubseteq_F Q' \Longrightarrow P \rhd Q \sqsubseteq_F P' \rhd Q'$›
unfolding *Sliding-def* **by** (*metis Ndet-is-BOT-iff idem-F mono-Ndet-F non-BOT-mono-Det-F*)

6.1 Definition

locale *OpSemTransitions* = *AfterExt* Ψ Ω
for Ψ :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
and Ω :: ‹$[('a, 'r)\ process_{ptick}, 'r] \Rightarrow ('a, 'r)\ process_{ptick}$›
— Just declaring the types $'a$ and $'r$. +
fixes τ-*trans* :: ‹$[('a, 'r)\ process_{ptick}, ('a, 'r)\ process_{ptick}] \Rightarrow bool$› (**infixl** ‹$\rightsquigarrow_\tau$› *50*)
assumes τ-*trans-NdetL*: ‹$P \sqcap Q \rightsquigarrow_\tau P$›

and τ-*trans-transitivity*: ‹$P \rightsquigarrow_\tau Q \Longrightarrow Q \rightsquigarrow_\tau R \Longrightarrow P \rightsquigarrow_\tau R$›
and τ-*trans-anti-mono-initials*: ‹$P \rightsquigarrow_\tau Q \Longrightarrow$ *initials* $Q \subseteq$ *initials* P›
and τ-*trans-mono-After*$_{tick}$: ‹$e \in$ *initials* $Q \Longrightarrow P \rightsquigarrow_\tau Q \Longrightarrow P$ *after*$_{\checkmark}$ $e \rightsquigarrow_\tau Q$ *after*$_{\checkmark}$ e›

begin

This locale needs to be instantiated with:

- a function Ψ::$('a, 'r)$ *process*$_{ptick} \Rightarrow 'a \Rightarrow ('a, 'r)$ *process*$_{ptick}$ that is a placeholder for the value of P *after* e when *ev* $e \notin P^0$
- a function Ω::$('a, 'r)$ *process*$_{ptick} \Rightarrow 'r \Rightarrow ('a, 'r)$ *process*$_{ptick}$ that is a placeholder for the value of P *after*$_{\checkmark}$ $\checkmark(r)$
- a binary relation ($\rightsquigarrow_\tau$) which:
  - is compatible with ($\sqcap$)
  - is transitive
  - makes *initials* anti-monotonic
  - makes *After*$_{tick}$ monotonic.

From the τ transition $P \rightsquigarrow_\tau Q$ we derive the event transition as follows:

abbreviation *ev-trans* :: ‹$[('a, 'r)$ *process*$_{ptick}, 'a, ('a, 'r)$ *process*$_{ptick}] \Rightarrow$ *bool*› (‹- $\rightsquigarrow$_ -› [*50*, *3*, *51*] *50*)
where ‹$P \rightsquigarrow_e Q \equiv$ *ev* $e \in P^0 \wedge P$ *after*$_{\checkmark}$ *ev* $e \rightsquigarrow_\tau Q$›

abbreviation *tick-trans* :: ‹$[('a, 'r)$ *process*$_{ptick}, 'r, ('a, 'r)$ *process*$_{ptick}] \Rightarrow$ *bool*› (‹- $\rightsquigarrow_{\checkmark}$ -› [*50*, *3*, *51*] *50*)
where ‹$P \rightsquigarrow_{\checkmark} r\ Q \equiv \checkmark(r) \in P^0 \wedge P$ *after*$_{\checkmark}$ $\checkmark(r) \rightsquigarrow_\tau Q$›

lemma *ev-trans-is*: ‹$P \rightsquigarrow_e Q \longleftrightarrow$ *ev* $e \in$ *initials* $P \wedge P$ *after* $e \rightsquigarrow_\tau Q$›
by (*simp add*: *After*$_{tick}$*-def*)

lemma *tick-trans-is*: ‹$P \rightsquigarrow_{\checkmark} r\ Q \longleftrightarrow \checkmark(r) \in P^0 \wedge \Omega\ P\ r \rightsquigarrow_\tau Q$›
by (*simp add*: *After*$_{tick}$*-def*)

lemma *reverse-event-trans-is*:
‹$e \in P^0 \wedge P$ *after*$_{\checkmark}$ $e \rightsquigarrow_\tau Q \longleftrightarrow$ (*case* e *of* $\checkmark(r) \Rightarrow P \rightsquigarrow_{\checkmark} r\ Q \mid$ *ev* $x \Rightarrow P \rightsquigarrow_x Q$)›
by (*simp split*: *event*$_{ptick}$*.split*)

From idempotence, commutativity and $\perp$ absorbency of ($\sqcap$), we get the following free of charge.

lemma τ-*trans-eq*: ‹$P \rightsquigarrow_\tau P$›
and τ-*trans-NdetR*: ‹$P \sqcap Q \rightsquigarrow_\tau Q$›

and *BOT-τ-trans-anything*: ‹⊥ $\rightsquigarrow_\tau$ *P*›
apply (*metis Ndet-id τ-trans-NdetL*)
apply (*metis Ndet-commute τ-trans-NdetL*)
by (*metis Ndet-BOT τ-trans-NdetL*)

lemma *BOT-ev-trans-anything*: ‹⊥ $\rightsquigarrow_e$ *P*›
and *BOT-tick-trans*: ‹⊥ $\rightsquigarrow_{\checkmark r}$ Ω ⊥ *r*›
by (*simp-all add*: *After*$_{tick}$*-BOT τ-trans-eq BOT-τ-trans-anything*)

As immediate consequences of the axioms, we prove that event transitions absorb τ transitions on right and on left.

lemma *ev-trans-τ-trans*: ‹*P* $\rightsquigarrow_e$ *P'* ⟹ *P'* $\rightsquigarrow_\tau$ *P''* ⟹ *P* $\rightsquigarrow_e$ *P''*›
and *tick-trans-τ-trans*: ‹*P* $\rightsquigarrow_{\checkmark r}$ *P'* ⟹ *P'* $\rightsquigarrow_\tau$ *P''* ⟹ *P* $\rightsquigarrow_{\checkmark r}$ *P''*›
by (*meson τ-trans-transitivity*)+

lemma *τ-trans-ev-trans*: ‹*P* $\rightsquigarrow_\tau$ *P'* ⟹ *P'* $\rightsquigarrow_e$ *P''* ⟹ *P* $\rightsquigarrow_e$ *P''*›
and *τ-trans-tick-trans*: ‹*P* $\rightsquigarrow_\tau$ *P'* ⟹ *P'* $\rightsquigarrow_{\checkmark r}$ *P''* ⟹ *P* $\rightsquigarrow_{\checkmark r}$ *P''*›
using *τ-trans-mono-After*$_{tick}$ *τ-trans-transitivity τ-trans-anti-mono-initials* **by** *blast*+

We can also add these result which will be useful later.

lemma *initial-tick-imp-τ-trans-SKIP*: ‹*P* $\rightsquigarrow_\tau$ *SKIP r*› **if** ‹✓(*r*) ∈ P^0›
proof −
from *that* **have** ‹*P* $\sqsubseteq_{FD}$ *SKIP r*› **by** (*simp add*: *initial-tick-iff-FD-SKIP*)
with *FD-iff-eq-Ndet* **have** ‹*P* = *P* ⊓ *SKIP r*› **..**
thus ‹*P* $\rightsquigarrow_\tau$ *SKIP r*› **by** (*metis τ-trans-NdetR*)
qed

lemma *exists-tick-trans-is-initial-tick*: ‹(∃ *P'*. *P* $\rightsquigarrow_{\checkmark r}$ *P'*) ⟷ ✓(*r*) ∈ P^0›
using *τ-trans-eq* **by** *blast*

There is also a major property we can already prove.

lemma *τ-trans-imp-leT* : ‹*P* $\rightsquigarrow_\tau$ *Q* ⟹ *P* $\sqsubseteq_T$ *Q*›
proof (*unfold trace-refine-def*, *rule subsetI*)
show ‹*P* $\rightsquigarrow_\tau$ *Q* ⟹ *s* ∈ $\mathcal{T}$ *Q* ⟹ *s* ∈ $\mathcal{T}$ *P*› **for** *s*
proof (*induct s arbitrary*: *P Q*)
show ‹⋀*P*. [] ∈ $\mathcal{T}$ *P*› **by** *simp*
next
fix *e s P Q*
assume *hyp* : ‹*P* $\rightsquigarrow_\tau$ *Q* ⟹ *s* ∈ $\mathcal{T}$ *Q* ⟹ *s* ∈ $\mathcal{T}$ *P*› **for** *P Q*
assume *prems* : ‹*P* $\rightsquigarrow_\tau$ *Q*› ‹*e* # *s* ∈ $\mathcal{T}$ *Q*›
from *prems*(*2*)[*THEN initials-memI*] **have** ‹*e* ∈ Q^0› **.**
show ‹*e* # *s* ∈ $\mathcal{T}$ *P*›
proof (*cases e*)
fix *r* **assume** ‹*e* = ✓(*r*)›
hence ‹*s* = []› **by** (*metis append-Cons append-Nil append-T-imp-tickFree non-tickFree-tick prems*(*2*))
with ‹*e* = ✓(*r*)› ‹*P* $\rightsquigarrow_\tau$ *Q*›[*THEN τ-trans-anti-mono-initials*] ‹*e* # *s* ∈ $\mathcal{T}$ *Q*›
show ‹*e* # *s* ∈ $\mathcal{T}$ *P*› **by** (*simp add*: *initials-def subset-iff*)

```
    next
      fix x
      assume ‹e = ev x›
      from ‹e ∈ Q⁰› prems(2) have ‹s ∈ 𝒯 (Q after✓ e)›
        by (simp add: ‹e = ev x› T-After_tick)
      from τ-trans-mono-After_tick[OF ‹e ∈ Q⁰› ‹P ⇝_τ Q›]
      have ‹P after✓ e ⇝_τ Q after✓ e› .
      from hyp[OF this ‹s ∈ 𝒯 (Q after✓ e)›] have ‹s ∈ 𝒯 (P after✓ e)› .
       with ‹e ∈ Q⁰›[THEN ‹P ⇝_τ Q›[THEN τ-trans-anti-mono-initials, THEN
set-mp]]
      show ‹e # s ∈ 𝒯 P› by (simp add: T-After_tick ‹e = ev x›)
    qed
  qed
qed
```

We can now define the concept of transition with a trace and demonstrate the first properties.

```
inductive trace-trans :: ‹('a, 'r) process_ptick ⇒ ('a, 'r) trace_ptick ⇒ ('a, 'r) pro-
cess_ptick ⇒ bool›
  (‹- ⇝*- -› [50, 3, 51] 50)
  where    trace-τ-trans : ‹P ⇝_τ P' ⟹ P ⇝* [] P'›
  |     trace-tick-trans : ‹P ⇝✓r P' ⟹ P ⇝* [✓(r)] P'›
  |  trace-Cons-ev-trans : ‹P ⇝_e  P' ⟹ P' ⇝* s P'' ⟹ P ⇝* (ev e) # s P''›

lemma trace-trans-τ-trans: ‹P ⇝*s P' ⟹ P' ⇝_τ P'' ⟹ P ⇝*s P''›
  apply (induct rule: trace-trans.induct)
  using τ-trans-transitivity trace-τ-trans apply blast
  using τ-trans-transitivity trace-tick-trans apply blast
  using trace-Cons-ev-trans by blast

lemma τ-trans-trace-trans:  ‹P ⇝_τ P' ⟹ P' ⇝*s P'' ⟹ P ⇝*s P''›
  apply (rotate-tac, induct rule: trace-trans.induct)
  using τ-trans-transitivity trace-τ-trans apply blast
  using τ-trans-tick-trans trace-trans.simps apply blast
  using τ-trans-ev-trans trace-Cons-ev-trans by blast


lemma BOT-trace-trans-tickFree-anything : ‹tickFree s ⟹ ⊥ ⇝*s P›
proof (induct s arbitrary: P)
  show ‹⋀P. tickFree [] ⟹ ⊥ ⇝*[] P›
    by (simp add: BOT-τ-trans-anything trace-τ-trans)
next
  fix e s P
  assume prem: ‹tickFree (e # s)› and hyp: ‹tickFree s ⟹ ⊥ ⇝*s P› for P
  have ∗ : ‹tickFree s› using prem by auto
  obtain a where ‹e = ev a› by (meson is-ev-def prem tickFree-Cons-iff)
  thus ‹⊥ ⇝*e # s P›
    by simp (rule trace-Cons-ev-trans[OF - hyp];
        simp add: ∗ After_tick-BOT BOT-τ-trans-anything)
```

qed

6.2 Consequences of $P \rightsquigarrow^* s\ Q$ on $\mathcal{F}$, $\mathcal{T}$ and $\mathcal{D}$

lemma *trace-trans-imp-F-if-τ-trans-imp-leF*:
‹$P \rightsquigarrow^* s\ Q \Longrightarrow X \in \mathcal{R}\ Q \Longrightarrow (s,\ X) \in \mathcal{F}\ P$›
if ‹$\forall P\ Q.\ P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_F Q$›
proof (*induct rule*: *trace-trans.induct*)
show ‹$P \rightsquigarrow_\tau Q \Longrightarrow X \in \mathcal{R}\ Q \Longrightarrow ([],\ X) \in \mathcal{F}\ P$› **for** $P\ Q$
by (*meson failure-refine-def in-mono Refusals-iff that*)
next
show ‹$P \rightsquigarrow_{✓} r\ Q \Longrightarrow X \in \mathcal{R}\ Q \Longrightarrow ([✓(r)],\ X) \in \mathcal{F}\ P$› **for** $P\ Q\ r$
by (*metis append-Nil mem-Collect-eq initials-def tick-T-F*)
next
fix $P\ e\ Q\ s\ Q'$
assume $*$: ‹$P \rightsquigarrow_e Q$› ‹$X \in \mathcal{R}\ Q' \Longrightarrow (s,\ X) \in \mathcal{F}\ Q$› ‹$X \in \mathcal{R}\ Q'$›
have ‹$P\ after_{✓}\ ev\ e \sqsubseteq_F Q$› **using** $*(1)$ *that* **by** *blast*
hence ‹$(s,\ X) \in \mathcal{F}\ (P\ after_{✓}\ ev\ e)$› **by** (*simp add*: *failure-refine-def subsetD* $*(2,\ 3)$)
thus ‹$(ev\ e\ \#\ s,\ X) \in \mathcal{F}\ P$› **by** (*simp add*: *F-After*$_{tick}$ $*(1)$)
qed

lemma *trace-trans-imp-T*: ‹$P \rightsquigarrow^* s\ Q \Longrightarrow s \in \mathcal{T}\ P$›
proof (*induct rule*: *trace-trans.induct*)
show ‹$P \rightsquigarrow_\tau Q \Longrightarrow [] \in \mathcal{T}\ P$› **for** $P\ Q$ **by** *simp*
next
show ‹$P \rightsquigarrow_{✓} r\ Q \Longrightarrow [✓(r)] \in \mathcal{T}\ P$› **for** $P\ Q\ r$
by (*simp add*: *initials-def*)
next
fix $P\ e\ Q\ s\ Q'$
assume $*$: ‹$P \rightsquigarrow_e Q$› ‹$s \in \mathcal{T}\ Q$›
have ‹$P\ after_{✓}\ ev\ e \sqsubseteq_T Q$› **using** $*(1)$ *τ-trans-imp-leT* **by** *blast*
hence ‹$s \in \mathcal{T}\ (P\ after_{✓}\ ev\ e)$› **by** (*simp add*: $*(2)$ *subsetD trace-refine-def*)
with $*(1)$ *list.collapse* **show** ‹$ev\ e\ \#\ s \in \mathcal{T}\ P$›
by (*force simp add*: *T-After*$_{tick}$ *initials-def*)
qed

lemma *tickFree-trace-trans-BOT-imp-D-if-τ-trans-BOT-imp-eq-BOT-weak*:
‹*tickFree* $s \Longrightarrow P \rightsquigarrow^* s \perp \Longrightarrow s \in \mathcal{D}\ P$›
if ‹$\forall P.\ P \rightsquigarrow_\tau \perp \longrightarrow P = \perp$›
proof (*induct s arbitrary*: P)
show ‹$P \rightsquigarrow^* [] \perp \Longrightarrow [] \in \mathcal{D}\ P$› **for** P
apply (*erule trace-trans.cases*)
using *BOT-iff-Nil-D that* **by** *blast+*
next

fix $e\ s\ P$
assume *prems* : ‹*tickFree* $(e\ \#\ s)$› ‹$P \rightsquigarrow^* e\ \#\ s\ \bot$›
assume *hyp*: ‹*tickFree* $s \Longrightarrow P \rightsquigarrow^* s\ \bot \Longrightarrow s \in \mathcal{D}\ P$› **for** P
from *prems*(*1*) **have** ‹*tickFree* s› **by** *simp*
from *prems*(*2*) **have** ‹P *after*$_{\checkmark}$ $e \rightsquigarrow^* s\ \bot$›
by (*cases rule*: *trace-trans.cases*)
(*auto simp add*: *trace-*τ*-trans intro*: τ*-trans-trace-trans*)
show ‹$e\ \#\ s \in \mathcal{D}\ P$›
apply (*rule trace-trans.cases*[*OF prems*(*2*)])
using *hyp*[*OF* ‹*tickFree* s› ‹P *after*$_{\checkmark}$ $e \rightsquigarrow^* s\ \bot$›] *prems*(*1*)
by (*simp-all add*: *D-After*$_{tick}$)
qed

6.3 Characterizations for $P \rightsquigarrow^* s\ Q$

lemma *trace-trans-iff* :
‹$P \rightsquigarrow^*\ []\ Q \longleftrightarrow P \rightsquigarrow_\tau\ Q$›
‹$P \rightsquigarrow^*\ [\checkmark(r)]\ Q \longleftrightarrow P \rightsquigarrow_{\checkmark r}\ Q$›
‹$P \rightsquigarrow^*\ (ev\ e)\ \#\ s\ Q' \longleftrightarrow (\exists\, Q.\ P \rightsquigarrow_e\ Q \wedge Q \rightsquigarrow^*\ s\ Q')$›
‹$(P \rightsquigarrow^*\ s\ @\ [f]\ Q') \longleftrightarrow$
tickFree $s \wedge (\exists\, Q.\ P \rightsquigarrow^* s\ Q \wedge (\mathit{case}\ f\ \mathit{of}\ \checkmark(r) \Rightarrow Q \rightsquigarrow_{\checkmark r}\ Q' \mid ev\ x \Rightarrow Q \rightsquigarrow_x Q'))$›
‹*front-tickFree* $(s\ @\ t) \Longrightarrow (P \rightsquigarrow^* s\ @\ t\ Q') \longleftrightarrow (\exists\, Q.\ P \rightsquigarrow^* s\ Q \wedge Q \rightsquigarrow^* t\ Q')$›
proof −
show *f1* : ‹$\bigwedge P\ Q.\ P \rightsquigarrow^*\ []\ Q \longleftrightarrow P \rightsquigarrow_\tau\ Q$›
and *f2* : ‹$\bigwedge P\ Q.\ P \rightsquigarrow^*\ [\checkmark(r)]\ Q \longleftrightarrow P \rightsquigarrow_{\checkmark r}\ Q$›
and *f3* : ‹$\bigwedge P\ Q'\ e.\ P \rightsquigarrow^*\ (ev\ e)\ \#\ s\ Q' \longleftrightarrow (\exists\, Q.\ P \rightsquigarrow_e\ Q \wedge Q \rightsquigarrow^*\ s\ Q')$›
by ((*subst trace-trans.simps*, *auto*)[*1*])+

show *f4* : ‹$(P \rightsquigarrow^*\ s\ @\ [f]\ Q') \longleftrightarrow$
tickFree $s \wedge (\exists\, Q.\ P \rightsquigarrow^* s\ Q \wedge (\mathit{case}\ f\ \mathit{of}\ \checkmark(r) \Rightarrow Q \rightsquigarrow_{\checkmark r}\ Q' \mid ev\ x \Rightarrow Q \rightsquigarrow_x Q'))$› **for** $s\ f\ P\ Q'$
proof *safe*
from *append-T-imp-tickFree trace-trans-imp-T*
show ‹$P \rightsquigarrow^* s\ @\ [f]\ Q' \Longrightarrow$ *tickFree* s› **by** *blast*
next
show ‹$P \rightsquigarrow^* s\ @\ [f]\ Q' \Longrightarrow \exists\, Q.\ P \rightsquigarrow^* s\ Q \wedge (\mathit{case}\ f\ \mathit{of}\ ev\ x \Rightarrow Q \rightsquigarrow_x Q' \mid \checkmark(r) \Rightarrow Q \rightsquigarrow_{\checkmark r} Q')$›
proof (*induct s arbitrary*: *P*)
case *Nil*
thus *?case*
apply (*subst* (*asm*) *trace-trans.simps*, *cases f*; *simp*)
using τ*-trans-eq* τ*-trans-transitivity f1* **by** *blast*+
next
case (*Cons e s*)
from *Cons.prems* **have** $*$: ‹$e \in \mathit{initials}\ P \wedge P$ *after*$_{\checkmark}$ $e \rightsquigarrow^* s\ @\ [f]\ Q'$›
by (*subst* (*asm*) *trace-trans.simps*)
(*auto simp add*: *intro*: τ*-trans-trace-trans*)
with *Cons.hyps* **obtain** Q

```
      where ∗∗ : ‹P after✓ e ⇝∗s Q›
        ‹(case f of ev x ⇒ Q ⇝x Q′ | ✓(r) ⇒ Q ⇝✓r Q′)› by blast
    have ‹P ⇝∗e # s Q›
    proof (cases e)
      fix e′
      assume ‹e = ev e′›
      thus ‹P ⇝∗e # s Q›
        apply simp
        by (rule trace-Cons-ev-trans[OF - ∗∗(1)]) (use ∗ τ-trans-eq in blast)
    next
       from Cons.prems have ‹e = ✓(r) ⟹ s = []› for r by (subst (asm)
trace-trans.simps) auto
     thus ‹e = ✓(r) ⟹ P ⇝∗e # s Q› for r using ∗ ∗∗(1) f1 f2 trace-tick-trans
by auto
    qed
    with ∗∗(2) show ‹∃ Q. P ⇝∗ e # s Q ∧ (case f of ev x ⇒ Q ⇝x Q′ | ✓(r)
⇒ Q ⇝✓r Q′)› by blast
  qed
 next
  show ‹tickFree s ⟹ P ⇝∗s Q ⟹ (case f of ev x ⇒ Q ⇝x Q′ | ✓(r) ⇒ Q
⇝✓r Q′)
       ⟹ P ⇝∗s @ [f] Q′› for Q
  proof (induct s arbitrary: P Q)
    show ‹tickFree [] ⟹ P ⇝∗[] Q ⟹
        (case f of ev x ⇒ Q ⇝x Q′ | ✓(r) ⇒ Q ⇝✓r Q′) ⟹ P ⇝∗[] @ [f] Q′›
for P Q
      apply (cases f; simp add: f1 f2)
      apply (meson τ-trans-eq τ-trans-ev-trans trace-Cons-ev-trans trace-τ-trans)
      using τ-trans-tick-trans trace-tick-trans by blast
  next
    case (Cons e s)
    from Cons.prems(2) have ∗ : ‹e ∈ initials P ∧ P after✓ e ⇝∗s Q›
      by (subst (asm) trace-trans.simps)
        (auto simp add: f1 intro: τ-trans-trace-trans)
    show ?case
    proof (cases e)
      fix e′
      assume ‹e = ev e′›
      thus ‹P ⇝∗(e # s) @ [f] Q′›
        by (cases f; simp)
          (metis (no-types, lifting) ∗ Cons.hyps Cons.prems(1, 3)
            τ-trans-eq tickFree-Cons-iff trace-trans.simps)+
    next
      show ‹e = ✓(r) ⟹ P ⇝∗(e # s) @ [f] Q′› for r using Cons.prems(1)
by auto
    qed
  qed
 qed
```

show ‹*front-tickFree* $(s \mathbin{@} t) \Longrightarrow P \rightsquigarrow^* s \mathbin{@} t\ Q' \longleftrightarrow (\exists\, Q.\ P \rightsquigarrow^* s\ Q \wedge Q \rightsquigarrow^* t\ Q')$›
proof (*induct t arbitrary*: Q' *rule*: *rev-induct*)
show ‹$P \rightsquigarrow^* s \mathbin{@} []\ Q' \longleftrightarrow (\exists\, Q.\ P \rightsquigarrow^* s\ Q \wedge Q \rightsquigarrow^* []\ Q')$› **for** Q'
by (*metis* τ*-trans-eq append.right-neutral trace-trans-*τ*-trans f1*)
next
case (*snoc e t*)
from *snoc.prems* **have** \$: ‹*tickFree s*› ‹*tickFree t*›
by (*simp-all add*: *front-tickFree-append-iff*)
show ‹$P \rightsquigarrow^* s \mathbin{@} t \mathbin{@} [e]\ Q' \longleftrightarrow (\exists\, Q.\ P \rightsquigarrow^* s\ Q \wedge Q \rightsquigarrow^* t \mathbin{@} [e]\ Q')$›
proof (*intro iffI*)
assume *assm* : ‹$P \rightsquigarrow^* s \mathbin{@} t \mathbin{@} [e]\ Q'$›
from *f4*[*of P* ‹$s \mathbin{@} t$› *e* Q', *simplified*] *assm* \$ **obtain** Q
where $*$: ‹$P \rightsquigarrow^* s \mathbin{@} t\ Q$› ‹*case e of ev x* $\Rightarrow Q \rightsquigarrow_x Q' \mid \checkmark(r) \Rightarrow Q \rightsquigarrow_{\checkmark r} Q'$› **by** *blast*
obtain R **where** $**$: ‹$P \rightsquigarrow^* s\ R$› ‹$R \rightsquigarrow^* t\ Q$›
by (*metis* $*(1)$ *append.assoc front-tickFree-dw-closed snoc.hyps snoc.prems*)
show ‹$\exists\, Q.\ P \rightsquigarrow^* s\ Q \wedge Q \rightsquigarrow^* t \mathbin{@} [e]\ Q'$›
proof (*intro exI conjI*)
show ‹$P \rightsquigarrow^* s\ R$› **by** (*fact* $**(1)$)
next
from $*(2)$ $**(2)$ **show** ‹$R \rightsquigarrow^* t \mathbin{@} [e]\ Q'$› **by** (*simp add*: *f4* \$(*2*)) *blast*
qed
next
assume ‹$\exists\, Q.\ P \rightsquigarrow^* s\ Q \wedge Q \rightsquigarrow^* t \mathbin{@} [e]\ Q'$›
then obtain Q **where** $*$: ‹$P \rightsquigarrow^* s\ Q$› ‹$Q \rightsquigarrow^* t \mathbin{@} [e]\ Q'$› **by** *blast*
from *f4*[*of Q t e* Q', *simplified this*(*2*)]
obtain R **where** ‹$Q \rightsquigarrow^* t\ R$› ‹*case e of ev x* $\Rightarrow R \rightsquigarrow_x Q' \mid \checkmark(r) \Rightarrow R \rightsquigarrow_{\checkmark r} Q'$› **by** *blast*
with $*(1)$ **show** ‹$P \rightsquigarrow^* s \mathbin{@} t \mathbin{@} [e]\ Q'$›
by (*simp add*: *f4*[*of P* ‹$s \mathbin{@} t$› *e* Q', *simplified*] \$, *cases e*; *simp*)
(*metis append.assoc front-tickFree-dw-closed snoc.hyps snoc.prems*)+
qed
qed
qed

6.4 Finally: $P \rightsquigarrow^* s\ Q$ is $P\ after_{\mathcal{T}}\ s \rightsquigarrow_\tau Q$

theorem *trace-trans-iff-T-and-After*$_{trace}$-τ-*trans* :
‹$(P \rightsquigarrow^* s\ Q) \longleftrightarrow s \in \mathcal{T}\ P \wedge P\ after_{\mathcal{T}}\ s \rightsquigarrow_\tau Q$›
proof −
have ‹$s \in \mathcal{T}\ P \Longrightarrow (P \rightsquigarrow^* s\ Q) \longleftrightarrow P\ after_{\mathcal{T}}\ s \rightsquigarrow_\tau Q$›
— This is a trick to reuse the proof of this less powerful version (*trace-trans-imp-T* was not proven at the time...).
proof (*intro iffI*)
show ‹$P \rightsquigarrow^* s\ Q \Longrightarrow s \in \mathcal{T}\ P \Longrightarrow P\ after_{\mathcal{T}}\ s \rightsquigarrow_\tau Q$›
proof (*induct s arbitrary*: $P\ Q$)
show ‹$\bigwedge P\ Q.\ P \rightsquigarrow^* []\ Q \Longrightarrow [] \in \mathcal{T}\ P \Longrightarrow P\ after_{\mathcal{T}}\ [] \rightsquigarrow_\tau Q$›
using *trace-trans.cases* **by** *auto*

next
show ‹($\bigwedge P\ Q.\ P \rightsquigarrow^* s\ Q \Longrightarrow s \in \mathcal{T}\ P \Longrightarrow P\ after_{\mathcal{T}}\ s \rightsquigarrow_\tau Q) \Longrightarrow$
$P \rightsquigarrow^* e \# s\ Q \Longrightarrow e \# s \in \mathcal{T}\ P \Longrightarrow P\ after_{\mathcal{T}}\ (e \# s) \rightsquigarrow_\tau Q$› **for** $s\ e\ P\ Q$
apply (*cases e*; *simp*)
by (*meson* τ*-trans-trace-trans trace-trans-iff*(*3*) *trace-trans-imp-T*)
(*metis After*$_{trace}$*.simps*(*1*) *T-imp-front-tickFree front-tickFree-Cons-iff*
non-tickFree-tick tickFree-Cons-iff tickFree-Nil trace-trans-iff(*2*))
qed
next
show ‹$P\ after_{\mathcal{T}}\ s \rightsquigarrow_\tau Q \Longrightarrow s \in \mathcal{T}\ P \Longrightarrow P \rightsquigarrow^* s\ Q$›
proof (*induct arbitrary*: $P\ Q$ *rule*: *After*$_{trace}$*.induct*)
show ‹$\bigwedge P\ Q.\ P\ after_{\mathcal{T}}\ [] \rightsquigarrow_\tau Q \Longrightarrow [] \in \mathcal{T}\ P \Longrightarrow P \rightsquigarrow^* []\ Q$›
by (*simp add*: τ*-trans-trans*)
next
fix e **and** $s :: \langle('a, 'r)\ trace_{ptick}\rangle$ **and** $Q\ P$
assume *hyp* : ‹$P\ after_{\mathcal{T}}\ s \rightsquigarrow_\tau Q \Longrightarrow s \in \mathcal{T}\ P \Longrightarrow P \rightsquigarrow^* s\ Q$› **for** $P\ Q$
assume *prems* : ‹$P\ after_{\mathcal{T}}\ (e \# s) \rightsquigarrow_\tau Q$› ‹$e \# s \in \mathcal{T}\ P$›
have $*$: ‹$e \in initials\ P \wedge s \in \mathcal{T}\ (P\ after_{\checkmark}\ e)$›
by (*metis After*$_{trace}$*.simps*(*1*, *2*) *initials-memI*
T-After$_{trace}$ *append-Cons append-Nil prems*(*2*))
show ‹$P \rightsquigarrow^* e \# s\ Q$›
proof (*cases e*)
fix e'
assume $**$: ‹$e = ev\ e'$›
from $*$ $**$ **have** ‹$P \rightsquigarrow_{e'} P\ after_{\checkmark}\ (ev\ e')$› **by** (*simp add*: τ*-trans-eq*)
thus ‹$P \rightsquigarrow^* e \# s\ Q$›
by (*subst* $**$, *rule trace-Cons-ev-trans*[*OF - hyp*[*OF prems*(*1*)[*simplified*]
$*$[*THEN conjunct2*], *simplified* $**$]])
next
have ‹$e = \checkmark(r) \Longrightarrow s = []$› **for** r
by (*metis T-imp-front-tickFree event*$_{ptick}$*.disc*(*2*) *front-tickFree-Cons-iff*
prems(*2*))
with $*$ *prems*(*1*) *trace-tick-trans*
show ‹$e = \checkmark(r) \Longrightarrow P \rightsquigarrow^* e \# s\ Q$› **for** r **by** *auto*
qed
qed
qed
with *trace-trans-imp-T* **show** ‹$(P \rightsquigarrow^* s\ Q) \longleftrightarrow s \in \mathcal{T}\ P \wedge P\ after_{\mathcal{T}}\ s \rightsquigarrow_\tau Q$›
by *blast*
qed

As corollaries we obtain the reciprocal results of

$[\![\forall P\ Q.\ P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_F Q;\ ?P \rightsquigarrow^* ?s\ ?Q;\ ?X \in \mathcal{R}\ ?Q]\!] \Longrightarrow (?s,\ ?X) \in \mathcal{F}\ ?P$

$?P \rightsquigarrow^* ?s\ ?Q \Longrightarrow ?s \in \mathcal{T}\ ?P$

$[\![\forall P.\ P \rightsquigarrow_\tau \bot \longrightarrow P = \bot;\ tF\ ?s;\ ?P \rightsquigarrow^* ?s\ \bot]\!] \Longrightarrow ?s \in \mathcal{D}\ ?P$

lemma *tickFree-F-imp-exists-trace-trans*:

$\langle tickFree\ s \Longrightarrow (s,\ X) \in \mathcal{F}\ P \Longrightarrow \exists\, Q.\ (P \rightsquigarrow^* s\ Q) \wedge X \in \mathcal{R}\ Q\rangle$
by (*meson F-T F-imp-R-After*$_{trace}$ *trace-trans-iff-T-and-After*$_{trace}$-τ-*trans* τ-*trans-eq*)

lemma *T-imp-exists-trace-trans*: $\langle s \in \mathcal{T}\ P \Longrightarrow \exists\, Q.\ P \rightsquigarrow^* s\ Q\rangle$
using *trace-trans-iff-T-and-After*$_{trace}$-τ-*trans* τ-*trans-eq* **by** *blast*

lemma *tickFree-D-imp-trace-trans-BOT*: $\langle tickFree\ s \Longrightarrow s \in \mathcal{D}\ P \Longrightarrow P \rightsquigarrow^* s\ \bot\rangle$
by (*simp add*: *D-imp-After*$_{trace}$-*is-BOT D-T* τ-*trans-eq*
trace-trans-iff-T-and-After$_{trace}$-τ-*trans*)

And therefore, we obtain equivalences.

lemma *F-trace-trans-reality-check-weak*:
$\langle \forall\, P\ Q.\ P \rightsquigarrow_\tau\ Q \longrightarrow P \sqsubseteq_F\ Q \Longrightarrow tickFree\ s \Longrightarrow$
$(s,\ X) \in \mathcal{F}\ P \longleftrightarrow (\exists\, Q.\ (P \rightsquigarrow^* s\ Q) \wedge X \in \mathcal{R}\ Q)\rangle$
using *tickFree-F-imp-exists-trace-trans trace-trans-imp-F-if-*τ-*trans-imp-leF* **by** *blast*

lemma *T-trace-trans-reality-check*: $\langle s \in \mathcal{T}\ P \longleftrightarrow (\exists\, Q.\ P \rightsquigarrow^* s\ Q)\rangle$
using *T-imp-exists-trace-trans trace-trans-imp-T* **by** *blast*

lemma *D-trace-trans-reality-check-weak*:
$\langle \forall\, P.\ P \rightsquigarrow_\tau \bot \longrightarrow P = \bot \Longrightarrow tickFree\ s \Longrightarrow s \in \mathcal{D}\ P \longleftrightarrow P \rightsquigarrow^* s\ \bot\rangle$
using *tickFree-D-imp-trace-trans-BOT*
tickFree-trace-trans-BOT-imp-D-if-τ-*trans-BOT-imp-eq-BOT-weak* **by** *blast*

When we have more information on $P \rightsquigarrow_\tau Q$, we obtain:

lemma *STOP-trace-trans-iff*: $\langle STOP \rightsquigarrow^* s\ P \longleftrightarrow s = [] \wedge P = STOP\rangle$
using *STOP-T-iff* τ-*trans-imp-leT*
by (*subst trace-trans.simps*) (*auto simp add*: τ-*trans-eq*)

lemma Ω-*SKIP-is-STOP-imp-*τ-*trans-imp-leF-imp-SKIP-trace-trans-iff*:
$\langle \Omega\ (SKIP\ r)\ r = STOP \Longrightarrow (\bigwedge P\ Q.\ P \rightsquigarrow_\tau\ Q \Longrightarrow P \sqsubseteq_F\ Q) \Longrightarrow$
$(SKIP\ r \rightsquigarrow^* s\ P) \longleftrightarrow s = [] \wedge P = SKIP\ r \vee s = [\checkmark(r)] \wedge P = STOP\rangle$
using *SKIP-F-iff STOP-F-iff*
apply (*subst trace-trans.simps*)
apply (*auto simp add*: *After*$_{tick}$-*SKIP* τ-*trans-eq*)
by *fastforce blast+*

lemma *trace-trans-imp-initials-subset-initials-After*$_{trace}$:
$\langle P \rightsquigarrow^* s\ Q \Longrightarrow initials\ Q \subseteq initials\ (P\ after_{\mathcal{T}}\ s)\rangle$
using *trace-trans-iff-T-and-After*$_{trace}$-τ-*trans*
anti-mono-initials-T trace-trans-imp-T τ-*trans-imp-leT* **by** *blast*

lemma *imp-trace-trans-imp-initial*:
$\langle P \rightsquigarrow^* (s\ @\ e\ \#\ t)\ Q \Longrightarrow e \in initials\ (P\ after_{\mathcal{T}}\ s)\rangle$
using *initials-After*$_{trace}$ *trace-trans-imp-T* **by** *blast*

Under additional assumptions, we can show that the event transition and the trace transition are admissible.

lemma *ev-trans-adm-weak*[*simp*]:
assumes τ-*trans-adm*:
‹$\bigwedge u\ v.$ *cont* $(u :: 'b :: cpo \Rightarrow ('a, 'r)\ process_{ptick}) \Longrightarrow$ *monofun* $v \Longrightarrow$ *adm*$(\lambda x.\ u\ x \rightsquigarrow_\tau v\ x)$›
and Ψ-*cont-hyp* : ‹*cont* $(\lambda P.\ \Psi\ P\ e)$›
and *cont-u*: ‹*cont* $(u :: 'b \Rightarrow ('a, 'r)\ process_{ptick})$› **and** *monofun-v* : ‹*monofun* *v*›
shows ‹*adm*$(\lambda x.\ u\ x \rightsquigarrow_e (v\ x))$›
apply (*intro adm-conj*)
by (*fact initial-adm*[*OF cont-u*])
(*rule* τ-*trans-adm*[*OF - monofun-v*], *simp add*: Ψ-*cont-hyp cont-u*)

lemma *tick-trans-adm-weak*[*simp*]:
assumes τ-*trans-adm*:
‹$\bigwedge u\ v.$ *cont* $(u :: 'b :: cpo \Rightarrow ('a, 'r)\ process_{ptick}) \Longrightarrow$ *monofun* $v \Longrightarrow$ *adm*$(\lambda x.\ u\ x \rightsquigarrow_\tau v\ x)$›
and Ω-*cont-hyp* : ‹*cont* $(\lambda P.\ \Omega\ P\ r)$›
and *cont-u*: ‹*cont* $(u :: 'b \Rightarrow ('a, 'r)\ process_{ptick})$› **and** *monofun-v* : ‹*monofun* *v*›
shows ‹*adm*$(\lambda x.\ u\ x \rightsquigarrow_{\checkmark r} (v\ x))$›
apply (*intro adm-conj*)
by (*fact initial-adm*[*OF cont-u*])
(*rule* τ-*trans-adm*[*OF - monofun-v*], *simp add*: Ω-*cont-hyp cont-u*)

lemma *trace-trans-adm-weak*[*simp*]:
assumes τ-*trans-adm*: ‹$\bigwedge u\ v.$ *cont* $(u :: 'b :: cpo \Rightarrow ('a, 'r)\ process_{ptick}) \Longrightarrow$ *monofun* $v \Longrightarrow$ *adm* $(\lambda x.\ u\ x \rightsquigarrow_\tau v\ x)$›
and $After_{trace}$-*cont-hyps*: ‹$\forall x.$ *ev* $x \in$ *set* $s \longrightarrow$ *cont* $(\lambda P.\ \Psi\ P\ x)$› ‹$\forall r.$ ✓$(r) \in$ *set* $s \longrightarrow$ *cont* $(\lambda P.\ \Omega\ P\ r)$›
and *cont-u*: ‹*cont* $(u :: 'b :: cpo \Rightarrow ('a, 'r)\ process_{ptick})$› **and** *monofun-v*: ‹*monofun* *v*›
shows ‹*adm* $(\lambda x.\ u\ x \rightsquigarrow^*\ s\ (v\ x))$›
apply (*subst trace-trans-iff-T-and-After*$_{trace}$-τ-*trans*)
apply (*intro adm-conj*)
apply (*solves* ‹*simp add*: *adm-in-T cont-u*›)
by (*rule* τ-*trans-adm*[*OF - monofun-v*], *rule* $After_{trace}$-*cont*;
simp add: $After_{trace}$-*cont-hyps cont-u*)

6.5 General Rules of Operational Semantics

We can now derive some rules or the operational semantics that we are defining.

lemma *SKIP-trans-tick-*Ω-*SKIP*: ‹*SKIP* $r \rightsquigarrow_{\checkmark r} \Omega$ (*SKIP* r) r›
by (*simp add*: $After_{tick}$-*SKIP* τ-*trans-eq*)

lemmas *SKIP-OpSem-rule* = *SKIP-trans-tick-Ω-SKIP*

This is quite obvious, but we can get better.

lemma *initial-tick-imp-tick-trans-Ω-SKIP*: ‹✓(*r*) ∈ P^0 ⟹ $P \leadsto_{✓ r}$ Ω (*SKIP r*) *r*›
using *SKIP-trans-tick-Ω-SKIP* *τ-trans-tick-trans*
initial-tick-imp-τ-trans-SKIP **by** *blast*

lemma *tick-trans-imp-tick-trans-Ω-SKIP* : ‹∃ *P′*. *P* $\leadsto_{✓ r}$ *P′* ⟹ $P \leadsto_{✓ r}$ Ω (*SKIP r*) *r*›
using *initial-tick-imp-tick-trans-Ω-SKIP* **by** *blast*

lemma *SKIP-cant-ev-trans*: ‹¬ *SKIP r* $\leadsto_e$ *P*›
and *STOP-cant-ev-trans*: ‹¬ *STOP* $\leadsto_e$ *P*›
and *STOP-cant-tick-trans*: ‹¬ *STOP* $\leadsto_{✓ r}$ *P*› **by** *simp-all*

lemma *ev-trans-Mprefix*: ‹*e* ∈ *A* ⟹ □*a* ∈ *A* → *P a* $\leadsto_e$ (*P e*)›
by (*simp add*: $After_{tick}$*-def After-Mprefix τ-trans-eq initials-Mprefix*)

lemma *ev-trans-Mndetprefix*: ‹*e* ∈ *A* ⟹ ⊓*a* ∈ *A* → *P a* $\leadsto_e$ (*P e*)›
by (*simp add*: $After_{tick}$*-def After-Mndetprefix τ-trans-eq initials-Mndetprefix*)

lemma *ev-trans-prefix*: ‹*e* → *P* $\leadsto_e$ *P*›
by (*metis ev-trans-Mprefix insertI1 write0-def*)

lemmas *prefix-OpSem-rules* = *ev-trans-prefix ev-trans-Mprefix ev-trans-Mndetprefix*

lemma *τ-trans-GlobalNdet*: ‹⊓*a* ∈ *A*. *P a* $\leadsto_\tau$ *P e*› **if** ‹*e* ∈ *A*›
proof −
have ‹⊓*a* ∈ *A*. *P a* = *P e* ⊓ (⊓*a* ∈ *A*. *P a*)›
by (*metis GlobalNdet-factorization-union GlobalNdet-unit*
empty-iff insertI1 insert-absorb insert-is-Un that)
thus ‹⊓*a* ∈ *A*. *P a* $\leadsto_\tau$ *P e*› **by** (*metis τ-trans-NdetL*)
qed

lemmas *Ndet-OpSem-rules* = *τ-trans-NdetL τ-trans-NdetR τ-trans-GlobalNdet*

lemma *τ-trans-fix-point*: ‹*cont f* ⟹ *P* = (*μ X*. *f X*) ⟹ *P* ⇝$_\tau$ *f P*›
by (*metis τ-trans-eq cont-process-rec*)

lemmas *fix-point-OpSem-rule* = *τ-trans-fix-point*

lemma *ev-trans-DetL*: ‹*P* ⇝$_e$ *P'* ⟹ *P* □ *Q* ⇝$_e$ *P'›*
by (*metis After*$_{tick}$*-Det-is-After*$_{tick}$*-Ndet UnCI τ-trans-NdetL τ-trans-ev-trans initials-Det*)

lemma *ev-trans-DetR*: ‹*Q* ⇝$_e$ *Q'* ⟹ *P* □ *Q* ⇝$_e$ *Q'›*
by (*metis Det-commute ev-trans-DetL*)

lemma *tick-trans-DetL*: ‹*P* ⇝$_{✓r}$ *P'* ⟹ *P* □ *Q* ⇝$_{✓r}$ Ω (*SKIP r*) *r›*
by (*metis UnCI initials-Det initial-tick-imp-tick-trans-Ω-SKIP*)

lemma *tick-trans-DetR*: ‹*Q* ⇝$_{✓r}$ *Q'* ⟹ *P* □ *Q* ⇝$_{✓r}$ Ω (*SKIP r*) *r›*
by (*metis Det-commute tick-trans-DetL*)

lemma *ev-trans-GlobalDet*:
‹□*a* ∈ *A*. *P a* ⇝$_e$ *Q*› **if** ‹*a* ∈ *A*› **and** ‹*P a* ⇝$_e$ *Q*›
proof (*cases* ‹*A* = {*a*}›)
show ‹*A* = {*a*} ⟹ □*a* ∈ *A*. *P a* ⇝$_e$ *Q*› **by** (*simp add*: ‹*P a* ⇝$_e$ *Q*›)
next
assume ‹*A* ≠ {*a*}›
with ‹*a* ∈ *A*› **obtain** *A'* **where** ‹*a* ∉ *A'*› ‹*A* = {*a*} ∪ *A'*›
by (*metis insert-is-Un mk-disjoint-insert*)
have ‹□*a* ∈ *A*. *P a* = *P a* □ *GlobalDet A' P*›
by (*simp add*: ‹*A* = {*a*} ∪ *A'*› *GlobalDet-distrib-unit-bis*[*OF* ‹*a* ∉ *A'*›])
also from *ev-trans-DetL* ‹*P a* ⇝$_e$ *Q*› **have** ‹... ⇝$_e$ *Q*› **by** *blast*
finally show ‹□*a* ∈ *A*. *P a* ⇝$_e$ *Q*› **.**
qed

lemma *tick-trans-GlobalDet*:
‹□*a* ∈ *A*. *P a* ⇝$_{✓r}$ Ω (*SKIP r*) *r*› **if** ‹*a* ∈ *A*› **and** ‹*P a* ⇝$_{✓r}$ *Q*›
proof (*cases* ‹*A* = {*a*}›)
show ‹*A* = {*a*} ⟹ □*a* ∈ *A*. *P a* ⇝$_{✓r}$ Ω (*SKIP r*) *r*›
by (*simp add*: *initial-tick-imp-tick-trans-Ω-SKIP* ‹*P a* ⇝$_{✓r}$ *Q*›)
next
assume ‹*A* ≠ {*a*}›
with ‹*a* ∈ *A*› **obtain** *A'* **where** ‹*a* ∉ *A'*› ‹*A* = {*a*} ∪ *A'*›
by (*metis insert-is-Un mk-disjoint-insert*)
have ‹□*a* ∈ *A*. *P a* = *P a* □ *GlobalDet A' P*›
by (*simp add*: ‹*A* = {*a*} ∪ *A'*› *GlobalDet-distrib-unit-bis*[*OF* ‹*a* ∉ *A'*›])
also from *tick-trans-DetL* ‹*P a* ⇝$_{✓r}$ *Q*› **have** ‹... ⇝$_{✓r}$ Ω (*SKIP r*) *r*› **by** *blast*
finally show ‹□*a* ∈ *A*. *P a* ⇝$_{✓r}$ Ω (*SKIP r*) *r*› **.**

qed

lemma *ev-trans-SlidingL*: ‹$P \rightsquigarrow_e P' \Longrightarrow P \rhd Q \rightsquigarrow_e P'$›
by (*simp add*: *Sliding-def*)
(*meson* τ*-trans-NdetL* τ*-trans-ev-trans ev-trans-DetL*)

lemma *tick-trans-SlidingL*: ‹$P \rightsquigarrow_{✓r} P' \Longrightarrow P \rhd Q \rightsquigarrow_{✓r} \Omega\ (SKIP\ r)\ r$›
by (*metis UnCI initials-Sliding initial-tick-imp-tick-trans-*Ω*-SKIP*)

lemma τ*-trans-SlidingR*: ‹$P \rhd Q \rightsquigarrow_\tau Q$›
by (*simp add*: *Sliding-def* τ*-trans-NdetR*)

lemma τ*-trans-SeqR*: ‹$P\ ;\ Q \rightsquigarrow_\tau Q'$› **if** ‹$P \rightsquigarrow_{✓r} P'$› **and** ‹$Q \rightsquigarrow_\tau Q'$›
proof −
from *that*(*1*) **have** ‹$P \sqsubseteq_{FD} SKIP\ r$› **by** (*simp add*: *initial-tick-iff-FD-SKIP*)
with *FD-iff-eq-Ndet* **have** ‹$P = P \sqcap SKIP\ r$› **..**
hence ‹$P\ ;\ Q = (P\ ;\ Q) \sqcap (SKIP\ r\ ;\ Q)$› **by** (*metis Seq-distrib-Ndet-right*)
also have ‹$\ldots = (P\ ;\ Q) \sqcap Q$› **by** *simp*
also have ‹$\ldots \rightsquigarrow_\tau Q$› **by** (*simp add*: τ*-trans-NdetR*)
finally show ‹$P\ ;\ Q \rightsquigarrow_\tau Q'$› **by** (*rule* τ*-trans-transitivity*) (*fact that*(*2*))
qed

lemma ‹$✓(r) \in P^0 \Longrightarrow Q \rightsquigarrow_e Q' \Longrightarrow P\ ;\ Q \rightsquigarrow_e Q'$›
using τ*-trans-SeqR* τ*-trans-eq* τ*-trans-ev-trans* **by** *blast*

lemma *tick-trans-Hiding*: ‹$P \rightsquigarrow_{✓r} P' \Longrightarrow P \setminus B \rightsquigarrow_{✓r} \Omega\ (SKIP\ r)\ r$›
by (*simp add*: *initial-tick-imp-tick-trans-*Ω*-SKIP initial-tick-imp-initial-tick-Hiding*)

lemma τ*-trans-SKIP-SyncL*: ‹$P \llbracket S \rrbracket Q \rightsquigarrow_\tau SKIP\ r \llbracket S \rrbracket Q$› **if** ‹$P \rightsquigarrow_{✓r} P'$›
proof −
from *that* **have** ‹$P \sqsubseteq_{FD} SKIP\ r$› **by** (*simp add*: *initial-tick-iff-FD-SKIP*)
with *FD-iff-eq-Ndet* **have** ‹$P = P \sqcap SKIP\ r$› **..**
hence ‹$P \llbracket S \rrbracket Q = (P \llbracket S \rrbracket Q) \sqcap (SKIP\ r \llbracket S \rrbracket Q)$› **by** (*metis Sync-distrib-Ndet-right*)
also have ‹$\ldots \rightsquigarrow_\tau (SKIP\ r \llbracket S \rrbracket Q)$› **by** (*rule* τ*-trans-NdetR*)

finally show ‹P ⟦S⟧ $Q \rightsquigarrow_\tau$ $SKIP\ r$ ⟦S⟧ Q› **.**
qed

lemma τ*-trans-SKIP-SyncR*: ‹$Q \rightsquigarrow_{\checkmark r} Q' \Longrightarrow P$ ⟦S⟧ $Q \rightsquigarrow_\tau P$ ⟦S⟧ $SKIP\ r$›
by (*metis Sync-commute* τ*-trans-SKIP-SyncL*)

lemma *tick-trans-SKIP-Sync-SKIP*: ‹$SKIP\ r$ ⟦S⟧ $SKIP\ r \rightsquigarrow_{\checkmark r} \Omega\ (SKIP\ r)\ r$›
by (*simp add*: *SKIP-trans-tick-*Ω*-SKIP*)

lemma ‹$SKIP\ r$ ⟦S⟧ $SKIP\ r \rightsquigarrow_\tau SKIP\ r$›
by (*simp add*: τ*-trans-eq*)

lemma *tick-trans-InterruptL* : ‹$P \rightsquigarrow_{\checkmark r} P' \Longrightarrow P \triangle Q \rightsquigarrow_{\checkmark r} \Omega\ (SKIP\ r)\ r$›
and *tick-trans-InterruptR* : ‹$Q \rightsquigarrow_{\checkmark r} Q' \Longrightarrow P \triangle Q \rightsquigarrow_{\checkmark r} \Omega\ (SKIP\ r)\ r$›
by (*rule initial-tick-imp-tick-trans-*Ω*-SKIP*, *simp add*: *initials-Interrupt*)+

lemma *tick-trans-ThrowL* : ‹$P \rightsquigarrow_{\checkmark r} P' \Longrightarrow P\ \Theta\ a \in A.\ Q\ a \rightsquigarrow_{\checkmark r} \Omega\ (SKIP\ r)\ r$›
by (*rule initial-tick-imp-tick-trans-*Ω*-SKIP*)
(*simp add*: *initials-Throw*)

lemma *ev-trans-ThrowR-inside*:
‹$e \in A \Longrightarrow P \rightsquigarrow_e P' \Longrightarrow P\ \Theta\ a \in A.\ Q\ a \rightsquigarrow_e (Q\ e)$›
by (*simp add*: $After_{tick}$*-Throw initials-Throw BOT-*τ*-trans-anything* τ*-trans-eq*)

end

6.6 Recovering other operational rules

By adding a τ-transition hypothesis on each operator, we can recover the remaining operational rules.

6.6.1 *Det* Laws

locale *OpSemTransitionsDet* = *OpSemTransitions* $\Psi\ \Omega$ ‹($\rightsquigarrow_\tau$)›
for Ψ :: ‹[('a, 'r) $process_{ptick}$, 'a] ⇒ ('a, 'r) $process_{ptick}$›
and Ω :: ‹[('a, 'r) $process_{ptick}$, 'r] ⇒ ('a, 'r) $process_{ptick}$›
and τ*-trans* :: ‹[('a, 'r) $process_{ptick}$, ('a, 'r) $process_{ptick}$] ⇒ *bool*› (**infixl** ‹$\rightsquigarrow_\tau$› *50*) +
assumes τ*-trans-DetL* : ‹$P \rightsquigarrow_\tau P' \Longrightarrow P \Box Q \rightsquigarrow_\tau P' \Box Q$›
begin

lemma τ*-trans-DetR* : ‹$Q \rightsquigarrow_\tau Q' \Longrightarrow P \Box Q \rightsquigarrow_\tau P \Box Q'$›
by (*metis Det-commute* τ*-trans-DetL*)

lemmas *Det-OpSem-rules* = τ*-trans-DetL* τ*-trans-DetR*

ev-trans-DetL ev-trans-DetR
tick-trans-DetL tick-trans-DetR

end

6.6.2 *Det* relaxed Laws

locale *OpSemTransitionsDetRelaxed* = *OpSemTransitions* Ψ Ω ‹$(\rightsquigarrow_\tau)$›
for Ψ :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
and Ω :: ‹$[('a, 'r)\ process_{ptick}, 'r] \Rightarrow ('a, 'r)\ process_{ptick}$›
and τ-*trans* :: ‹$[('a, 'r)\ process_{ptick}, ('a, 'r)\ process_{ptick}] \Rightarrow bool$› (**infixl** ‹$\rightsquigarrow_\tau$› *50*) +
assumes τ-*trans-DetL* : ‹$P = \bot \vee P' \neq \bot \vee Q = \bot \Longrightarrow P \rightsquigarrow_\tau P' \Longrightarrow P \Box Q \rightsquigarrow_\tau P' \Box Q$›
begin

lemma τ-*trans-DetR* : ‹$Q = \bot \vee Q' \neq \bot \vee Q = \bot \Longrightarrow Q \rightsquigarrow_\tau Q' \Longrightarrow P \Box Q \rightsquigarrow_\tau P \Box Q'$›
by (*metis Det-commute* τ-*trans-DetL*)

lemmas *Det-OpSem-rules* = τ-*trans-DetL* τ-*trans-DetR*
ev-trans-DetL ev-trans-DetR
tick-trans-DetL tick-trans-DetR

end

6.6.3 *Seq* Laws

locale *OpSemTransitionsSeq* = *OpSemTransitions* Ψ Ω ‹$(\rightsquigarrow_\tau)$›
for Ψ :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
and Ω :: ‹$[('a, 'r)\ process_{ptick}, 'r] \Rightarrow ('a, 'r)\ process_{ptick}$›
and τ-*trans* :: ‹$[('a, 'r)\ process_{ptick}, ('a, 'r)\ process_{ptick}] \Rightarrow bool$› (**infixl** ‹$\rightsquigarrow_\tau$› *50*) +
assumes τ-*trans-SeqL* : ‹$P \rightsquigarrow_\tau P' \Longrightarrow P ; Q \rightsquigarrow_\tau P' ; Q$›
begin

lemma *ev-trans-SeqL*: ‹$P \rightsquigarrow_e P' \Longrightarrow P ; Q \rightsquigarrow_e P' ; Q$›
apply (*cases* ‹$P = \bot$›, *solves* ‹*simp add*: *BOT-ev-trans-anything*›)
apply (*simp add*: *After*$_{tick}$-*Seq initials-Seq image-iff disjoint-iff*)
by (*meson* τ-*trans-NdetL* τ-*trans-SeqL* τ-*trans-transitivity ev-trans-is*)

lemmas *Seq-OpSem-rules* = τ-*trans-SeqL ev-trans-SeqL* τ-*trans-SeqR*

end

6.6.4 *Renaming* Laws

We are used to it now: we need to duplicate the locale in order to obtain the rules for the *Renaming* operator.

locale *OpSemTransitionsDuplicated* =
  $OpSemTransitions_\alpha$: *OpSemTransitions* Ψ_α Ω_α ‹$(_\alpha\rightsquigarrow_\tau)$› +
  $OpSemTransitions_\beta$: *OpSemTransitions* Ψ_β Ω_β ‹$(_\beta\rightsquigarrow_\tau)$›
  for Ψ_α :: ‹[('a, 'r) $process_{ptick}$, 'a] ⇒ ('a, 'r) $process_{ptick}$›
    and Ω_α :: ‹[('a, 'r) $process_{ptick}$, 'r] ⇒ ('a, 'r) $process_{ptick}$›
     and τ-$trans_\alpha$ :: ‹[('a, 'r) $process_{ptick}$, ('a, 'r) $process_{ptick}$] ⇒ *bool*› (**infixl** ‹$_\alpha\rightsquigarrow_\tau$› *50*)
    and Ψ_β :: ‹[('b, 's) $process_{ptick}$, 'b] ⇒ ('b, 's) $process_{ptick}$›
    and Ω_β :: ‹[('b, 's) $process_{ptick}$, 's] ⇒ ('b, 's) $process_{ptick}$›
   and τ-$trans_\beta$ :: ‹[('b, 's) $process_{ptick}$, ('b, 's) $process_{ptick}$] ⇒ *bool*› (**infixl** ‹$_\beta\rightsquigarrow_\tau$› *50*)
begin

notation $OpSemTransitions_\alpha$.*ev-trans* (‹- $_\alpha\rightsquigarrow$_ -› [*50*, *3*, *51*] *50*)
notation $OpSemTransitions_\alpha$.*tick-trans* (‹- $_\alpha\rightsquigarrow\checkmark$_ -› [*50*, *3*, *51*] *50*)
notation $OpSemTransitions_\beta$.*ev-trans* (‹- $_\beta\rightsquigarrow$_ -› [*50*, *3*, *51*] *50*)
notation $OpSemTransitions_\beta$.*tick-trans* (‹- $_\beta\rightsquigarrow\checkmark$_ -› [*50*, *3*, *51*] *50*)

lemma *tick-trans-Renaming*: ‹*P* $_\alpha\rightsquigarrow\checkmark r$ *P'* ⟹ *Renaming P f g* $_\beta\rightsquigarrow\checkmark g$ *r* (Ω_β (*SKIP* (*g r*)) (*g r*))›
  by (*metis* $OpSemTransitions_\beta$.*initial-tick-imp-tick-trans-Ω-SKIP* *Renaming-SKIP mono-Renaming-FD initial-tick-iff-FD-SKIP*)

end

sublocale *OpSemTransitionsDuplicated* ⊆ *AfterExtDuplicated* .
  — Recovering *AfterExtDuplicated.*$After_{tick}$*-Renaming* (and *AfterDuplicated.After-Renaming*)

locale *OpSemTransitionsRenaming* =
  OpSemTransitionsDuplicated Ψ_α Ω_α τ-$trans_\alpha$ Ψ_β Ω_β τ-$trans_\beta$
  for Ψ_α :: ‹[('a, 'r) $process_{ptick}$, 'a] ⇒ ('a, 'r) $process_{ptick}$›
    and Ω_α :: ‹[('a, 'r) $process_{ptick}$, 'r] ⇒ ('a, 'r) $process_{ptick}$›
     and τ-$trans_\alpha$ :: ‹[('a, 'r) $process_{ptick}$, ('a, 'r) $process_{ptick}$] ⇒ *bool*› (**infixl** ‹$_\alpha\rightsquigarrow_\tau$› *50*)
    and Ψ_β :: ‹[('b, 's) $process_{ptick}$, 'b] ⇒ ('b, 's) $process_{ptick}$›
    and Ω_β :: ‹[('b, 's) $process_{ptick}$, 's] ⇒ ('b, 's) $process_{ptick}$›
   and τ-$trans_\beta$ :: ‹[('b, 's) $process_{ptick}$, ('b, 's) $process_{ptick}$] ⇒ *bool*› (**infixl** ‹$_\beta\rightsquigarrow_\tau$› *50*) +
  assumes τ-*trans-Renaming* : ‹*P* $_\alpha\rightsquigarrow_\tau$ *P'* ⟹ *Renaming P f g* $_\beta\rightsquigarrow_\tau$ *Renaming P' f g*›
begin

lemma *ev-trans-Renaming*: ‹*Renaming P f g* $_\beta\rightsquigarrow_b$ (*Renaming P' f g*)›
  if ‹*f a* = *b*› **and** ‹*P* $_\alpha\rightsquigarrow_a$ *P'*›

proof (*cases* ‹$P = \bot$›)
  show ‹$P = \bot \Longrightarrow$ *Renaming* $P\ f\ g\ {}_\beta\!\rightsquigarrow_b$ (*Renaming* $P'\ f\ g$)›
    by (*simp add*: $After_{tick}$*-Renaming*)
      (*simp add*: $OpSemTransitions_\beta$*.BOT-*τ*-trans-anything*
        $OpSemTransitions_\beta$*.BOT-ev-trans-anything*)
next
  assume *non-BOT*: ‹$P \neq \bot$›
  from *that* **have** *initial* : ‹$\exists\, a.$ *ev* $a \in$ *initials* $P \wedge f\ a = b$› **by** *blast*
  show ‹*Renaming* $P\ f\ g\ {}_\beta\!\rightsquigarrow_b$ (*Renaming* $P'\ f\ g$)›
  proof (*intro conjI*)
    from *initial* **show** ‹*ev* $b \in$ (*Renaming* $P\ f\ g)^0$›
      by (*force simp add*: *initials-Renaming image-iff*)
  next
    show ‹$OpSemTransitions_\beta.After_{tick}$ (*Renaming* $P\ f\ g$) (*ev* b) ${}_\beta\!\rightsquigarrow_\tau$ *Renaming* $P'\ f\ g$›
      apply (*simp add*: $After_{tick}$*-Renaming non-BOT initial*)
      apply (*rule* $OpSemTransitions_\beta.\tau$*-trans-transitivity*
          [*OF* $OpSemTransitions_\beta.\tau$*-trans-GlobalNdet*[*of a*] τ*-trans-Renaming*])
       apply (*solves* ‹*simp add*: *that*›)
      using $OpSemTransitions_\alpha$*.ev-trans-is that*(*2*) **by** *blast*
  qed
qed

lemmas *Renaming-OpSem-rules* = τ*-trans-Renaming tick-trans-Renaming ev-trans-Renaming*

end

6.6.5 *Hiding* Laws

locale *OpSemTransitionsHiding* = *OpSemTransitions* $\Psi\ \Omega$ ‹$(\rightsquigarrow_\tau)$›
  for Ψ :: ‹[('a, 'r) $process_{ptick}$, 'a] ⇒ ('a, 'r) $process_{ptick}$›
    and Ω :: ‹[('a, 'r) $process_{ptick}$, 'r] ⇒ ('a, 'r) $process_{ptick}$›
    and τ*-trans* :: ‹[('a, 'r) $process_{ptick}$, ('a, 'r) $process_{ptick}$] ⇒ *bool*› (**infixl** ‹$\rightsquigarrow_\tau$›
50) +
  assumes τ*-trans-Hiding* : ‹$P \rightsquigarrow_\tau P' \Longrightarrow P \setminus A \rightsquigarrow_\tau P' \setminus A$›
begin

lemma τ*-trans-Hiding-inside*: ‹$P \setminus A \rightsquigarrow_\tau P' \setminus A$› **if** ‹$e \in A$› **and** ‹$P \rightsquigarrow_e P'$›
proof −
  have ‹$P \setminus A \sqsubseteq_{FD} P$ *after*$_\checkmark$ *ev* $e \setminus A$›
    by (*simp add*: *Hiding-FD-Hiding-*$After_{tick}$*-if-initial-inside that*)
  with *FD-iff-eq-Ndet* **have** ‹$P \setminus A = (P \setminus A) \sqcap (P$ *after*$_\checkmark$ *ev* $e \setminus A)$› **..**
  moreover have ‹$\ldots \rightsquigarrow_\tau P$ *after*$_\checkmark$ *ev* $e \setminus A$› **by** (*simp add*: τ*-trans-NdetR*)
  moreover have ‹$\ldots \rightsquigarrow_\tau P' \setminus A$› **by** (*simp add*: τ*-trans-Hiding that*(*2*))
  ultimately show ‹$P \setminus A \rightsquigarrow_\tau P' \setminus A$› **by** (*metis* τ*-trans-transitivity*)
qed

lemma *ev-trans-Hiding-notin*: ‹$P \setminus A \rightsquigarrow_e P' \setminus A$› **if** ‹$e \notin A$› **and** ‹$P \rightsquigarrow_e P'$›

proof −
note *initial* = *initial-notin-imp-initial-Hiding*[*OF that*(*2*)[*THEN conjunct1*] *that*(*1*)]
have ‹(*P* \ *A*) *after*$_{\checkmark}$ *ev e* $\sqsubseteq_{FD}$ *P after*$_{\checkmark}$ *ev e* \ *A*›
by (*simp add*: *After*$_{tick}$-*Hiding-FD-Hiding-After*$_{tick}$-*if-initial-notin that*)
with *FD-iff-eq-Ndet*
have ‹(*P* \ *A*) *after*$_{\checkmark}$ *ev e* = (*P* \ *A*) *after*$_{\checkmark}$ *ev e* ⊓ (*P after*$_{\checkmark}$ *ev e* \ *A*)› **..**
moreover have ‹... $\rightsquigarrow_{\tau}$ *P after*$_{\checkmark}$ *ev e* \ *A*› **by** (*simp add*: *τ-trans-NdetR*)
moreover have ‹... $\rightsquigarrow_{\tau}$ *P*′ \ *A*› **by** (*simp add*: *τ-trans-Hiding that*(*2*))
ultimately show ‹*P* \ *A* $\rightsquigarrow_{e}$ *P*′ \ *A*› **by** (*metis* (*no-types*) *τ-trans-transitivity initial*)
qed

lemmas *Hiding-OpSem-rules* = *τ-trans-Hiding tick-trans-Hiding*
ev-trans-Hiding-notin τ-trans-Hiding-inside

end

6.6.6 *Sync* Laws

locale *OpSemTransitionsSync* = *OpSemTransitions* Ψ Ω ‹($\rightsquigarrow_{\tau}$)›
for Ψ :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*a*] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
and Ω :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*r*] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
and *τ-trans* :: ‹[('*a*, '*r*) *process*$_{ptick}$, ('*a*, '*r*) *process*$_{ptick}$] ⇒ *bool*› (**infixl** ‹$\rightsquigarrow_{\tau}$› *50*) +
assumes *τ-trans-SyncL* : ‹*P* $\rightsquigarrow_{\tau}$ *P*′ ⟹ *P* ⟦*S*⟧ *Q* $\rightsquigarrow_{\tau}$ *P*′ ⟦*S*⟧ *Q*›
begin

lemma *τ-trans-SyncR* : ‹*Q* $\rightsquigarrow_{\tau}$ *Q*′ ⟹ *P* ⟦*S*⟧ *Q* $\rightsquigarrow_{\tau}$ *P* ⟦*S*⟧ *Q*′›
by (*metis Sync-commute τ-trans-SyncL*)

lemma *ev-trans-SyncL* : ‹*e* ∉ *S* ⟹ *P* $\rightsquigarrow_{e}$ *P*′ ⟹ *P* ⟦*S*⟧ *Q* $\rightsquigarrow_{e}$ *P*′ ⟦*S*⟧ *Q*›
by (*simp add*: *After*$_{tick}$-*Sync initials-Sync BOT-τ-trans-anything image-iff*)
(*meson τ-trans-NdetL τ-trans-SyncL τ-trans-transitivity ev-trans-is*)

lemma *ev-trans-SyncR* : ‹*e* ∉ *S* ⟹ *Q* $\rightsquigarrow_{e}$ *Q*′ ⟹ *P* ⟦*S*⟧ *Q* $\rightsquigarrow_{e}$ *P* ⟦*S*⟧ *Q*′›
by (*metis Sync-commute ev-trans-SyncL*)

lemma *ev-trans-SyncLR* :
‹*e* ∈ *S* ⟹ *P* $\rightsquigarrow_{e}$ *P*′ ⟹ *Q* $\rightsquigarrow_{e}$ *Q*′ ⟹ *P* ⟦*S*⟧ *Q* $\rightsquigarrow_{e}$ *P*′ ⟦*S*⟧ *Q*′›
by (*simp add*: *After*$_{tick}$-*Sync BOT-τ-trans-anything initials-Sync*)
(*meson τ-trans-SyncL τ-trans-SyncR τ-trans-transitivity ev-trans-is*)

lemmas *Sync-OpSem-rules* = *τ-trans-SyncL τ-trans-SyncR*
ev-trans-SyncL ev-trans-SyncR
ev-trans-SyncLR
τ-trans-SKIP-SyncL τ-trans-SKIP-SyncR
tick-trans-SKIP-Sync-SKIP

end

6.6.7 *Sliding* Laws

locale *OpSemTransitionsSliding* = *OpSemTransitions* Ψ Ω ‹$(\rightsquigarrow_\tau)$›
  for Ψ :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
    and Ω :: ‹$[('a, 'r)\ process_{ptick}, 'r] \Rightarrow ('a, 'r)\ process_{ptick}$›
    and τ-*trans* :: ‹$[('a, 'r)\ process_{ptick}, ('a, 'r)\ process_{ptick}] \Rightarrow bool$› (**infixl** ‹$\rightsquigarrow_\tau$› *50*) +
  assumes τ-*trans-SlidingL* : ‹$P \rightsquigarrow_\tau P' \Longrightarrow P \rhd Q \rightsquigarrow_\tau P' \rhd Q$›
    — We just add the τ-*trans-SlidingL* property.
begin

lemmas *Sliding-OpSem-rules* = τ-*trans-SlidingR* τ-*trans-SlidingL*
  ev-trans-SlidingL tick-trans-SlidingL

end

6.6.8 *Sliding* relaxed Laws

locale *OpSemTransitionsSlidingRelaxed* = *OpSemTransitions* Ψ Ω ‹$(\rightsquigarrow_\tau)$›
  for Ψ :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
    and Ω :: ‹$[('a, 'r)\ process_{ptick}, 'r] \Rightarrow ('a, 'r)\ process_{ptick}$›
    and τ-*trans* :: ‹$[('a, 'r)\ process_{ptick}, ('a, 'r)\ process_{ptick}] \Rightarrow bool$› (**infixl** ‹$\rightsquigarrow_\tau$› *50*) +
  assumes τ-*trans-SlidingL* : ‹$P = \bot \vee P' \neq \bot \vee Q = \bot \Longrightarrow P \rightsquigarrow_\tau P' \Longrightarrow P \rhd Q \rightsquigarrow_\tau P' \rhd Q$›
    — We just add the τ-*trans-SlidingL* property.
begin

lemmas *Sliding-OpSem-rules* = τ-*trans-SlidingR* τ-*trans-SlidingL*
  ev-trans-SlidingL tick-trans-SlidingL

end

6.6.9 *Interrupt* Laws

locale *OpSemTransitionsInterruptL* = *OpSemTransitions* Ψ Ω ‹$(\rightsquigarrow_\tau)$›
  for Ψ :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
    and Ω :: ‹$[('a, 'r)\ process_{ptick}, 'r] \Rightarrow ('a, 'r)\ process_{ptick}$›
    and τ-*trans* :: ‹$[('a, 'r)\ process_{ptick}, ('a, 'r)\ process_{ptick}] \Rightarrow bool$› (**infixl** ‹$\rightsquigarrow_\tau$› *50*) +
  assumes τ-*trans-InterruptL* : ‹$P \rightsquigarrow_\tau P' \Longrightarrow P \triangle Q \rightsquigarrow_\tau P' \triangle Q$ ›
begin

lemma *ev-trans-InterruptL*: ‹$P \rightsquigarrow_e P' \Longrightarrow P \triangle Q \rightsquigarrow_e P' \triangle Q$›
  apply (*simp add*: $After_{tick}$-*Interrupt initials-Interrupt*)
  using τ-*trans-InterruptL* τ-*trans-NdetR* τ-*trans-transitivity* **by** *blast*

lemma *ev-trans-InterruptR*: ‹$Q \rightsquigarrow_e Q' \Longrightarrow P \triangle Q \rightsquigarrow_e Q'$›
apply (*simp add*: $After_{tick}$*-Interrupt initials-Interrupt*)
using τ*-trans-NdetL* τ*-trans-transitivity* **by** *blast*

end

locale *OpSemTransitionsInterrupt* = *OpSemTransitionsInterruptL* Ψ Ω ‹$(\rightsquigarrow_\tau)$›
for Ψ :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
and Ω :: ‹$[('a, 'r)\ process_{ptick}, 'r] \Rightarrow ('a, 'r)\ process_{ptick}$›
and τ*-trans* :: ‹$[('a, 'r)\ process_{ptick}, ('a, 'r)\ process_{ptick}] \Rightarrow bool$› (**infixl** ‹$\rightsquigarrow_\tau$› 50) +
assumes τ*-trans-InterruptR* : ‹$Q \rightsquigarrow_\tau Q' \Longrightarrow P \triangle Q \rightsquigarrow_\tau P \triangle Q'$›
— We just add the τ*-trans-InterruptR* property.
begin

lemmas *Interrupt-OpSem-rules* = τ*-trans-InterruptL* τ*-trans-InterruptR*
ev-trans-InterruptL ev-trans-InterruptR
tick-trans-InterruptL tick-trans-InterruptR

end

6.6.10 *Throw* Laws

locale *OpSemTransitionsThrow* = *OpSemTransitions* Ψ Ω ‹$(\rightsquigarrow_\tau)$›
for Ψ :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
and Ω :: ‹$[('a, 'r)\ process_{ptick}, 'r] \Rightarrow ('a, 'r)\ process_{ptick}$›
and τ*-trans* :: ‹$[('a, 'r)\ process_{ptick}, ('a, 'r)\ process_{ptick}] \Rightarrow bool$› (**infixl** ‹$\rightsquigarrow_\tau$› 50) +
assumes τ*-trans-ThrowL* : ‹$P \rightsquigarrow_\tau P' \Longrightarrow P\ \Theta\ a \in A.\ Q\ a \rightsquigarrow_\tau P'\ \Theta\ a \in A.\ Q\ a$›
begin

lemma *ev-trans-ThrowL-notin*:
‹$e \notin A \Longrightarrow P \rightsquigarrow_e P' \Longrightarrow P\ \Theta\ a \in A.\ Q\ a \rightsquigarrow_e (P'\ \Theta\ a \in A.\ Q\ a)$›
by (*simp add*: $After_{tick}$*-Throw initials-Throw BOT-*τ*-trans-anything* τ*-trans-ThrowL*)

lemmas *Throw-OpSem-rules* = τ*-trans-ThrowL tick-trans-ThrowL*
ev-trans-ThrowL-notin ev-trans-ThrowR-inside

end

6.7 Locales, Assemble !

It is now time to assemble our locales.

locale *OpSemTransitionsAll* =
  OpSemTransitionsDet Ψ Ω ‹$(\rightsquigarrow_\tau)$› +
  OpSemTransitionsSeq Ψ Ω ‹$(\rightsquigarrow_\tau)$› +
  OpSemTransitionsHiding Ψ Ω ‹$(\rightsquigarrow_\tau)$› +
  OpSemTransitionsSync Ψ Ω ‹$(\rightsquigarrow_\tau)$› +
  OpSemTransitionsSliding Ψ Ω ‹$(\rightsquigarrow_\tau)$› +
  OpSemTransitionsInterrupt Ψ Ω ‹$(\rightsquigarrow_\tau)$› +
  OpSemTransitionsThrow Ψ Ω ‹$(\rightsquigarrow_\tau)$›
  for Ψ :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
    and Ω :: ‹$[('a, 'r)\ process_{ptick}, 'r] \Rightarrow ('a, 'r)\ process_{ptick}$›
    and τ-*trans* :: ‹$[('a, 'r)\ process_{ptick}, ('a, 'r)\ process_{ptick}] \Rightarrow bool$› (**infixl** ‹$\rightsquigarrow_\tau$› *50*)

Of course we need to duplicate the locale for obtaining *Renaming* rules.

locale *OpSemTransitionsAllDuplicated* =
  $OpSemTransitionsAll_\alpha$: *OpSemTransitionsAll* Ψ_α Ω_α ‹$({}_\alpha\rightsquigarrow_\tau)$› +
  $OpSemTransitionsAll_\beta$: *OpSemTransitionsAll* Ψ_β Ω_β ‹$({}_\beta\rightsquigarrow_\tau)$› +
  OpSemTransitionsRenaming Ψ_α Ω_α τ-$trans_\alpha$ Ψ_β Ω_β τ-$trans_\beta$
  for Ψ_α :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
    and Ω_α :: ‹$[('a, 'r)\ process_{ptick}, 'r] \Rightarrow ('a, 'r)\ process_{ptick}$›
     and τ-$trans_\alpha$ :: ‹$[('a, 'r)\ process_{ptick}, ('a, 'r)\ process_{ptick}] \Rightarrow bool$› (**infixl** ‹${}_\alpha\rightsquigarrow_\tau$› *50*)
    and Ψ_β :: ‹$[('b, 's)\ process_{ptick}, 'b] \Rightarrow ('b, 's)\ process_{ptick}$›
    and Ω_β :: ‹$[('b, 's)\ process_{ptick}, 's] \Rightarrow ('b, 's)\ process_{ptick}$›
   and τ-$trans_\beta$ :: ‹$[('b, 's)\ process_{ptick}, ('b, 's)\ process_{ptick}] \Rightarrow bool$› (**infixl** ‹${}_\beta\rightsquigarrow_\tau$› *50*)
begin

notation $OpSemTransitionsAll_\alpha$.*ev-trans* (‹- ${}_\alpha\rightsquigarrow$ - -› [*50*, *3*, *51*] *50*)
notation $OpSemTransitionsAll_\alpha$.*tick-trans* (‹- ${}_\alpha\rightsquigarrow_\checkmark$ - -› [*50*, *3*, *51*] *50*)
notation $OpSemTransitionsAll_\beta$.*ev-trans* (‹- ${}_\beta\rightsquigarrow$ - -› [*50*, *3*, *51*] *50*)
notation $OpSemTransitionsAll_\beta$.*tick-trans* (‹- ${}_\beta\rightsquigarrow_\checkmark$ - -› [*50*, *3*, *51*] *50*)

end

6.8 $(\rightsquigarrow_\tau)$ instantiated with $(\sqsubseteq_{FD})$ or $(\sqsubseteq_{DT})$

6.8.1 $(\rightsquigarrow_\tau)$ instantiated with $(\sqsubseteq_{FD})$

locale *OpSemFD* =
  fixes Ψ :: ‹$[('a, 'r)\ process_{ptick}, 'a] \Rightarrow ('a, 'r)\ process_{ptick}$›
    and Ω :: ‹$[('a, 'r)\ process_{ptick}, 'r] \Rightarrow ('a, 'r)\ process_{ptick}$›
  assumes *mono-*Ω*-FD*: ‹$\checkmark(r) \in Q^0 \Longrightarrow P \sqsubseteq_{FD} Q \Longrightarrow \Omega\ P\ r \sqsubseteq_{FD} \Omega\ Q\ r$›

sublocale *OpSemFD* $\subseteq$ *OpSemTransitionsAll* - - ‹$(\sqsubseteq_{FD})$ :: $('a, 'r)$ *process*$_{ptick}$ $\Rightarrow$ $('a, 'r)$ *process*$_{ptick}$ $\Rightarrow$ *bool*›
proof *unfold-locales*
show ‹$P \sqcap Q \sqsubseteq_{FD} P$› **for** $P\ Q$ :: ‹$('a, 'r)$ *process*$_{ptick}$› **by** (*fact Ndet-FD-self-left*)
next
show ‹$P \sqsubseteq_{FD} Q \Longrightarrow Q \sqsubseteq_{FD} R \Longrightarrow P \sqsubseteq_{FD} R$› **for** $P\ Q\ R$ :: ‹$('a, 'r)$ *process*$_{ptick}$› **by** (*fact trans-FD*)
next
show ‹$P \sqsubseteq_{FD} Q \Longrightarrow Q^0 \subseteq P^0$› **for** $P\ Q$ :: ‹$('a, 'r)$ *process*$_{ptick}$› **by** (*fact anti-mono-initials-FD*)
next
show ‹$e \in Q^0 \Longrightarrow P \sqsubseteq_{FD} Q \Longrightarrow$
AfterExt.After$_{tick}$ $\Psi\ \Omega\ P\ e \sqsubseteq_{FD}$ *AfterExt.After*$_{tick}$ $\Psi\ \Omega\ Q\ e$› **for** $e\ P\ Q$
by (*cases e*) (*simp-all add*: *AfterExt.After*$_{tick}$*-def After.mono-After-FD mono-*Ω*-FD*)
next
show ‹$P \sqsubseteq_{FD} P' \Longrightarrow P \,\Box\, Q \sqsubseteq_{FD} P' \,\Box\, Q$› **for** $P\ P'\ Q$ :: ‹$('a, 'r)$ *process*$_{ptick}$›
by (*intro mono-Det-FD idem-FD*)
next
show ‹$P \sqsubseteq_{FD} P' \Longrightarrow P \,;\, Q \sqsubseteq_{FD} P' \,;\, Q$› **for** $P\ P'\ Q$ :: ‹$('a, 'r)$ *process*$_{ptick}$›
by (*intro mono-Seq-FD idem-FD*)
next
show ‹$P \sqsubseteq_{FD} P' \Longrightarrow P \setminus A \sqsubseteq_{FD} P' \setminus A$› **for** $P\ P'$ :: ‹$('a, 'r)$ *process*$_{ptick}$› **and** A
by (*intro mono-Hiding-FD idem-FD*)
next
show ‹$P \sqsubseteq_{FD} P' \Longrightarrow P \,[\![S]\!]\, Q \sqsubseteq_{FD} P' \,[\![S]\!]\, Q$› **for** $P\ P'\ Q$ :: ‹$('a, 'r)$ *process*$_{ptick}$› **and** S
by (*intro mono-Sync-FD idem-FD*)
next
show ‹$P \sqsubseteq_{FD} P' \Longrightarrow P \rhd Q \sqsubseteq_{FD} P' \rhd Q$› **for** $P\ P'\ Q$ :: ‹$('a, 'r)$ *process*$_{ptick}$›
by (*intro mono-Sliding-FD idem-FD*)
next
show ‹$P \sqsubseteq_{FD} P' \Longrightarrow P \,\triangle\, Q \sqsubseteq_{FD} P' \,\triangle\, Q$› **for** $P\ P'\ Q$ :: ‹$('a, 'r)$ *process*$_{ptick}$›
by (*intro mono-Interrupt-FD idem-FD*)
next
show ‹$Q \sqsubseteq_{FD} Q' \Longrightarrow P \,\triangle\, Q \sqsubseteq_{FD} P \,\triangle\, Q'$› **for** $P\ Q\ Q'$ :: ‹$('a, 'r)$ *process*$_{ptick}$›
by (*intro mono-Interrupt-FD idem-FD*)
next
show ‹$P \sqsubseteq_{FD} P' \Longrightarrow P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_{FD} P'\ \Theta\ a \in A.\ Q\ a$› **for** $P\ P'$ :: ‹$('a, 'r)$ *process*$_{ptick}$› **and** $A\ Q$
by (*intro mono-Throw-FD idem-FD*)
qed

context *OpSemFD*
begin

Finally, the only remaining hypothesis is $[\![\checkmark(?r) \in ?Q^0;\ ?P \sqsubseteq_{FD} ?Q]\!] \Longrightarrow$

$\Omega\ ?P\ ?r \sqsubseteq_{FD} \Omega\ ?Q\ ?r$ when we instantiate our locale with the failure-divergence refinement ($\sqsubseteq_{FD}$).

Of course, we can strengthen some previous results.

notation *failure-divergence-refine* (**infixl** ‹$_{FD}\rightsquigarrow_{\tau}$› 50)
notation *ev-trans* (‹- $_{FD}\rightsquigarrow$- -› [50, 3, 51] 50)
notation *tick-trans* (‹- $_{FD}\rightsquigarrow_{\checkmark}$- -› [50, 3, 51] 50)
notation *trace-trans* (‹- $_{FD}\rightsquigarrow^{*}$- -› [50, 3, 51] 50)

lemma *trace-trans-imp-F*: ‹$P\ {}_{FD}\rightsquigarrow^{*} s\ Q \Longrightarrow X \in \mathcal{R}\ Q \Longrightarrow (s, X) \in \mathcal{F}\ P$›
by (*rule trace-trans-imp-F-if-τ-trans-imp-leF*) (*simp add*: *leFD-imp-leF*)

lemma *tickFree-trace-trans-BOT-imp-D*: ‹*tickFree* $s \Longrightarrow P\ {}_{FD}\rightsquigarrow^{*} s \perp \Longrightarrow s \in \mathcal{D}\ P$›
by (*rule tickFree-trace-trans-BOT-imp-D-if-τ-trans-BOT-imp-eq-BOT-weak*) (*simp add*: *FD-antisym*)

lemma *F-trace-trans-reality-check*: ‹*tickFree* $s \Longrightarrow (s, X) \in \mathcal{F}\ P \longleftrightarrow (\exists\, Q.\ (P\ {}_{FD}\rightsquigarrow^{*} s\ Q) \wedge X \in \mathcal{R}\ Q)$›
by (*simp add*: *F-trace-trans-reality-check-weak leFD-imp-leF*)

lemma *D-trace-trans-reality-check*: ‹*tickFree* $s \Longrightarrow s \in \mathcal{D}\ P \longleftrightarrow P\ {}_{FD}\rightsquigarrow^{*} s \perp$›
by (*simp add*: *D-trace-trans-reality-check-weak FD-antisym*)

lemma *Ω-SKIP-is-STOP-imp-SKIP-trace-trans-iff*:
‹$\Omega\ (SKIP\ r)\ r = STOP \Longrightarrow (SKIP\ r\ {}_{FD}\rightsquigarrow^{*} s\ P) \longleftrightarrow s = [] \wedge P = SKIP\ r \vee s = [\checkmark(r)] \wedge P = STOP$›
by (*erule Ω-SKIP-is-STOP-imp-τ-trans-imp-leF-imp-SKIP-trace-trans-iff*)
(*simp add*: *leFD-imp-leF*)

lemmas *τ-trans-adm* = *le-FD-adm*

lemma *ev-trans-adm*[*simp*]:
‹$[\![cont\ (\lambda P.\ \Psi\ P\ e);\ cont\ u;\ monofun\ v]\!] \Longrightarrow adm\ (\lambda x.\ u\ x\ {}_{FD}\rightsquigarrow_{e}\ v\ x)$›
by *simp*

lemma *tick-trans-adm*[*simp*]:
‹$[\![cont\ (\lambda P.\ \Omega\ P\ r);\ cont\ u;\ monofun\ v]\!] \Longrightarrow adm\ (\lambda x.\ u\ x\ {}_{FD}\rightsquigarrow_{\checkmark r}\ v\ x)$›
by *simp*

lemma *trace-trans-adm*[*simp*]:
‹$[\![\forall x.\ ev\ x \in set\ s \longrightarrow cont\ (\lambda P.\ \Psi\ P\ x);$
$\forall r.\ \checkmark(r) \in set\ s \longrightarrow cont\ (\lambda P.\ \Omega\ P\ r);\ cont\ u;\ monofun\ v]\!]$
$\Longrightarrow adm\ (\lambda x.\ u\ x\ {}_{FD}\rightsquigarrow^{*} s\ (v\ x))$›
by *simp*

end

locale *OpSemFDDuplicated* =
  $OpSemFD_\alpha$: *OpSemFD* Ψ_α Ω_α + $OpSemFD_\beta$: *OpSemFD* Ψ_β Ω_β
  for Ψ_α :: ‹[('a, 'r) $process_{ptick}$, 'a] ⇒ ('a, 'r) $process_{ptick}$›
    and Ω_α :: ‹[('a, 'r) $process_{ptick}$, 'r] ⇒ ('a, 'r) $process_{ptick}$›
    and Ψ_β :: ‹[('b, 's) $process_{ptick}$, 'b] ⇒ ('b, 's) $process_{ptick}$›
    and Ω_β :: ‹[('b, 's) $process_{ptick}$, 's] ⇒ ('b, 's) $process_{ptick}$›

sublocale *OpSemFDDuplicated* ⊆ *OpSemTransitionsAllDuplicated* - - ‹($\sqsubseteq_{FD}$)› - - ‹($\sqsubseteq_{FD}$)›
  by (*unfold-locales*) (*simp add*: *mono-Renaming-FD*)

6.8.2 ($\rightsquigarrow_\tau$) instantiated with ($\sqsubseteq_{DT}$)

locale *OpSemDT* =
  fixes Ψ :: ‹[('a, 'r) $process_{ptick}$, 'a] ⇒ ('a, 'r) $process_{ptick}$›
    and Ω :: ‹[('a, 'r) $process_{ptick}$, 'r] ⇒ ('a, 'r) $process_{ptick}$›
  assumes *mono-Ω-DT*: ‹✓(r) ∈ Q^0 ⟹ P $\sqsubseteq_{DT}$ Q ⟹ Ω P r $\sqsubseteq_{DT}$ Ω Q r›

sublocale *OpSemDT* ⊆ *OpSemTransitionsAll* - - ‹($\sqsubseteq_{DT}$) :: ('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒ *bool*›
proof *unfold-locales*
 show ‹P ⊓ Q $\sqsubseteq_{DT}$ P› **for** P Q :: ‹('a, 'r) $process_{ptick}$› **by** (*fact Ndet-DT-self-left*)
next
 show ‹P $\sqsubseteq_{DT}$ Q ⟹ Q $\sqsubseteq_{DT}$ R ⟹ P$\sqsubseteq_{DT}$ R› **for** P Q R :: ‹('a, 'r) $process_{ptick}$›
by (*fact trans-DT*)
next
  show ‹P $\sqsubseteq_{DT}$ Q ⟹ Q^0 ⊆ P^0› **for** P Q :: ‹('a, 'r) $process_{ptick}$› **by** (*fact anti-mono-initials-DT*)
next
 show ‹e ∈ Q^0 ⟹ P $\sqsubseteq_{DT}$ Q ⟹
      $AfterExt.After_{tick}$ Ψ Ω P e $\sqsubseteq_{DT}$ $AfterExt.After_{tick}$ Ψ Ω Q e› **for** e P Q
  by (*cases e*) (*simp-all add*: $AfterExt.After_{tick}$-*def After.mono-After-DT mono-Ω-DT*)
qed (*simp-all add*: *mono-Det-DT mono-Seq-DT mono-Hiding-DT mono-Sync-DT mono-Sliding-DT mono-Interrupt-DT mono-Throw-DT*)

context *OpSemDT*
begin

Finally, the only remaining hypothesis is ⟦✓(?r) ∈ ?Q^0; ?P $\sqsubseteq_{DT}$?Q⟧ ⟹ Ω ?P ?r $\sqsubseteq_{DT}$ Ω ?Q ?r when we instantiate our locale with the failure-divergence refinement ($\sqsubseteq_{DT}$).

Of course, we can strengthen some previous results.

notation *trace-divergence-refine* (**infixl** ‹$_{DT}\rightsquigarrow_\tau$› 50)

notation *ev-trans* (‹- $_{DT}\!\rightsquigarrow$- -› [*50*, *3*, *51*] *50*)
notation *tick-trans* (‹- $_{DT}\!\rightsquigarrow_{\checkmark}$- -› [*50*, *3*, *51*] *50*)
notation *trace-trans* (‹- $_{DT}\!\rightsquigarrow^{*}$- -› [*50*, *3*, *51*] *50*)

lemma *tickFree-trace-trans-BOT-imp-D*: ‹*tickFree s* $\Longrightarrow$ *P* $_{DT}\!\rightsquigarrow^{*}$ *s* $\bot$ $\Longrightarrow$ *s* $\in$ $\mathcal{D}$ *P*›
by (*rule tickFree-trace-trans-BOT-imp-D-if-τ-trans-BOT-imp-eq-BOT-weak*)
(*meson BOT-iff-Nil-D divergence-refine-def leDT-imp-leD subsetD*)

lemma *D-trace-trans-reality-check*: ‹*tickFree s* $\Longrightarrow$ *s* $\in$ $\mathcal{D}$ *P* $\longleftrightarrow$ *P* $_{DT}\!\rightsquigarrow^{*}$ *s* $\bot$›
by (*simp add*: *D-trace-trans-reality-check-weak BOT-iff-Nil-D tickFree-trace-trans-BOT-imp-D trace-τ-trans*)

lemmas *τ-trans-adm* = *le-DT-adm*

lemma *ev-trans-adm*[*simp*]:
‹⟦*cont* (λ*P*. Ψ *P e*); *cont u*; *monofun v*⟧ $\Longrightarrow$ *adm* (λ*x*. *u x* $_{DT}\!\rightsquigarrow_{e}$ *v x*)›
by *simp*

lemma *tick-trans-adm*[*simp*]:
‹⟦*cont* (λ*P*. Ω *P r*); *cont u*; *monofun v*⟧ $\Longrightarrow$ *adm* (λ*x*. *u x* $_{DT}\!\rightsquigarrow_{\checkmark r}$ *v x*)›
by *simp*

lemma *trace-trans-adm*[*simp*]:
‹⟦∀ *x*. *ev x* $\in$ *set s* $\longrightarrow$ *cont* (λ*P*. Ψ *P x*);
∀ *r*. ✓(*r*) $\in$ *set s* $\longrightarrow$ *cont* (λ*P*. Ω *P r*); *cont u*; *monofun v*⟧
$\Longrightarrow$ *adm* (λ*x*. *u x* $_{DT}\!\rightsquigarrow^{*}$ *s* (*v x*))›
by *simp*

If we only look at the traces and the divergences, non-deterministic and deterministic choices are the same. Therefore we can obtain even stronger results for the operational rules.

lemma *τ-trans-Det-is-τ-trans-Ndet*: ‹*P* □ *Q* $_{DT}\!\rightsquigarrow_{\tau}$ *R* $\longleftrightarrow$ *P* ⊓ *Q* $_{DT}\!\rightsquigarrow_{\tau}$ *R*›
unfolding *trace-divergence-refine-def trace-refine-def divergence-refine-def*
by (*simp add*: *T-Det T-Ndet D-Det D-Ndet*)

lemma *τ-trans-Sliding-is-τ-trans-Ndet*: ‹*P* ▷ *Q* $_{DT}\!\rightsquigarrow_{\tau}$ *R* $\longleftrightarrow$ *P* ⊓ *Q* $_{DT}\!\rightsquigarrow_{\tau}$ *R*›
unfolding *Sliding-def* **by** (*metis Det-assoc Det-id τ-trans-Det-is-τ-trans-Ndet*)

end

locale *OpSemDTDuplicated* =
OpSemDT$_{\alpha}$: *OpSemDT* Ψ$_{\alpha}$ Ω$_{\alpha}$ + *OpSemDT*$_{\beta}$: *OpSemDT* Ψ$_{\beta}$ Ω$_{\beta}$
for Ψ$_{\alpha}$ :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*a*] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
and Ω$_{\alpha}$ :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*r*] ⇒ ('*a*, '*r*) *process*$_{ptick}$›

and Ψ_β :: ‹[('b, 's) $process_{ptick}$, 'b] ⇒ ('b, 's) $process_{ptick}$›
and Ω_β :: ‹[('b, 's) $process_{ptick}$, 's] ⇒ ('b, 's) $process_{ptick}$›

sublocale *OpSemDTDuplicated* ⊆ *OpSemTransitionsAllDuplicated* - - ‹($\sqsubseteq_{DT}$)› - - ‹($\sqsubseteq_{DT}$)›
by (*unfold-locales*) (*simp add*: *mono-Renaming-DT*)

6.9 ($\rightsquigarrow_\tau$) instantiated with ($\sqsubseteq_F$) or ($\sqsubseteq_T$)

We will only recover the rules for some operators.

6.9.1 ($\rightsquigarrow_\tau$) instantiated with ($\sqsubseteq_F$)

locale *OpSemF* =
fixes Ψ :: ‹[('a, 'r) $process_{ptick}$, 'a] ⇒ ('a, 'r) $process_{ptick}$›
and Ω :: ‹[('a, 'r) $process_{ptick}$, 'r] ⇒ ('a, 'r) $process_{ptick}$›
assumes *mono-Ω-F*: ‹✓(r) ∈ Q^0 ⟹ P $\sqsubseteq_F$ Q ⟹ Ω P r $\sqsubseteq_F$ Ω Q r›

sublocale *OpSemF* ⊆ *OpSemTransitionsHiding* - - ‹($\sqsubseteq_F$) :: ('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒ *bool*› +
OpSemTransitionsDetRelaxed - - ‹($\sqsubseteq_F$) :: ('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒ *bool*› +
OpSemTransitionsSlidingRelaxed - - ‹($\sqsubseteq_F$) :: ('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒ *bool*›
proof *unfold-locales*
show ‹P ⊓ Q $\sqsubseteq_F$ P› **for** P Q :: ‹('a, 'r) $process_{ptick}$› **by** (*fact Ndet-F-self-left*)
next
show ‹P $\sqsubseteq_F$ Q ⟹ Q $\sqsubseteq_F$ R ⟹ P$\sqsubseteq_F$ R› **for** P Q R :: ‹('a, 'r) $process_{ptick}$›
by (*fact trans-F*)
next
show ‹P $\sqsubseteq_F$ Q ⟹ Q^0 ⊆ P^0› **for** P Q :: ‹('a, 'r) $process_{ptick}$› **by** (*fact anti-mono-initials-F*)
next
show ‹e ∈ Q^0 ⟹ P $\sqsubseteq_F$ Q ⟹
$AfterExt.After_{tick}$ Ψ Ω P e $\sqsubseteq_F$ $AfterExt.After_{tick}$ Ψ Ω Q e› **for** e P Q
by (*cases e*) (*simp-all add*: $AfterExt.After_{tick}$-*def After.mono-After-F mono-Ω-F*)
qed (*simp-all add*: *non-BOT-mono-Det-left-F non-BOT-mono-Sliding-F mono-Hiding-F*)

context *OpSemF*
begin

notation *failure-refine* (**infixl** ‹$_F\rightsquigarrow_\tau$› *50*)
notation *ev-trans* (‹- $_F\rightsquigarrow_-$ -› [*50*, *3*, *51*] *50*)
notation *tick-trans* (‹- $_F\rightsquigarrow_✓$ -› [*50*, *3*, *51*] *50*)
notation *trace-trans* (‹- $_F\rightsquigarrow^*$ -› [*50*, *3*, *51*] *50*)

For *Det* and *Sliding*, we have relaxed versions on τ transitions.

end

By duplicating the locale, we can recover a rules for *Renaming*.

locale *OpSemFDuplicated* =
  $OpSemF_\alpha$: *OpSemF* Ψ_α Ω_α + $OpSemF_\beta$: *OpSemF* Ψ_β Ω_β
  for Ψ_α :: ‹[('a, 'r) $process_{ptick}$, 'a] ⇒ ('a, 'r) $process_{ptick}$›
    and Ω_α :: ‹[('a, 'r) $process_{ptick}$, 'r] ⇒ ('a, 'r) $process_{ptick}$›
    and Ψ_β :: ‹[('b, 's) $process_{ptick}$, 'b] ⇒ ('b, 's) $process_{ptick}$›
    and Ω_β :: ‹[('b, 's) $process_{ptick}$, 's] ⇒ ('b, 's) $process_{ptick}$›

sublocale *OpSemFDuplicated* ⊆ *OpSemTransitionsDuplicated* - - ‹($\sqsubseteq_F$)› - - ‹($\sqsubseteq_F$)›
  by *unfold-locales*

context *OpSemFDuplicated*
begin

notation $OpSemF_\alpha$.*ev-trans* (‹- $_{\alpha F}\!\rightsquigarrow$ - -› [50, 3, 51] 50)
notation $OpSemF_\alpha$.*tick-trans* (‹- $_{\alpha F}\!\rightsquigarrow_{\checkmark}$ - -› [50, 3, 51] 50)
notation $OpSemF_\beta$.*ev-trans* (‹- $_{\beta F}\!\rightsquigarrow$ - -› [50, 3, 51] 50)
notation $OpSemF_\beta$.*tick-trans* (‹- $_{\beta F}\!\rightsquigarrow_{\checkmark}$ - -› [50, 3, 51] 50)

end

context *OpSemF*
begin

lemma *trace-trans-imp-F*: ‹P $_F\!\rightsquigarrow^* s$ $Q \Longrightarrow X \in \mathcal{R}$ $Q \Longrightarrow (s, X) \in \mathcal{F}$ P›
  by (*rule trace-trans-imp-F-if-τ-trans-imp-leF*) *simp*

lemma Ω*-SKIP-is-STOP-imp-SKIP-trace-trans-iff*:
  ‹Ω (*SKIP* r) r = *STOP* ⟹ (*SKIP* r $_F\!\rightsquigarrow^* s$ P) ⟷ s = [] ∧ P = *SKIP* r ∨ s = [✓(r)] ∧ P = *STOP*›
  by (*erule* Ω*-SKIP-is-STOP-imp-τ-trans-imp-leF-imp-SKIP-trace-trans-iff*) *simp*

lemmas *τ-trans-adm* = *le-F-adm*

lemma *ev-trans-adm*[*simp*]:
  ‹⟦*cont* (λP. Ψ P e); *cont* u; *monofun* v⟧ ⟹ *adm* (λx. u x $_F\!\rightsquigarrow_e$ v x)›
  by *simp*

lemma *tick-trans-adm*[*simp*]:
  ‹⟦*cont* (λP. Ω P r); *cont* u; *monofun* v⟧ ⟹ *adm* (λx. u x $_F\!\rightsquigarrow_{\checkmark r}$ v x)›
  by *simp*

lemma *trace-trans-adm*[*simp*]:
‹⟦∀ *x*. *ev x* ∈ *set s* ⟶ *cont* (λ*P*. Ψ *P x*);
∀ *r*. ✓(*r*) ∈ *set s* ⟶ *cont* (λ*P*. Ω *P r*);
cont u; *monofun v*⟧ ⟹ *adm* (λ*x*. *u x* $_F\rightsquigarrow^*$ *s* (*v x*))›
by *simp*

end

6.9.2 $(\rightsquigarrow_\tau)$ instantiated with $(\sqsubseteq_T)$

locale *OpSemTransitionsForT* =
OpSemTransitionsDet Ψ Ω ‹$(\rightsquigarrow_\tau)$› +
OpSemTransitionsHiding Ψ Ω ‹$(\rightsquigarrow_\tau)$› +
OpSemTransitionsSliding Ψ Ω ‹$(\rightsquigarrow_\tau)$› +
OpSemTransitionsInterrupt Ψ Ω ‹$(\rightsquigarrow_\tau)$›
for Ψ :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*a*] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
and Ω :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*r*] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
and *τ-trans* :: ‹[('*a*, '*r*) *process*$_{ptick}$, ('*a*, '*r*) *process*$_{ptick}$] ⇒ *bool*› (**infixl** ‹$\rightsquigarrow_\tau$› *50*)

locale *OpSemT* =
fixes Ψ :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*a*] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
and Ω :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*r*] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
assumes *mono-Ω-T*: ‹✓(*r*) ∈ Q^0 ⟹ *P* $\sqsubseteq_T$ *Q* ⟹ Ω *P r* $\sqsubseteq_T$ Ω *Q r*›

sublocale *OpSemT* ⊆ *OpSemTransitionsForT* - - ‹$(\sqsubseteq_T)$ :: ('*a*, '*r*) *process*$_{ptick}$ ⇒ ('*a*, '*r*) *process*$_{ptick}$ ⇒ *bool*›
proof *unfold-locales*
show ‹*P* ⊓ *Q* $\sqsubseteq_T$ *P*› **for** *P Q* :: ‹('*a*, '*r*) *process*$_{ptick}$› **by** (*fact Ndet-T-self-left*)
next
show ‹*P* $\sqsubseteq_T$ *Q* ⟹ *Q* $\sqsubseteq_T$ *R* ⟹ *P*$\sqsubseteq_T$ *R*› **for** *P Q R* :: ‹('*a*, '*r*) *process*$_{ptick}$›
by (*fact trans-T*)
next
show ‹*P* $\sqsubseteq_T$ *Q* ⟹ Q^0 ⊆ P^0› **for** *P Q* :: ‹('*a*, '*r*) *process*$_{ptick}$› **by** (*fact anti-mono-initials-T*)
next
show ‹*e* ∈ Q^0 ⟹ *P* $\sqsubseteq_T$ *Q* ⟹
AfterExt.After$_{tick}$ Ψ Ω *P e* $\sqsubseteq_T$ *AfterExt.After*$_{tick}$ Ψ Ω *Q e*› **for** *e P Q*
by (*cases e*) (*simp-all add*: *AfterExt.After*$_{tick}$*-def After.mono-After-T mono-Ω-T*)
qed (*simp-all add*: *mono-Det-T mono-Sliding-T mono-Hiding-T mono-Interrupt-T*)

context *OpSemT*
begin

notation *trace-refine* (**infixl** ‹$_T\rightsquigarrow_\tau$› *50*)
notation *ev-trans* (‹- $_T\rightsquigarrow_-$ -› [*50*, *3*, *51*] *50*)
notation *tick-trans* (‹- $_T\rightsquigarrow_✓$ -› [*50*, *3*, *51*] *50*)
notation *trace-trans* (‹- $_T\rightsquigarrow^*$ -› [*50*, *3*, *51*] *50*)

end

By duplicating the locale, we can recover a rules for *Renaming*.

locale *OpSemTDuplicated* =
  $OpSemT_\alpha$: *OpSemT* Ψ_α Ω_α + $OpSemT_\beta$: *OpSemT* Ψ_β Ω_β
  for Ψ_α :: ‹[('a, 'r) $process_{ptick}$, 'a] ⇒ ('a, 'r) $process_{ptick}$›
    and Ω_α :: ‹[('a, 'r) $process_{ptick}$, 'r] ⇒ ('a, 'r) $process_{ptick}$›
    and Ψ_β :: ‹[('b, 's) $process_{ptick}$, 'b] ⇒ ('b, 's) $process_{ptick}$›
    and Ω_β :: ‹[('b, 's) $process_{ptick}$, 's] ⇒ ('b, 's) $process_{ptick}$›

sublocale *OpSemTDuplicated* ⊆ *OpSemTransitionsDuplicated* - - ‹($\sqsubseteq_T$)› - - ‹($\sqsubseteq_T$)›
  by *unfold-locales*

context *OpSemTDuplicated*
begin

notation $OpSemT_\alpha$.*ev-trans* (‹- $_{\alpha T}\!\rightsquigarrow$ -› [50, 3, 51] 50)
notation $OpSemT_\alpha$.*tick-trans* (‹- $_{\alpha T}\!\rightsquigarrow\!\checkmark$ -› [50, 3, 51] 50)
notation $OpSemT_\beta$.*ev-trans* (‹- $_{\beta T}\!\rightsquigarrow$ -› [50, 3, 51] 50)
notation $OpSemT_\beta$.*tick-trans* (‹- $_{\beta T}\!\rightsquigarrow\!\checkmark$ -› [50, 3, 51] 50)

end

context *OpSemT*
begin

lemmas *τ-trans-adm* = *le-T-adm*

lemma *ev-trans-adm*[*simp*]:
  ‹⟦*cont* (λP. Ψ P e); *cont u*; *monofun v*⟧ ⟹ *adm* (λx. u x $_T\!\rightsquigarrow_e$ v x)›
  by *simp*

lemma *tick-trans-adm*[*simp*]:
  ‹⟦*cont* (λP. Ω P r); *cont u*; *monofun v*⟧ ⟹ *adm* (λx. u x $_T\!\rightsquigarrow\!\checkmark_r$ v x)›
  by *simp*

lemma *trace-trans-adm*[*simp*]:
  ‹⟦∀ x. *ev* x ∈ *set s* ⟶ *cont* (λP. Ψ P x);
    ∀ r. ✓(r) ∈ *set s* ⟶ *cont* (λP. Ω P r); *cont u*; *monofun v*⟧
   ⟹ *adm* (λx. u x $_T\!\rightsquigarrow^*$ s (v x))›
  by *simp*

If we only look at the traces, non-deterministic and deterministic choices are the same. Therefore we can obtain even stronger results for the operational rules.

lemma *τ-trans-Det-is-τ-trans-Ndet*: ‹P □ Q $_T\!\rightsquigarrow_\tau$ R ⟷ P ⊓ Q $_T\!\rightsquigarrow_\tau$ R›
  unfolding *trace-divergence-refine-def trace-refine-def divergence-refine-def*

by (*simp add*: *T-Det T-Ndet D-Det D-Ndet*)

lemma *τ-trans-Sliding-is-τ-trans-Ndet*: ‹$P \rhd Q \ {}_T\rightsquigarrow_\tau \ R \longleftrightarrow P \sqcap Q \ {}_T\rightsquigarrow_\tau \ R$›
unfolding *Sliding-def* **by** (*metis Det-assoc Det-id τ-trans-Det-is-τ-trans-Ndet*)

end

Chapter 7

Recovered Laws pretty printed

7.1 General Case

This is the general case, working for ($\sqsubseteq_{FD}$) and ($\sqsubseteq_{DT}$).

context *OpSemTransitionsAll* **begin**

Absorbency rules

$$\frac{?P \rightsquigarrow_{?e} ?P' \qquad ?P' \rightsquigarrow_{\tau} ?P''}{?P \rightsquigarrow_{?e} ?P''} \qquad \frac{?P \rightsquigarrow_{\tau} ?P' \qquad ?P' \rightsquigarrow_{?e} ?P''}{?P \rightsquigarrow_{?e} ?P''}$$

$$\frac{?P \rightsquigarrow_{\checkmark ?r} ?P' \qquad ?P' \rightsquigarrow_{\tau} ?P''}{?P \rightsquigarrow_{\checkmark ?r} ?P''} \qquad \frac{?P \rightsquigarrow_{\tau} ?P' \qquad ?P' \rightsquigarrow_{\checkmark ?r} ?P''}{?P \rightsquigarrow_{\checkmark ?r} ?P''}$$

SKIP **rule**

$$\frac{}{SKIP\ ?r \rightsquigarrow_{\checkmark ?r} \Omega\ (SKIP\ ?r)\ ?r}$$

$e \rightarrow P$ **rules**

$$\frac{}{?e \rightarrow ?P \rightsquigarrow_{?e} ?P}$$

$$\frac{?e \in ?A}{\Box a{\in}?A \rightarrow ?P\ a \rightsquigarrow_{?e} ?P\ ?e} \qquad \frac{?e \in ?A}{\sqcap a{\in}?A \rightarrow ?P\ a \rightsquigarrow_{?e} ?P\ ?e}$$

($\sqcap$) **rules**

$$\frac{}{?P \sqcap ?Q \rightsquigarrow_\tau ?P} \qquad \frac{}{?P \sqcap ?Q \rightsquigarrow_\tau ?Q}$$

$$\frac{?e \in ?A}{\sqcap a{\in}?A.\ ?P\ a \rightsquigarrow_\tau ?P\ ?e}$$

$\mu\ x.\ f\ x$ **rule**

$$\frac{cont\ ?f \qquad ?P = (\mu\ x.\ ?f\ x)}{?P \rightsquigarrow_\tau ?f\ ?P}$$

($\Box$) **rules**

$$\frac{?P \rightsquigarrow_\tau ?P'}{?P \Box ?Q \rightsquigarrow_\tau ?P' \Box ?Q} \qquad \frac{?Q \rightsquigarrow_\tau ?Q'}{?P \Box ?Q \rightsquigarrow_\tau ?P \Box ?Q'}$$

$$\frac{?P \rightsquigarrow_{?e} ?P'}{?P \Box ?Q \rightsquigarrow_{?e} ?P'} \qquad \frac{?Q \rightsquigarrow_{?e} ?Q'}{?P \Box ?Q \rightsquigarrow_{?e} ?Q'}$$

$$\frac{?P \rightsquigarrow_{\checkmark ?r} ?P'}{?P \Box ?Q \rightsquigarrow_{\checkmark ?r} \Omega\ (SKIP\ ?r)\ ?r} \qquad \frac{?Q \rightsquigarrow_{\checkmark ?r} ?Q'}{?P \Box ?Q \rightsquigarrow_{\checkmark ?r} \Omega\ (SKIP\ ?r)\ ?r}$$

(;) **rules**

$$\frac{?P \rightsquigarrow_\tau ?P'}{?P\ ;\ ?Q \rightsquigarrow_\tau ?P'\ ;\ ?Q}$$

$$\frac{?P \rightsquigarrow_{?e} ?P'}{?P\ ;\ ?Q \rightsquigarrow_{?e} ?P'\ ;\ ?Q} \qquad \frac{?P \rightsquigarrow_{\checkmark ?r} ?P' \qquad ?Q \rightsquigarrow_\tau ?Q'}{?P\ ;\ ?Q \rightsquigarrow_\tau ?Q'}$$

($\setminus$) **rules**

$$\frac{?P \rightsquigarrow_\tau ?P'}{?P \setminus ?A \rightsquigarrow_\tau ?P' \setminus ?A} \qquad \frac{?P \rightsquigarrow_{\checkmark ?r} ?P'}{?P \setminus ?B \rightsquigarrow_{\checkmark ?r} \Omega\ (SKIP\ ?r)\ ?r}$$

$$\frac{?e \notin ?A \qquad ?P \rightsquigarrow_{?e} ?P'}{?P \setminus ?A \rightsquigarrow_{?e} ?P' \setminus ?A} \qquad \frac{?e \in ?A \qquad ?P \rightsquigarrow_{?e} ?P'}{?P \setminus ?A \rightsquigarrow_\tau ?P' \setminus ?A}$$

Sync **rules**

$$\frac{?P \rightsquigarrow_\tau ?P'}{?P \ [\![?S]\!] \ ?Q \rightsquigarrow_\tau ?P' \ [\![?S]\!] \ ?Q} \qquad \frac{?Q \rightsquigarrow_\tau ?Q'}{?P \ [\![?S]\!] \ ?Q \rightsquigarrow_\tau ?P \ [\![?S]\!] \ ?Q'}$$

$$\frac{?e \notin ?S \qquad ?P \rightsquigarrow_{?e} ?P'}{?P \ [\![?S]\!] \ ?Q \rightsquigarrow_{?e} ?P' \ [\![?S]\!] \ ?Q} \qquad \frac{?e \notin ?S \qquad ?Q \rightsquigarrow_{?e} ?Q'}{?P \ [\![?S]\!] \ ?Q \rightsquigarrow_{?e} ?P \ [\![?S]\!] \ ?Q'}$$

$$\frac{?e \in ?S \qquad ?P \rightsquigarrow_{?e} ?P' \qquad ?Q \rightsquigarrow_{?e} ?Q'}{?P \ [\![?S]\!] \ ?Q \rightsquigarrow_{?e} ?P' \ [\![?S]\!] \ ?Q'}$$

$$\frac{?P \rightsquigarrow_{\checkmark ?r} ?P'}{?P \ [\![?S]\!] \ ?Q \rightsquigarrow_\tau SKIP \ ?r \ [\![?S]\!] \ ?Q}$$

$$\frac{?Q \rightsquigarrow_{\checkmark ?r} ?Q'}{?P \ [\![?S]\!] \ ?Q \rightsquigarrow_\tau ?P \ [\![?S]\!] \ SKIP \ ?r}$$

$$\frac{}{SKIP \ ?r \ [\![?S]\!] \ SKIP \ ?r \rightsquigarrow_{\checkmark ?r} \Omega \ (SKIP \ ?r) \ ?r}$$

$(\triangleright)$ **rules**

$$\frac{}{?P \triangleright ?Q \rightsquigarrow_\tau ?Q} \qquad \frac{?P \rightsquigarrow_\tau ?P'}{?P \triangleright ?Q \rightsquigarrow_\tau ?P' \triangleright ?Q}$$

$$\frac{?P \rightsquigarrow_{?e} ?P'}{?P \triangleright ?Q \rightsquigarrow_{?e} ?P'} \qquad \frac{?P \rightsquigarrow_{\checkmark ?r} ?P'}{?P \triangleright ?Q \rightsquigarrow_{\checkmark ?r} \Omega \ (SKIP \ ?r) \ ?r}$$

$(\triangle)$ **rules**

$$\frac{?P \rightsquigarrow_\tau ?P'}{?P \triangle ?Q \rightsquigarrow_\tau ?P' \triangle ?Q} \qquad \frac{?Q \rightsquigarrow_\tau ?Q'}{?P \triangle ?Q \rightsquigarrow_\tau ?P \triangle ?Q'}$$

$$\frac{?P \rightsquigarrow_{?e} ?P'}{?P \triangle ?Q \rightsquigarrow_{?e} ?P' \triangle ?Q} \qquad \frac{?Q \rightsquigarrow_{?e} ?Q'}{?P \triangle ?Q \rightsquigarrow_{?e} ?Q'}$$

$$\frac{?P \rightsquigarrow_{\checkmark ?r} ?P'}{?P \triangle ?Q \rightsquigarrow_{\checkmark ?r} \Omega \ (SKIP \ ?r) \ ?r} \qquad \frac{?Q \rightsquigarrow_{\checkmark ?r} ?Q'}{?P \triangle ?Q \rightsquigarrow_{\checkmark ?r} \Omega \ (SKIP \ ?r) \ ?r}$$

Throw **rules**

$$\frac{?P \rightsquigarrow_\tau ?P'}{?P \ \Theta \ a{\in}?A. \ ?Q \ a \rightsquigarrow_\tau ?P' \ \Theta \ a{\in}?A. \ ?Q \ a}$$

$$\frac{?P \rightsquigarrow_{\checkmark ?r} ?P'}{?P \ \Theta \ a{\in}?A. \ ?Q \ a \rightsquigarrow_{\checkmark ?r} \Omega \ (SKIP \ ?r) \ ?r}$$

$$\frac{?e \notin ?A \qquad ?P \rightsquigarrow_{?e} ?P'}{?P \;\Theta\; a{\in}?A.\; ?Q\; a \rightsquigarrow_{?e} ?P' \;\Theta\; a{\in}?A.\; ?Q\; a}$$

$$\frac{?e \in ?A \qquad ?P \rightsquigarrow_{?e} ?P'}{?P \;\Theta\; a{\in}?A.\; ?Q\; a \rightsquigarrow_{?e} ?Q\; ?e}$$

end

context *OpSemTransitionsAllDuplicated* **begin**

Renaming **rules**

$$\frac{?P \;{}_{\alpha}\!\rightsquigarrow_{\tau}\; ?P'}{Renaming\; ?P\; ?f\; ?g \;{}_{\beta}\!\rightsquigarrow_{\tau}\; Renaming\; ?P'\; ?f\; ?g}$$

$$\frac{?P \;{}_{\alpha}\!\rightsquigarrow_{\checkmark ?r}\; ?P'}{Renaming\; ?P\; ?f\; ?g \;{}_{\beta}\!\rightsquigarrow_{\checkmark ?g\; ?r}\; \Omega_{\beta}\; (SKIP\; (?g\; ?r))\; (?g\; ?r)}$$

$$\frac{?f\; ?a = ?b \qquad ?P \;{}_{\alpha}\!\rightsquigarrow_{?a}\; ?P'}{Renaming\; ?P\; ?f\; ?g \;{}_{\beta}\!\rightsquigarrow_{?b}\; Renaming\; ?P'\; ?f\; ?g}$$

end

7.2 Special Cases

7.2.1 With the Refinement ($\sqsubseteq_{DT}$)

context *OpSemDT* **begin**

($\Box$) **rules**

$$\frac{}{P \;\Box\; Q \;{}_{DT}\!\rightsquigarrow_{\tau}\; P} \qquad \frac{}{P \;\Box\; Q \;{}_{DT}\!\rightsquigarrow_{\tau}\; Q}$$

($\rhd$) **rules**

$$\frac{}{P \;\rhd\; Q \;{}_{DT}\!\rightsquigarrow_{\tau}\; P} \qquad \frac{}{P \;\rhd\; Q \;{}_{DT}\!\rightsquigarrow_{\tau}\; Q}$$

end

7.2.2 With the Refinement ($\sqsubseteq_{F}$)

context *OpSemF* **begin**

Absorbency rules

$$\frac{?P \ {}_F\!\rightsquigarrow_{?e} \ ?P' \qquad ?P' \ {}_F\!\rightsquigarrow_{\tau} \ ?P''}{?P \ {}_F\!\rightsquigarrow_{?e} \ ?P''} \qquad \frac{?P \ {}_F\!\rightsquigarrow_{\tau} \ ?P' \qquad ?P' \ {}_F\!\rightsquigarrow_{?e} \ ?P''}{?P \ {}_F\!\rightsquigarrow_{?e} \ ?P''}$$

$$\frac{?P \ {}_F\!\rightsquigarrow_{\checkmark ?r} \ ?P' \qquad ?P' \ {}_F\!\rightsquigarrow_{\tau} \ ?P''}{?P \ {}_F\!\rightsquigarrow_{\checkmark ?r} \ ?P''}$$

$$\frac{?P \ {}_F\!\rightsquigarrow_{\tau} \ ?P' \qquad ?P' \ {}_F\!\rightsquigarrow_{\checkmark ?r} \ ?P''}{?P \ {}_F\!\rightsquigarrow_{\checkmark ?r} \ ?P''}$$

SKIP **rule**

$$\frac{}{SKIP \ ?r \ {}_F\!\rightsquigarrow_{\checkmark ?r} \ \Omega \ (SKIP \ ?r) \ ?r}$$

$e \rightarrow P$ **rules**

$$\frac{}{?e \rightarrow ?Pa \ {}_F\!\rightsquigarrow_{?e} \ ?Pa}$$

$$\frac{?e \in ?A}{\Box a{\in}?A \rightarrow ?P \ a \ {}_F\!\rightsquigarrow_{?e} \ ?P \ ?e} \qquad \frac{?e \in ?A}{\sqcap a{\in}?A \rightarrow ?P \ a \ {}_F\!\rightsquigarrow_{?e} \ ?P \ ?e}$$

$(\sqcap)$ **rules**

$$\frac{}{?Pa \sqcap ?Q \ {}_F\!\rightsquigarrow_{\tau} \ ?Pa} \qquad \frac{}{?Pa \sqcap ?Q \ {}_F\!\rightsquigarrow_{\tau} \ ?Q}$$

$$\frac{?e \in ?A}{\sqcap a{\in}?A.\ ?P \ a \ {}_F\!\rightsquigarrow_{\tau} \ ?P \ ?e}$$

$\mu \ x.\ f \ x$ **rule**

$$\frac{cont \ ?f \qquad ?P = (\mu \ x.\ ?f \ x)}{?P \ {}_F\!\rightsquigarrow_{\tau} \ ?f \ ?P}$$

($\Box$) **rules**

$$\frac{?P = \bot \vee ?P' \neq \bot \vee ?Q = \bot \qquad ?P \ {}_{F}\!\rightsquigarrow_{\tau} \ ?P'}{?P \ \Box \ ?Q \ {}_{F}\!\rightsquigarrow_{\tau} \ ?P' \ \Box \ ?Q}$$

$$\frac{?Q = \bot \vee ?Q' \neq \bot \vee ?Q = \bot \qquad ?Q \ {}_{F}\!\rightsquigarrow_{\tau} \ ?Q'}{?P \ \Box \ ?Q \ {}_{F}\!\rightsquigarrow_{\tau} \ ?P \ \Box \ ?Q'}$$

$$\frac{?P \ {}_{F}\!\rightsquigarrow_{?e} \ ?P'}{?P \ \Box \ ?Q \ {}_{F}\!\rightsquigarrow_{?e} \ ?P'} \qquad \frac{?Q \ {}_{F}\!\rightsquigarrow_{?e} \ ?Q'}{?P \ \Box \ ?Q \ {}_{F}\!\rightsquigarrow_{?e} \ ?Q'}$$

$$\frac{?P \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ ?P'}{?P \ \Box \ ?Q \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ \Omega \ (SKIP \ ?r) \ ?r} \qquad \frac{?Q \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ ?Q'}{?P \ \Box \ ?Q \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ \Omega \ (SKIP \ ?r) \ ?r}$$

(;) **rules**

$$\frac{?P \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ ?P' \qquad ?Q \ {}_{F}\!\rightsquigarrow_{\tau} \ ?Q'}{?P \ ; \ ?Q \ {}_{F}\!\rightsquigarrow_{\tau} \ ?Q'}$$

($\setminus$) **rules**

$$\frac{?P \ {}_{F}\!\rightsquigarrow_{\tau} \ ?P'}{?P \setminus ?A \ {}_{F}\!\rightsquigarrow_{\tau} \ ?P' \setminus ?A} \qquad \frac{?P \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ ?P'}{?P \setminus ?B \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ \Omega \ (SKIP \ ?r) \ ?r}$$

$$\frac{?e \notin ?A \qquad ?P \ {}_{F}\!\rightsquigarrow_{?e} \ ?P'}{?P \setminus ?A \ {}_{F}\!\rightsquigarrow_{?e} \ ?P' \setminus ?A} \qquad \frac{?e \in ?A \qquad ?P \ {}_{F}\!\rightsquigarrow_{?e} \ ?P'}{?P \setminus ?A \ {}_{F}\!\rightsquigarrow_{\tau} \ ?P' \setminus ?A}$$

Sync **rules**

$$\frac{?P \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ ?P'}{?P \ [\![?S]\!] \ ?Q \ {}_{F}\!\rightsquigarrow_{\tau} \ SKIP \ ?r \ [\![?S]\!] \ ?Q}$$

$$\frac{?Q \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ ?Q'}{?P \ [\![?S]\!] \ ?Q \ {}_{F}\!\rightsquigarrow_{\tau} \ ?P \ [\![?S]\!] \ SKIP \ ?r}$$

$$\frac{}{SKIP \ ?r \ [\![?S]\!] \ SKIP \ ?r \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ \Omega \ (SKIP \ ?r) \ ?r}$$

($\rhd$) **rules**

$$\frac{}{?P \rhd ?Q \ {}_{F}\!\rightsquigarrow_{\tau} \ ?Q} \qquad \frac{?P = \bot \vee ?P' \neq \bot \vee ?Q = \bot \qquad ?P \ {}_{F}\!\rightsquigarrow_{\tau} \ ?P'}{?P \rhd ?Q \ {}_{F}\!\rightsquigarrow_{\tau} \ ?P' \rhd ?Q}$$

$$\frac{?P \ {}_{F}\!\rightsquigarrow_{?e} \ ?P'}{?P \rhd ?Q \ {}_{F}\!\rightsquigarrow_{?e} \ ?P'} \qquad \frac{?P \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ ?P'}{?P \rhd ?Q \ {}_{F}\!\rightsquigarrow_{\checkmark ?r} \ \Omega \ (SKIP \ ?r) \ ?r}$$

($\triangle$) **rules**

$$\frac{?P\ {}_{F}\!\rightsquigarrow\!\checkmark_{?r}\ ?P'}{?P\ \triangle\ ?Q\ {}_{F}\!\rightsquigarrow\!\checkmark_{?r}\ \Omega\ (SKIP\ ?r)\ ?r} \qquad \frac{?Q\ {}_{F}\!\rightsquigarrow\!\checkmark_{?r}\ ?Q'}{?P\ \triangle\ ?Q\ {}_{F}\!\rightsquigarrow\!\checkmark_{?r}\ \Omega\ (SKIP\ ?r)\ ?r}$$

Throw **rules**

$$\frac{\dfrac{?e \in ?A \qquad ?P\ {}_{F}\!\rightsquigarrow_{?e}\ ?P'}{?P\ \Theta\ a{\in}?A.\ ?Q\ a\ {}_{F}\!\rightsquigarrow_{?e}\ ?Q\ ?e} \qquad ?P\ {}_{F}\!\rightsquigarrow\!\checkmark_{?r}\ ?P'}{?P\ \Theta\ a{\in}?A.\ ?Q\ a\ {}_{F}\!\rightsquigarrow\!\checkmark_{?r}\ \Omega\ (SKIP\ ?r)\ ?r}$$

end

context *OpSemFDuplicated* **begin**

Renaming **rules**

$$\frac{?P\ {}_{\alpha F}\!\rightsquigarrow\!\checkmark_{?r}\ ?P'}{Renaming\ ?P\ ?f\ ?g\ {}_{\beta F}\!\rightsquigarrow\!\checkmark_{?g\ ?r}\ \Omega_{\beta}\ (SKIP\ (?g\ ?r))\ (?g\ ?r)}$$

end

7.2.3 With the Refinement ($\sqsubseteq_T$)

context *OpSemT* **begin**

Absorbency rules

$$\frac{?P\ {}_{T}\!\rightsquigarrow_{?e}\ ?P' \qquad ?P'\ {}_{T}\!\rightsquigarrow_{\tau}\ ?P''}{?P\ {}_{T}\!\rightsquigarrow_{?e}\ ?P''} \qquad \frac{?P\ {}_{T}\!\rightsquigarrow_{\tau}\ ?P' \qquad ?P'\ {}_{T}\!\rightsquigarrow_{?e}\ ?P''}{?P\ {}_{T}\!\rightsquigarrow_{?e}\ ?P''}$$

$$\frac{?P\ {}_{T}\!\rightsquigarrow\!\checkmark_{?r}\ ?P' \qquad ?P'\ {}_{T}\!\rightsquigarrow_{\tau}\ ?P''}{?P\ {}_{T}\!\rightsquigarrow\!\checkmark_{?r}\ ?P''} \qquad \frac{?P\ {}_{T}\!\rightsquigarrow_{\tau}\ ?P' \qquad ?P'\ {}_{T}\!\rightsquigarrow\!\checkmark_{?r}\ ?P''}{?P\ {}_{T}\!\rightsquigarrow\!\checkmark_{?r}\ ?P''}$$

SKIP **rule**

$$\frac{}{SKIP\ ?r\ {}_{T}\!\rightsquigarrow\!\checkmark_{?r}\ \Omega\ (SKIP\ ?r)\ ?r}$$

$e \rightarrow P$ **rules**

$$\frac{}{?e \rightarrow ?Pa \ {}_T\!\rightsquigarrow_{?e} \ ?Pa}$$

$$\frac{?e \in ?A}{\Box a{\in}?A \rightarrow ?P \ a \ {}_T\!\rightsquigarrow_{?e} \ ?P \ ?e} \qquad \frac{?e \in ?A}{\sqcap a{\in}?A \rightarrow ?P \ a \ {}_T\!\rightsquigarrow_{?e} \ ?P \ ?e}$$

$(\sqcap)$ **rules**

$$\frac{}{?Pa \sqcap ?Q \ {}_T\!\rightsquigarrow_{\tau} \ ?Pa} \qquad \frac{}{?Pa \sqcap ?Q \ {}_T\!\rightsquigarrow_{\tau} \ ?Q}$$

$$\frac{?e \in ?A}{\sqcap a{\in}?A.\ ?P \ a \ {}_T\!\rightsquigarrow_{\tau} \ ?P \ ?e}$$

$\mu \ x. \ f \ x$ **rule**

$$\frac{cont \ ?f \qquad ?P = (\mu \ x. \ ?f \ x)}{?P \ {}_T\!\rightsquigarrow_{\tau} \ ?f \ ?P}$$

$(\Box)$ **rules**

$$\frac{?P \ {}_T\!\rightsquigarrow_{\tau} \ ?P'}{?P \ \Box \ ?Q \ {}_T\!\rightsquigarrow_{\tau} \ ?P' \ \Box \ ?Q} \qquad \frac{?Q \ {}_T\!\rightsquigarrow_{\tau} \ ?Q'}{?P \ \Box \ ?Q \ {}_T\!\rightsquigarrow_{\tau} \ ?P \ \Box \ ?Q'}$$

$$\frac{?P \ {}_T\!\rightsquigarrow_{?e} \ ?P'}{?P \ \Box \ ?Q \ {}_T\!\rightsquigarrow_{?e} \ ?P'} \qquad \frac{?Q \ {}_T\!\rightsquigarrow_{?e} \ ?Q'}{?P \ \Box \ ?Q \ {}_T\!\rightsquigarrow_{?e} \ ?Q'}$$

$$\frac{?P \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ ?P'}{?P \ \Box \ ?Q \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ \Omega \ (SKIP \ ?r) \ ?r} \qquad \frac{?Q \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ ?Q'}{?P \ \Box \ ?Q \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ \Omega \ (SKIP \ ?r) \ ?r}$$

$(;)$ **rules**

$$\frac{?P \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ ?P' \qquad ?Q \ {}_T\!\rightsquigarrow_{\tau} \ ?Q'}{?P \ ; \ ?Q \ {}_T\!\rightsquigarrow_{\tau} \ ?Q'}$$

($\setminus$) **rules**

$$\frac{?P \ {}_T\!\rightsquigarrow_\tau \ ?P'}{?P \setminus ?A \ {}_T\!\rightsquigarrow_\tau \ ?P' \setminus ?A} \qquad \frac{?P \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ ?P'}{?P \setminus ?B \ {}_T\!\rightsquigarrow_{\checkmark ?r} \Omega\ (SKIP\ ?r)\ ?r}$$

$$\frac{?e \notin ?A \qquad ?P \ {}_T\!\rightsquigarrow_{?e} \ ?P'}{?P \setminus ?A \ {}_T\!\rightsquigarrow_{?e} \ ?P' \setminus ?A} \qquad \frac{?e \in ?A \qquad ?P \ {}_T\!\rightsquigarrow_{?e} \ ?P'}{?P \setminus ?A \ {}_T\!\rightsquigarrow_\tau \ ?P' \setminus ?A}$$

Sync **rules**

$$\frac{?P \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ ?P'}{?P \ [\![?S]\!] \ ?Q \ {}_T\!\rightsquigarrow_\tau \ SKIP\ ?r \ [\![?S]\!] \ ?Q}$$

$$\frac{?Q \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ ?Q'}{?P \ [\![?S]\!] \ ?Q \ {}_T\!\rightsquigarrow_\tau \ ?P \ [\![?S]\!] \ SKIP\ ?r}$$

$$\frac{}{SKIP\ ?r \ [\![?S]\!] \ SKIP\ ?r \ {}_T\!\rightsquigarrow_{\checkmark ?r} \Omega\ (SKIP\ ?r)\ ?r}$$

($\rhd$) **rules**

$$\frac{}{?P \rhd ?Q \ {}_T\!\rightsquigarrow_\tau \ ?Q} \qquad \frac{?P \ {}_T\!\rightsquigarrow_\tau \ ?P'}{?P \rhd ?Q \ {}_T\!\rightsquigarrow_\tau \ ?P' \rhd ?Q}$$

$$\frac{?P \ {}_T\!\rightsquigarrow_{?e} \ ?P'}{?P \rhd ?Q \ {}_T\!\rightsquigarrow_{?e} \ ?P'} \qquad \frac{?P \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ ?P'}{?P \rhd ?Q \ {}_T\!\rightsquigarrow_{\checkmark ?r} \Omega\ (SKIP\ ?r)\ ?r}$$

($\triangle$) **rules**

$$\frac{?P \ {}_T\!\rightsquigarrow_\tau \ ?P'}{?P \triangle ?Q \ {}_T\!\rightsquigarrow_\tau \ ?P' \triangle ?Q} \qquad \frac{?Q \ {}_T\!\rightsquigarrow_\tau \ ?Q'}{?P \triangle ?Q \ {}_T\!\rightsquigarrow_\tau \ ?P \triangle ?Q'}$$

$$\frac{?P \ {}_T\!\rightsquigarrow_{?e} \ ?P'}{?P \triangle ?Q \ {}_T\!\rightsquigarrow_{?e} \ ?P' \triangle ?Q} \qquad \frac{?Q \ {}_T\!\rightsquigarrow_{?e} \ ?Q'}{?P \triangle ?Q \ {}_T\!\rightsquigarrow_{?e} \ ?Q'}$$

$$\frac{?P \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ ?P'}{?P \triangle ?Q \ {}_T\!\rightsquigarrow_{\checkmark ?r} \Omega\ (SKIP\ ?r)\ ?r} \qquad \frac{?Q \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ ?Q'}{?P \triangle ?Q \ {}_T\!\rightsquigarrow_{\checkmark ?r} \Omega\ (SKIP\ ?r)\ ?r}$$

Throw **rules**

$$\frac{?e \in ?A \qquad ?P \ {}_T\!\rightsquigarrow_{?e} \ ?P'}{?P \ \Theta\ a{\in}?A.\ ?Q\ a \ {}_T\!\rightsquigarrow_{?e} \ ?Q\ ?e}$$

$$\frac{?P \ {}_T\!\rightsquigarrow_{\checkmark ?r} \ ?P'}{?P \ \Theta\ a{\in}?A.\ ?Q\ a \ {}_T\!\rightsquigarrow_{\checkmark ?r} \Omega\ (SKIP\ ?r)\ ?r}$$

Because we only look at the traces, we actully have the following results.

($\Box$) **rules**

$$\frac{}{P \;\Box\; Q \;{}_T\!\rightsquigarrow_\tau\; P} \qquad \frac{}{P \;\Box\; Q \;{}_T\!\rightsquigarrow_\tau\; Q}$$

($\rhd$) **rules**

$$\frac{}{P \;\rhd\; Q \;{}_T\!\rightsquigarrow_\tau\; P} \qquad \frac{}{P \;\rhd\; Q \;{}_T\!\rightsquigarrow_\tau\; Q}$$

end

context *OpSemTDuplicated* **begin**

Renaming **rules**

$$\frac{?P \;{}_{\alpha T}\!\rightsquigarrow_{\checkmark ?r}\; ?P'}{Renaming\; ?P\; ?f\; ?g \;{}_{\beta T}\!\rightsquigarrow_{\checkmark ?g\; ?r}\; \Omega_\beta\; (SKIP\; (?g\; ?r))\; (?g\; ?r)}$$

end

Chapter 8

Comparison with He and Hoare

lemma (**in** *After*) *initial-ev-imp-eq-prefix-After-Sliding* :
  ‹$P = (e \rightarrow (P \ after \ e)) \rhd P$› **if** ‹$ev \ e \in P^0$›
proof (*subst Process-eq-spec*, *safe*)
  show ‹$(s, X) \in \mathcal{F} \ P \Longrightarrow (s, X) \in \mathcal{F} \ ((e \rightarrow (P \ after \ e)) \rhd P)$› **for** $s \ X$
    by (*simp add*: *F-Sliding*)
next
  show ‹$(s, X) \in \mathcal{F} \ ((e \rightarrow (P \ after \ e)) \rhd P) \Longrightarrow (s, X) \in \mathcal{F} \ P$› **for** $s \ X$
    by (*cases s*) (*auto simp add*: *F-Sliding write0-def F-Mprefix F-After* ‹$ev \ e \in P^0$›)
next
  show ‹$s \in \mathcal{D} \ P \Longrightarrow s \in \mathcal{D} \ ((e \rightarrow (P \ after \ e)) \rhd P)$› **for** s
    by (*simp add*: *D-Sliding*)
next
  show ‹$s \in \mathcal{D} \ ((e \rightarrow (P \ after \ e)) \rhd P) \Longrightarrow s \in \mathcal{D} \ P$› **for** s
    by (*cases s*) (*auto simp add*: *D-Sliding write0-def D-Mprefix D-After* ‹$ev \ e \in P^0$›)
qed

context *OpSemTransitions*
begin

abbreviation τ-*eq* :: ‹$[('a, 'r) \ process_{ptick}, ('a, 'r) \ process_{ptick}] \Rightarrow bool$› (**infix** ‹$=_\tau$› *50*)
  where ‹$P =_\tau Q \equiv P \rightsquigarrow_\tau Q \wedge Q \rightsquigarrow_\tau P$›

lemma τ-*eqI* : $\langle P \rightsquigarrow_\tau Q \Longrightarrow Q \rightsquigarrow_\tau P \Longrightarrow P =_\tau Q\rangle$
and τ-*eqD1* : $\langle P =_\tau Q \Longrightarrow P \rightsquigarrow_\tau Q\rangle$
and τ-*eqD2* : $\langle P =_\tau Q \Longrightarrow Q \rightsquigarrow_\tau P\rangle$
by *simp-all*

lemma τ-*trans-iff-*τ-*eq-Ndet*:
$\langle \forall P\ Q\ P'\ Q'.\ P \rightsquigarrow_\tau P' \longrightarrow Q \rightsquigarrow_\tau Q' \longrightarrow P \sqcap Q \rightsquigarrow_\tau P' \sqcap Q' \Longrightarrow P \rightsquigarrow_\tau Q \longleftrightarrow P =_\tau P \sqcap Q\rangle$
by (*metis Ndet-id* τ-*trans-NdetL* τ-*trans-NdetR* τ-*trans-transitivity*)

lemma *eq-imp-*τ-*eq*: $\langle P = Q \Longrightarrow P =_\tau Q\rangle$ **by** (*simp add*: τ-*trans-eq*)

definition *ev-trans*$_{HOARE}$ :: $\langle('a, 'r)\ process_{ptick} \Rightarrow 'a \Rightarrow ('a, 'r)\ process_{ptick} \Rightarrow bool\rangle$ ($\langle$- $_{HOARE}\rightsquigarrow$- -$\rangle$ [50, 3, 51] 50)
where $\langle P\ {}_{HOARE}\rightsquigarrow_e\ Q \equiv P \rightsquigarrow_\tau (e \rightarrow Q) \Box P\rangle$

lemma *ev-trans*$_{HOARE}$-*imp-in-initials*:
$\langle P\ {}_{HOARE}\rightsquigarrow_e\ Q \Longrightarrow ev\ e \in P^0\rangle$
using τ-*trans-ev-trans ev-trans-DetL ev-trans-prefix* **unfolding** *ev-trans*$_{HOARE}$-*def*
by *blast*

lemma *ev-trans*$_{HOARE}$-*imp-ev-trans*: $\langle P \rightsquigarrow_e Q\rangle$ **if** $\langle P\ {}_{HOARE}\rightsquigarrow_e\ Q\rangle$
proof (*rule conjI*)
from $\langle P\ {}_{HOARE}\rightsquigarrow_e\ Q\rangle$ **show** $\langle ev\ e \in P^0\rangle$ **by** (*fact ev-trans*$_{HOARE}$-*imp-in-initials*)

from $\langle P\ {}_{HOARE}\rightsquigarrow_e\ Q\rangle$[*unfolded ev-trans*$_{HOARE}$-*def*]
have $\langle P\ after_{\checkmark}\ ev\ e \rightsquigarrow_\tau ((e \rightarrow Q) \Box P)\ after_{\checkmark}\ ev\ e\rangle$
by (*rule* τ-*trans-mono-After*$_{tick}$[*rotated*]) (*simp add*: *initials-Det initials-write0*)
moreover have $\langle \ldots = Q \sqcap (P\ after_{\checkmark}\ ev\ e)\rangle$
by (*simp add*: *After*$_{tick}$-*Det* $\langle ev\ e \in P^0\rangle$ *initials-write0 After*$_{tick}$-*write0*)
moreover have $\langle \ldots \rightsquigarrow_\tau Q\rangle$ **by** (*fact* τ-*trans-NdetL*)
ultimately show $\langle P\ after_{\checkmark}\ ev\ e \rightsquigarrow_\tau Q\rangle$
using $\langle ev\ e \in P^0\rangle$ *ev-trans-*τ-*trans* **by** *presburger*
qed

Two assumptions on τ transitions are necessary in the following proof, but are automatic when we instantiate ($\rightsquigarrow_\tau$) with ($\sqsubseteq_{FD}$), ($\sqsubseteq_{DT}$), ($\sqsubseteq_F$) or ($\sqsubseteq_T$).

lemma *hyps-on-*τ-*trans-imp-ev-trans-imp-ev-trans*$_{HOARE}$: $\langle P\ {}_{HOARE}\rightsquigarrow_e\ Q\rangle$
if *non-BOT-*τ-*trans-DetL* : $\langle \forall P\ P'\ Q.\ P = \bot \vee P' \neq \bot \longrightarrow P \rightsquigarrow_\tau P' \longrightarrow P \Box Q \rightsquigarrow_\tau P' \Box Q\rangle$
and τ-*trans-prefix* : $\langle \forall P\ P'\ e.\ P \rightsquigarrow_\tau P' \longrightarrow (e \rightarrow P) \rightsquigarrow_\tau (e \rightarrow P')\rangle$
and $\langle P \rightsquigarrow_e Q\rangle$
proof (*unfold ev-trans*$_{HOARE}$-*def*)
show $\langle P \rightsquigarrow_\tau (e \rightarrow Q) \Box P\rangle$

proof (*rule* τ*-trans-transitivity*)
have ‹$P = (e \rightarrow (P\ after\ e)) \rhd P$›
by (*simp add*: *initial-ev-imp-eq-prefix-After-Sliding* ‹$P \rightsquigarrow_e Q$›)
also have ‹$(e \rightarrow (P\ after\ e)) \rhd P \rightsquigarrow_\tau (e \rightarrow (P\ after\ e)) \mathbin{\square} P$›
by (*simp add*: *Sliding-def* τ*-trans-NdetL*)
finally show ‹$P \rightsquigarrow_\tau (e \rightarrow (P\ after\ e)) \mathbin{\square} P$› **.**
next
show ‹$(e \rightarrow (P\ after\ e)) \mathbin{\square} P \rightsquigarrow_\tau (e \rightarrow Q) \mathbin{\square} P$›
by (*rule non-BOT-*τ*-trans-DetL*[*rule-format*],
solves ‹*simp add*: *write0-def*›)
(*rule* τ*-trans-prefix*[*rule-format*],
fact ‹$P \rightsquigarrow_e Q$›[*simplified ev-trans-is*, *THEN conjunct2*])
qed
qed

— Of course, we obtain an equivalence.
lemma *hyps-on-*τ*-trans-imp-ev-trans*$_{HOARE}$*-iff-ev-trans*:
‹$\forall P\ P'\ Q.\ P = \bot \vee P' \neq \bot \longrightarrow P \rightsquigarrow_\tau P' \longrightarrow P \mathbin{\square} Q \rightsquigarrow_\tau P' \mathbin{\square} Q \Longrightarrow$
$\forall P\ P'\ e.\ P \rightsquigarrow_\tau P' \longrightarrow (e \rightarrow P) \rightsquigarrow_\tau (e \rightarrow P') \Longrightarrow P\ _{HOARE}{\rightsquigarrow}_e\ Q \longleftrightarrow P \rightsquigarrow_e Q$›
by (*rule iffI*[*OF ev-trans*$_{HOARE}$*-imp-ev-trans hyps-on-*τ*-trans-imp-ev-trans-imp-ev-trans*$_{HOARE}$])

— When $P = \bot$, we have the following result.
lemma *BOT-ev-trans*$_{HOARE}$*-anything*: ‹$\bot\ _{HOARE}{\rightsquigarrow}_e\ P$›
proof −
have ‹$\bot = (e \rightarrow P) \mathbin{\square} \bot$› **by** *simp*
thus ‹$\bot\ _{HOARE}{\rightsquigarrow}_e\ P$› **unfolding** *ev-trans*$_{HOARE}$*-def* **by** (*simp add*: τ*-trans-eq*)
qed

— Finally, again under two (automatic during instantiation) assumption on ($\rightsquigarrow_\tau$), we have an equivalent definition with an τ equality instead of a τ transition.
lemma *hyps-on-*τ*-trans-imp-ev-trans*$_{HOARE}$*-def-bis*: ‹$P\ _{HOARE}{\rightsquigarrow}_e\ Q \longleftrightarrow P =_\tau (e \rightarrow Q) \rhd P$›
if *non-BOT-*τ*-trans-SlidingL* : ‹$\forall P\ P'\ Q.\ P = \bot \vee P' \neq \bot \longrightarrow P \rightsquigarrow_\tau P' \longrightarrow P \rhd Q \rightsquigarrow_\tau P' \rhd Q$›
and τ*-trans-prefix* : ‹$\forall P\ P'\ e.\ P \rightsquigarrow_\tau P' \longrightarrow (e \rightarrow P) \rightsquigarrow_\tau (e \rightarrow P')$›
proof (*rule iffI*)
show ‹$P =_\tau (e \rightarrow Q) \rhd P \Longrightarrow P\ _{HOARE}{\rightsquigarrow}_e\ Q$›
by (*metis Sliding-def* τ*-trans-NdetL* τ*-trans-transitivity ev-trans*$_{HOARE}$*-def*)
next
show ‹$P =_\tau (e \rightarrow Q) \rhd P$› **if** ‹$P\ _{HOARE}{\rightsquigarrow}_e\ Q$›
proof (*rule* τ*-eqI*)
show ‹$(e \rightarrow Q) \rhd P \rightsquigarrow_\tau P$› **by** (*simp add*: τ*-trans-SlidingR*)
next
from ‹$P\ _{HOARE}{\rightsquigarrow}_e\ Q$› **have** ‹$P = (e \rightarrow (P\ after\ e)) \rhd P$›
by (*simp add*: *ev-trans*$_{HOARE}$*-imp-in-initials initial-ev-imp-eq-prefix-After-Sliding*)

also have ‹... $\rightsquigarrow_\tau$ $(e \rightarrow Q) \rhd P$›
by (*rule non-BOT-τ-trans-SlidingL*[*rule-format*],
solves ‹*simp add*: *write0-def*›)
(*rule τ-trans-prefix*[*rule-format*],
fact ‹P $_{HOARE}\!\!\rightsquigarrow_e$ Q›[*THEN ev-trans*$_{HOARE}$*-imp-ev-trans*,
unfolded ev-trans-is, *THEN conjunct2*])
finally show ‹$P \rightsquigarrow_\tau (e \rightarrow Q) \rhd P$› **.**
qed
qed

end

context *OpSemFD*
begin

notation *ev-trans*$_{HOARE}$ (‹- $_{FD-HOARE}\!\!\rightsquigarrow$- -› [*50*, *3*, *51*] *50*)
notation *τ-eq* (**infix** ‹$_{FD}=_\tau$› *50*)

theorem *ev-trans*$_{HOARE}$*-iff-ev-trans* : ‹P $_{FD-HOARE}\!\!\rightsquigarrow_e$ $Q \longleftrightarrow P$ $_{FD}\!\!\rightsquigarrow_e$ Q›
by (*simp add*: *τ-trans-DetL hyps-on-τ-trans-imp-ev-trans*$_{HOARE}$*-iff-ev-trans mono-write0-FD*)

theorem *ev-trans*$_{HOARE}$*-def-bis*: ‹P $_{FD-HOARE}\!\!\rightsquigarrow_e$ $Q \longleftrightarrow P$ $_{FD}=_\tau$ $(e \rightarrow Q)$
$\rhd P$›
by (*simp add*: *τ-trans-SlidingL hyps-on-τ-trans-imp-ev-trans*$_{HOARE}$*-def-bis mono-write0-FD*)

end

context *OpSemDT*
begin

notation *ev-trans*$_{HOARE}$ (‹- $_{DT-HOARE}\!\!\rightsquigarrow$- -› [*50*, *3*, *51*] *50*)
notation *τ-eq* (**infix** ‹$_{DT}=_\tau$› *50*)

theorem *ev-trans*$_{HOARE}$*-iff-ev-trans* : ‹P $_{DT-HOARE}\!\!\rightsquigarrow_e$ $Q \longleftrightarrow P$ $_{DT}\!\!\rightsquigarrow_e$ Q›
by (*simp add*: *τ-trans-DetL hyps-on-τ-trans-imp-ev-trans*$_{HOARE}$*-iff-ev-trans mono-write0-DT*)

theorem *ev-trans*$_{HOARE}$*-def-bis*: ‹P $_{DT-HOARE}\!\!\rightsquigarrow_e$ $Q \longleftrightarrow P$ $_{DT}=_\tau$ $(e \rightarrow Q)$
$\rhd P$›
by (*simp add*: *τ-trans-SlidingL hyps-on-τ-trans-imp-ev-trans*$_{HOARE}$*-def-bis mono-write0-DT*)

end

context *OpSemF*
begin

notation *ev-trans$_{HOARE}$* (‹- $_{F-HOARE}$⇝- -› [*50*, *3*, *51*] *50*)
notation *τ-eq* (**infix** ‹$_{F}$=$_{\tau}$› *50*)

theorem *ev-trans$_{HOARE}$-iff-ev-trans* : ‹*P* $_{F-HOARE}$⇝*e* *Q* ⟷ *P* $_{F}$⇝*e* *Q*›
by (*simp add*: *hyps-on-τ-trans-imp-ev-trans$_{HOARE}$-iff-ev-trans τ-trans-DetL mono-write0-F*)

theorem *ev-trans$_{HOARE}$-def-bis*: ‹*P* $_{F-HOARE}$⇝*e* *Q* ⟷ *P* $_{F}$=$_{\tau}$ (*e* → *Q*) ▷ *P*›
by (*simp add*: *hyps-on-τ-trans-imp-ev-trans$_{HOARE}$-def-bis τ-trans-SlidingL mono-write0-F*)

end

context *OpSemT*
begin

notation *ev-trans$_{HOARE}$* (‹- $_{T-HOARE}$⇝- -› [*50*, *3*, *51*] *50*)
notation *τ-eq* (**infix** ‹$_{T}$=$_{\tau}$› *50*)

theorem *ev-trans$_{HOARE}$-iff-ev-trans* : ‹*P* $_{T-HOARE}$⇝*e* *Q* ⟷ *P* $_{T}$⇝*e* *Q*›
using *τ-trans-DetL ev-trans$_{HOARE}$-imp-ev-trans hyps-on-τ-trans-imp-ev-trans-imp-ev-trans$_{HOARE}$ mono-write0-T* **by** *blast*

theorem *ev-trans$_{HOARE}$-def-bis*: ‹*P* $_{T-HOARE}$⇝*e* *Q* ⟷ *P* $_{T}$=$_{\tau}$ (*e* → *Q*) ▷ *P*›
by (*simp add*: *τ-trans-SlidingL hyps-on-τ-trans-imp-ev-trans$_{HOARE}$-def-bis mono-write0-T*)

end

8.1 Deadlock Results

lemma *initial-ev-imp-in-events-of*: ‹*ev a* ∈ *P*^{0} ⟹ *a* ∈ *α*(*P*)›
by (*meson AfterExt.events-of-iff-reachable-ev AfterExt.initial-ev-reachable*)

lemma *initial-tick-imp-in-ticks-of*: ‹✓(*r*) ∈ *P*^{0} ⟹ *r* ∈ ✓*s*(*P*)›
by (*meson AfterExt.initial-tick-reachable AfterExt.ticks-of-iff-reachable-tick*)

lemma ‹*UNIV* ∈ ℛ *P* ⟷ *P* ⊑$_F$ *STOP*›
unfolding *failure-refine-def*
apply (*simp add*: *D-STOP F-STOP Refusals-iff*)
using *is-processT4* **by** *blast*

lemma *no-events-of-if-at-most-initial-tick*: ‹*P*^{0} ⊆ *range tick* ⟹ *α*(*P*) = {}›
using *empty-ev-initials-iff-empty-events-of* **by** *fast*

lemma *deadlock-free-initial-evE*:

‹*deadlock-free* $P \Longrightarrow (\bigwedge a.\ ev\ a \in P^0 \Longrightarrow thesis) \Longrightarrow thesis$›
using *empty-ev-initials-iff-empty-events-of*
nonempty-events-of-if-deadlock-free **by** *fastforce*

context *AfterExt*
begin

As we said earlier, $After_{trace}$ allows us to obtain some very powerful results about *deadlock-free* and $deadlock\text{-}free_{SKIPS}$.

8.1.1 Preliminaries and induction Rules

context fixes P :: ‹('*a*, '*r*) $process_{ptick}$› **begin**

lemma $initials\text{-}After_{trace}\text{-}subset\text{-}events\text{-}of$:
‹$(P\ after_{\mathcal{T}}\ t)^0 \subseteq ev\ `\ \alpha(P)$› **if** ‹*non-terminating* P› ‹$t \in \mathcal{T}\ P$›
proof −
have ‹$tF\ t \Longrightarrow ev\ a \in (P\ after_{\mathcal{T}}\ t)^0 \Longrightarrow$ *reachable-ev* $P\ a$› **for** a
using $reachable\text{-}ev\text{-}iff\text{-}in\text{-}initials\text{-}After_{trace}\text{-}for\text{-}some\text{-}tickFree\text{-}T$ *that*(*2*) **by** *blast*
also have ‹$tF\ t \wedge$ *range tick* $\cap\ (P\ after_{\mathcal{T}}\ t)^0 = \{\}$›
by (*simp add*: *disjoint-iff image-iff*)
(*metis* $T\text{-}After_{trace}\text{-}eq$ *initials-def mem-Collect-eq non-terminating-is-right*
non-tickFree-tick that(*1*) *that*(*2*) *tickFree-append-iff*)
ultimately show ‹$(P\ after_{\mathcal{T}}\ t)^0 \subseteq ev\ `\ \alpha(P)$›
by (*simp add*: *subset-iff image-iff disjoint-iff*)
(*metis* $event_{ptick}.exhaust$ *events-of-iff-reachable-ev*)
qed

end

With the next result, the general idea appears: instead of doing an induction only on the process P we are interested in, we include a quantification over all the processes than can be reached from P after some trace of P.

theorem $After_{trace}\text{-}fix\text{-}ind$ [*consumes 2*, *case-names cont step*]:
fixes *ref* :: ‹[('*a*, '*r*) $process_{ptick}$, ('*a*, '*r*) $process_{ptick}$] ⇒ *bool*› (**infix** ‹$\sqsubseteq_{ref}$› *60*)
assumes *adm-ref* : ‹$\bigwedge u\ v.$ *cont* (u :: ('*a*, '*r*) $process_{ptick}$ ⇒ ('*a*, '*r*) $process_{ptick}$) $\Longrightarrow$ *monofun* v
$\Longrightarrow$ *adm* $(\lambda x.\ u\ x \sqsubseteq_{ref} v\ x)$›
and *BOT-le-ref* : ‹$\bigwedge Q.\ \bot \sqsubseteq_{ref} Q$›
and *cont-f* : ‹*cont* f›
and *hyp* : ‹$\bigwedge s\ x.\ \forall Q \in \{Q.\ \exists s \in \mathcal{T}\ P.\ g\ s\ Q \wedge Q = P\ after_{\mathcal{T}}\ s\}.\ x \sqsubseteq_{ref} Q$
$\Longrightarrow$
$s \in \mathcal{T}\ P \Longrightarrow g\ s\ (P\ after_{\mathcal{T}}\ s) \Longrightarrow f\ x \sqsubseteq_{ref} P\ after_{\mathcal{T}}\ s$›
shows ‹$\forall Q \in \{Q.\ \exists s \in \mathcal{T}\ P.\ g\ s\ Q \wedge Q = P\ after_{\mathcal{T}}\ s\}.\ (\mu\ X.\ f\ X) \sqsubseteq_{ref} Q$›
apply (*induct rule*: *fix-ind*)
apply (*solves* ‹*simp add*: *adm-ref monofunI*›)
apply (*solves* ‹*simp add*: *BOT-le-ref*›)

using *hyp cont-f* **by** *auto*

lemma *After$_{trace}$-fix-ind-F* [*consumes 1*, *case-names cont step*]:
‹⟦$Q \in \{Q.\ \exists t \in \mathcal{T}\ P.\ g\ t\ Q \wedge Q = P\ after_{\mathcal{T}}\ t\}$; *cont f*;
$\bigwedge t\ x.\ ⟦\forall Q \in \{Q.\ \exists t \in \mathcal{T}\ P.\ g\ t\ Q \wedge Q = P\ after_{\mathcal{T}}\ t\}.\ x \sqsubseteq_F Q$;
$t \in \mathcal{T}\ P$; $g\ t\ (P\ after_{\mathcal{T}}\ t)⟧ \Longrightarrow f\ x \sqsubseteq_F P\ after_{\mathcal{T}}\ t⟧ \Longrightarrow$
$(\mu\ X.\ f\ X) \sqsubseteq_F Q$›
and *After$_{trace}$-fix-ind-D* [*consumes 1*, *case-names cont step*]:
‹⟦$Q \in \{Q.\ \exists t \in \mathcal{T}\ P.\ g\ t\ Q \wedge Q = P\ after_{\mathcal{T}}\ t\}$; *cont f*;
$\bigwedge t\ x.\ ⟦\forall Q \in \{Q.\ \exists t \in \mathcal{T}\ P.\ g\ t\ Q \wedge Q = P\ after_{\mathcal{T}}\ t\}.\ x \sqsubseteq_D Q$;
$t \in \mathcal{T}\ P$; $g\ t\ (P\ after_{\mathcal{T}}\ t)⟧ \Longrightarrow f\ x \sqsubseteq_D P\ after_{\mathcal{T}}\ t⟧ \Longrightarrow$
$(\mu\ X.\ f\ X) \sqsubseteq_D Q$›
and *After$_{trace}$-fix-ind-T* [*consumes 1*, *case-names cont step*]:
‹⟦$Q \in \{Q.\ \exists t \in \mathcal{T}\ P.\ g\ t\ Q \wedge Q = P\ after_{\mathcal{T}}\ t\}$; *cont f*;
$\bigwedge t\ x.\ ⟦\forall Q \in \{Q.\ \exists t \in \mathcal{T}\ P.\ g\ t\ Q \wedge Q = P\ after_{\mathcal{T}}\ t\}.\ x \sqsubseteq_T Q$;
$t \in \mathcal{T}\ P$; $g\ t\ (P\ after_{\mathcal{T}}\ t)⟧ \Longrightarrow f\ x \sqsubseteq_T P\ after_{\mathcal{T}}\ t⟧ \Longrightarrow$
$(\mu\ X.\ f\ X) \sqsubseteq_T Q$›
and *After$_{trace}$-fix-ind-FD* [*consumes 1*, *case-names cont step*]:
‹⟦$Q \in \{Q.\ \exists t \in \mathcal{T}\ P.\ g\ t\ Q \wedge Q = P\ after_{\mathcal{T}}\ t\}$; *cont f*;
$\bigwedge t\ x.\ ⟦\forall Q \in \{Q.\ \exists t \in \mathcal{T}\ P.\ g\ t\ Q \wedge Q = P\ after_{\mathcal{T}}\ t\}.\ x \sqsubseteq_{FD} Q$;
$t \in \mathcal{T}\ P$; $g\ t\ (P\ after_{\mathcal{T}}\ t)⟧ \Longrightarrow f\ x \sqsubseteq_{FD} P\ after_{\mathcal{T}}\ t⟧ \Longrightarrow$
$(\mu\ X.\ f\ X) \sqsubseteq_{FD} Q$›
and *After$_{trace}$-fix-ind-DT* [*consumes 1*, *case-names cont step*]:
‹⟦$Q \in \{Q.\ \exists t \in \mathcal{T}\ P.\ g\ t\ Q \wedge Q = P\ after_{\mathcal{T}}\ t\}$; *cont f*;
$\bigwedge t\ x.\ ⟦\forall Q \in \{Q.\ \exists t \in \mathcal{T}\ P.\ g\ t\ Q \wedge Q = P\ after_{\mathcal{T}}\ t\}.\ x \sqsubseteq_{DT} Q$;
$t \in \mathcal{T}\ P$; $g\ t\ (P\ after_{\mathcal{T}}\ t)⟧ \Longrightarrow f\ x \sqsubseteq_{DT} P\ after_{\mathcal{T}}\ t⟧ \Longrightarrow$
$(\mu\ X.\ f\ X) \sqsubseteq_{DT} Q$›
by (*all* ‹*rule After$_{trace}$-fix-ind*[*rule-format*]›) *simp-all*

corollary *reachable-processes-fix-ind* [*consumes 3*, *case-names cont step*]:
‹⟦$Q \in \mathcal{R}_{proc}\ P$;
$\bigwedge u\ v.\ ⟦cont\ (u :: ('a, 'r)\ process_{ptick} \Rightarrow ('a, 'r)\ process_{ptick});\ monofun\ v⟧ \Longrightarrow$
adm $(\lambda x.\ ref\ (u\ x)\ (v\ x))$;
$\bigwedge Q.\ ref\ \bot\ Q$;
cont f;
$\bigwedge t\ x.\ ⟦\forall Q \in \mathcal{R}_{proc}\ P.\ ref\ x\ Q;\ t \in \mathcal{T}\ P;\ tickFree\ t⟧ \Longrightarrow ref\ (f\ x)\ (P\ after_{\mathcal{T}}\ t)⟧$
$\Longrightarrow$
$ref\ (\mu\ x.\ f\ x)\ Q$›
by (*simp add*: *reachable-processes-is*,
rule After$_{trace}$-fix-ind[**where** $g = ‹\lambda s\ Q.\ tickFree\ s›$, *simplified*, *rule-format*];
simp)

corollary *reachable-processes-fix-ind-F* [*consumes 1*, *case-names cont step*]:
‹⟦$Q \in \mathcal{R}_{proc}\ P$; *cont f*;
$\bigwedge t\ x.\ \forall Q \in \mathcal{R}_{proc}\ P.\ x \sqsubseteq_F Q \Longrightarrow t \in \mathcal{T}\ P \Longrightarrow tickFree\ t \Longrightarrow f\ x \sqsubseteq_F P\ after_{\mathcal{T}}\ t⟧ \Longrightarrow$
$(\mu\ X.\ f\ X) \sqsubseteq_F Q$›

and *reachable-processes-fix-ind-D* [*consumes 1*, *case-names cont step*]:
‹⟦$Q \in \mathcal{R}_{proc}\ P$; *cont f*;
$\bigwedge t\ x.\ \forall\, Q \in \mathcal{R}_{proc}\ P.\ x \sqsubseteq_D\ Q \Longrightarrow t \in \mathcal{T}\ P \Longrightarrow$ *tickFree t* $\Longrightarrow f\ x \sqsubseteq_D\ P\ after_{\mathcal{T}}\ t$⟧ $\Longrightarrow$
$(\mu\ X.\ f\ X) \sqsubseteq_D\ Q$›
and *reachable-processes-fix-ind-T* [*consumes 1*, *case-names cont step*]:
‹⟦$Q \in \mathcal{R}_{proc}\ P$; *cont f*;
$\bigwedge t\ x.\ \forall\, Q \in \mathcal{R}_{proc}\ P.\ x \sqsubseteq_T\ Q \Longrightarrow t \in \mathcal{T}\ P \Longrightarrow$ *tickFree t* $\Longrightarrow f\ x \sqsubseteq_T\ P\ after_{\mathcal{T}}\ t$⟧ $\Longrightarrow$
$(\mu\ X.\ f\ X) \sqsubseteq_T\ Q$›
and *reachable-processes-fix-ind-FD* [*consumes 1*, *case-names cont step*]:
‹⟦$Q \in \mathcal{R}_{proc}\ P$; *cont f*;
$\bigwedge t\ x.\ \forall\, Q \in \mathcal{R}_{proc}\ P.\ x \sqsubseteq_{FD}\ Q \Longrightarrow t \in \mathcal{T}\ P \Longrightarrow$ *tickFree t* $\Longrightarrow f\ x \sqsubseteq_{FD}\ P\ after_{\mathcal{T}}\ t$⟧ $\Longrightarrow$
$(\mu\ X.\ f\ X) \sqsubseteq_{FD}\ Q$›
and *reachable-processes-fix-ind-DT* [*consumes 1*, *case-names cont step*]:
‹⟦$Q \in \mathcal{R}_{proc}\ P$; *cont f*;
$\bigwedge t\ x.\ \forall\, Q \in \mathcal{R}_{proc}\ P.\ x \sqsubseteq_{DT}\ Q \Longrightarrow t \in \mathcal{T}\ P \Longrightarrow$ *tickFree t* $\Longrightarrow f\ x \sqsubseteq_{DT}\ P\ after_{\mathcal{T}}\ t$⟧ $\Longrightarrow$
$(\mu\ X.\ f\ X) \sqsubseteq_{DT}\ Q$›
by (*all* ‹*rule reachable-processes-fix-ind*›) *simp-all*

8.1.2 New idea: (*after*) induct instead of ($after_{\mathcal{T}}$)

8.1.3 New results on $\mathcal{R}_{proc}$

lemma *reachable-processes-FD-refinement-propagation-induct* [*consumes 1*, *case-names cont base step*]:
— May be generalized or duplicated to other refinements.
assumes *reachable* : ‹$(Q :: ('a, 'r)\ process_{ptick}) \in \mathcal{R}_{proc}\ P$›
and *cont-f* : ‹*cont f*›
and *base* : ‹$(\mu\ x.\ f\ x) \sqsubseteq_{FD}\ P$›
and *step* : ‹$\bigwedge a.\ a \in \alpha(P) \Longrightarrow f\ (\mu\ x.\ f\ x)\ after\ a = (\mu\ x.\ f\ x)$›
shows ‹$(\mu\ x.\ f\ x) \sqsubseteq_{FD}\ Q$›
proof (*use reachable* **in** ‹*induct rule*: *reachable-processes.induct*›)
case *reachable-self*
show ‹$(\mu\ x.\ f\ x) \sqsubseteq_{FD}\ P$› **by** (*fact base*)
next
case (*reachable-after Q a*)
from *reachable-after.hyps*(*2*) **have** ‹$f\ (\mu\ x.\ f\ x) \sqsubseteq_{FD}\ Q$›
by (*subst* (*asm*) *cont-process-rec*[*OF refl cont-f*])
hence $*$: ‹$f\ (\mu\ x.\ f\ x)\ after\ a \sqsubseteq_{FD}\ Q\ after\ a$›
by (*simp add*: *mono-After-FD reachable-after.hyps*(*3*))
from *reachable-after.hyps*(*1*, *3*) **have** ‹$a \in \alpha(P)$›
using *events-of-reachable-processes-subset initial-ev-imp-in-events-of* **by** (*metis in-mono*)
hence $**$: ‹$f\ (\mu\ x.\ f\ x)\ after\ a = (\mu\ x.\ f\ x)$› **by** (*fact local.step*)
from $*$[*unfolded* $**$] **show** ‹$(\mu\ x.\ f\ x) \sqsubseteq_{FD}\ Q\ after\ a$› .
qed

theorem $\mathcal{R}_{proc}$*-fix-ind* [*consumes 3*, *case-names cont step*]:
fixes *ref* :: ‹[('*a*, '*r*) *process*$_{ptick}$, ('*a*, '*r*) *process*$_{ptick}$] ⇒ *bool*› (**infix** ‹$\sqsubseteq_{ref}$› *60*)
assumes *reachable* : ‹$Q \in \mathcal{R}_{proc}\ P$›
and *adm-ref* : ‹$\bigwedge u\ v.$ *cont* (u :: ('*a*, '*r*) *process*$_{ptick}$ ⇒ ('*a*, '*r*) *process*$_{ptick}$) $\Longrightarrow$ *monofun* v
$\Longrightarrow$ *adm* ($\lambda x.\ u\ x \sqsubseteq_{ref}\ v\ x$)›
and *BOT-le-ref* : ‹$\bigwedge Q.\ \bot \sqsubseteq_{ref}\ Q$›
and *cont-f* : ‹*cont* f›
and *hyp* : ‹$\bigwedge x.\ \forall\, Q \in \mathcal{R}_{proc}\ P.\ x \sqsubseteq_{ref}\ Q \Longrightarrow \forall\, Q \in \mathcal{R}_{proc}\ P.\ f\ x \sqsubseteq_{ref}\ Q$›
shows ‹$(\mu\ X.\ f\ X) \sqsubseteq_{ref}\ Q$›
proof −
have ‹$\forall\, Q \in \mathcal{R}_{proc}\ P.\ (\mu\ X.\ f\ X) \sqsubseteq_{ref}\ Q$›
apply (*induct rule*: *fix-ind*)
apply (*solves* ‹*simp add*: *adm-ref monofunI*›)
apply (*solves* ‹*simp add*: *BOT-le-ref*›)
using *hyp cont-f* **by** *auto*
with *reachable* **show** ‹$(\mu\ X.\ f\ X) \sqsubseteq_{ref}\ Q$› **by** *blast*
qed

lemma $\mathcal{R}_{proc}$*-fix-ind-FD* [*consumes 1*, *case-names cont step*]:
‹$[\![Q \in \mathcal{R}_{proc}\ P;$ *cont* $f;$ $\bigwedge x\ Q.\ \forall\, Q \in \mathcal{R}_{proc}\ P.\ x \sqsubseteq_{FD}\ Q \Longrightarrow Q \in \mathcal{R}_{proc}\ P \Longrightarrow f\ x \sqsubseteq_{FD}\ Q]\!]$
$\Longrightarrow (\mu\ X.\ f\ X) \sqsubseteq_{FD}\ Q$›
by (*rule* $\mathcal{R}_{proc}$*-fix-ind*) *auto*

— See what else we can add.

8.1.4 Induction Proofs

Generalizations

lemma ‹*Mprefix A P* = *Mprefix B Q* ⟹ *A* = *B*›
by (*drule arg-cong*[**where** *f* = *initials*]) (*auto simp add*: *initials-Mprefix*)

lemma ‹*Mndetprefix A P* = *Mprefix B Q* ⟹ *A* = *B*›
by (*drule arg-cong*[**where** *f* = *initials*]) (*auto simp add*: *initials-Mprefix initials-Mndetprefix*)

lemma ‹*Mndetprefix A P* = *Mndetprefix B Q* ⟹ *A* = *B*›
by (*drule arg-cong*[**where** *f* = *initials*]) (*auto simp add*: *initials-Mndetprefix*)

print-context

lemma *superset-initials-restriction-Mndetprefix-FD*:
‹$\sqcap a \in B \to P\ a \sqsubseteq_{FD}\ Q$›
if ‹$\sqcap a \in A \to P\ a \sqsubseteq_{FD}\ Q$› **and** ‹$\{e.\ ev\ e \in Q^0\} \subseteq B$› **and** ‹$A \neq \{\} \vee B = \{\}$›
supply *that'* = *that*(*1*)[*unfolded failure-divergence-refine-def failure-refine-def divergence-refine-def*]
proof (*unfold failure-divergence-refine-def failure-refine-def divergence-refine-def*, *safe*)
from *that'*[*THEN conjunct2*] *that*(*2*)
show ‹$s \in \mathcal{D}\ Q \Longrightarrow s \in \mathcal{D}$ (*Mndetprefix B P*)› **for** *s*
by (*simp add*: *D-Mndetprefix write0-def D-Mprefix subset-iff split*: *if-split-asm*)
(*metis initials-memI D-T empty-iff*)
next
from *that* **have** *that''* : ‹$Q^0 \subseteq ev\ `\ A$›
using *anti-mono-initials-FD initials-Mndetprefix* **by** *blast*
fix *s X*
assume ∗ : ‹$(s,\ X) \in \mathcal{F}\ Q$›
with *set-mp*[*OF that'*[*THEN conjunct1*] *this*]
consider ‹$s = []$› | *e s'* **where** ‹$s = ev\ e\ \#\ s'$› ‹$ev\ e \in Q^0$›
by (*simp add*: *F-Mndetprefix write0-def F-Mprefix split*: *if-split-asm*)
(*metis initials-memI F-T*)
thus ‹$(s,\ X) \in \mathcal{F}$ (*Mndetprefix B P*)›
proof *cases*
assume ‹$s = []$›
hence ‹$([],\ X \cup (-\ Q^0)) \in \mathcal{F}\ Q$›
by (*metis* ∗ *ComplD initials-memI is-processT5-S7' self-append-conv2*)
from *set-mp*[*OF that'*[*THEN conjunct1*] *this*] *that''* *that*(*2*, *3*)
show ‹$(s,\ X) \in \mathcal{F}$ (*Mndetprefix B P*)›
by (*simp add*: ‹$s = []$› *F-Mndetprefix write0-def F-Mprefix split*: *if-split-asm*) *blast*
next
fix *e s'* **assume** ‹$s = ev\ e\ \#\ s'$› ‹$ev\ e \in Q^0$›
with *set-mp*[*OF that'*[*THEN conjunct1*] ∗] *that*(*2*)
show ‹$(s,\ X) \in \mathcal{F}$ (*Mndetprefix B P*)›
by (*simp add*: *F-Mndetprefix write0-def F-Mprefix subset-iff split*: *if-split-asm*) *blast*
qed
qed

corollary *initials-restriction-Mndetprefix-FD*:
‹$\sqcap a \in A \to P\ a \sqsubseteq_{FD}\ Q \Longrightarrow \sqcap a \in \{e.\ ev\ e \in Q^0\} \to P\ a \sqsubseteq_{FD}\ Q$›
by (*cases* ‹$A = \{\}$›, *simp add*: *STOP-FD-iff*)
(*simp add*: *superset-initials-restriction-Mndetprefix-FD*)

corollary *events-of-restriction-Mndetprefix-FD*:
‹$\sqcap a \in A \to P\ a \sqsubseteq_{FD}\ (Q :: ('a,\ 'r)\ process_{ptick}) \Longrightarrow \sqcap a \in \alpha(Q) \to P\ a \sqsubseteq_{FD}\ Q$›
by (*cases* ‹$A = \{\}$›, *simp add*: *STOP-FD-iff*)
(*erule superset-initials-restriction-Mndetprefix-FD*,
simp-all add: *subset-iff initial-ev-imp-in-events-of*)

lemma *superset-initials-restriction-Mprefix-FD*:
‹$\Box a \in B \to P\ a \sqsubseteq_{FD}\ Q$›
if ‹$\Box a \in A \to P\ a \sqsubseteq_{FD}\ Q$› **and** ‹$\{e.\ ev\ e \in Q^0\} \subseteq B$›
and ‹$B \subseteq A$› — Stronger assumption than with *Mndetprefix*.
supply *that'* = *that*(*1*)[*unfolded failure-divergence-refine-def failure-refine-def divergence-refine-def*]
proof (*unfold failure-divergence-refine-def failure-refine-def divergence-refine-def*, *safe*)
from *that'*[*THEN conjunct2*] *that*(*2*)
show ‹$s \in \mathcal{D}\ Q \Longrightarrow s \in \mathcal{D}\ (Mprefix\ B\ P)$› **for** *s*
by (*simp add*: *D-Mprefix subset-iff image-iff*)
(*metis initials-memI D-T*)
next
from *that* **have** *that''* : ‹$Q^0 \subseteq ev\ `\ A$›
using *anti-mono-initials-FD initials-Mprefix* **by** *blast*
fix *s X*
assume ∗ : ‹$(s,\ X) \in \mathcal{F}\ Q$›
with *set-mp*[*OF that'*[*THEN conjunct1*] *this*]
consider ‹$s = []$› | ‹$\exists e\ s'.\ s = ev\ e\ \#\ s' \wedge ev\ e \in Q^0$›
by (*simp add*: *F-Mprefix*) (*metis initials-memI F-T*)
thus ‹$(s,\ X) \in \mathcal{F}\ (Mprefix\ B\ P)$›
proof *cases*
assume ‹$s = []$›
hence ‹$([],\ X \cup (-\ Q^0)) \in \mathcal{F}\ Q$›
by (*metis* ∗ *ComplD initials-memI is-processT5-S7' self-append-conv2*)
from *set-mp*[*OF that'*[*THEN conjunct1*] *this*] *that''* *that*(*2*, *3*)
show ‹$(s,\ X) \in \mathcal{F}\ (Mprefix\ B\ P)$› **by** (*simp add*: ‹$s = []$› *F-Mprefix*) *blast*
next
assume ‹$\exists e\ s'.\ s = ev\ e\ \#\ s' \wedge ev\ e \in Q^0$›
with *set-mp*[*OF that'*[*THEN conjunct1*] ∗] *that*(*2*)
show ‹$(s,\ X) \in \mathcal{F}\ (Mprefix\ B\ P)$› **by** (*auto simp add*: *F-Mprefix image-iff*)
qed
qed

corollary *initials-restriction-Mprefix-FD*:
‹$\{e.\ ev\ e \in Q^0\} \subseteq A \Longrightarrow \Box a \in A \to P\ a \sqsubseteq_{FD}\ Q \Longrightarrow$
$\Box a \in \{e.\ ev\ e \in Q^0\} \to P\ a \sqsubseteq_{FD}\ Q$›
by (*erule superset-initials-restriction-Mprefix-FD*; *simp*)

corollary *events-of-restriction-Mprefix-FD*:
‹$\alpha(Q) \subseteq A \Longrightarrow \Box a \in A \to P\ a \sqsubseteq_{FD}\ (Q :: ('a,\ 'r)\ process_{ptick}) \Longrightarrow$
$\Box a \in \alpha(Q) \to P\ a \sqsubseteq_{FD}\ Q$›
by (*erule superset-initials-restriction-Mprefix-FD*,
simp add: *subset-iff initial-ev-imp-in-events-of*)

— This is much better with ($\sqsubseteq_{DT}$) refinement.
lemma *superset-initials-restriction-Mprefix-DT*:
‹$\Box a \in B \to P\ a \sqsubseteq_{DT} Q$› **if** ‹$\Box a \in A \to P\ a \sqsubseteq_{DT} Q$› **and** ‹$\{e.\ ev\ e \in Q^0\} \subseteq B$›
supply *that'* = *that*(*1*)[*unfolded trace-divergence-refine-def*
trace-refine-def divergence-refine-def]
proof (*unfold trace-divergence-refine-def trace-divergence-refine-def*
trace-refine-def divergence-refine-def, *safe*)
from *that'*[*THEN conjunct2*] *that*(*2*)
show ‹$s \in \mathcal{D}\ Q \Longrightarrow s \in \mathcal{D}\ (Mprefix\ B\ P)$› **for** *s*
by (*simp add*: *D-Mprefix subset-iff image-iff*)
(*metis initials-memI D-T*)
next
from *that* **have** *that''* : ‹$Q^0 \subseteq ev\ `\ A$›
using *anti-mono-initials-DT initials-Mprefix* **by** *blast*
fix *s*
assume ∗ : ‹$s \in \mathcal{T}\ Q$›
with *set-mp*[*OF that'*[*THEN conjunct1*] *this*]
consider ‹$s = []$› | ‹$\exists e\ s'.\ s = ev\ e\ \#\ s' \wedge ev\ e \in Q^0$›
by (*simp add*: *T-Mprefix*) (*metis initials-memI*)
thus ‹$s \in \mathcal{T}\ (Mprefix\ B\ P)$›
proof *cases*
show ‹$s = [] \Longrightarrow s \in \mathcal{T}\ (Mprefix\ B\ P)$› **by** *simp*
next
assume ‹$\exists e\ s'.\ s = ev\ e\ \#\ s' \wedge ev\ e \in Q^0$›
with *set-mp*[*OF that'*[*THEN conjunct1*] ∗] *that*(*2*)
show ‹$s \in \mathcal{T}\ (Mprefix\ B\ P)$› **by** (*auto simp add*: *T-Mprefix*)
qed
qed

corollary *initials-restriction-Mprefix-DT*:
‹$\Box a \in A \to P\ a \sqsubseteq_{DT} Q \Longrightarrow \Box a \in \{e.\ ev\ e \in Q^0\} \to P\ a \sqsubseteq_{DT} Q$›
by (*erule superset-initials-restriction-Mprefix-DT*) *simp*

corollary *events-restriction-Mprefix-DT*:
‹$\Box a \in A \to P\ a \sqsubseteq_{DT} (Q :: ('a, 'r)\ process_{ptick}) \Longrightarrow \Box a \in \alpha(Q) \to P\ a \sqsubseteq_{DT} Q$›
by (*erule superset-initials-restriction-Mprefix-DT*)
(*simp add*: *subset-iff initial-ev-imp-in-events-of*)

Admissibility **lemma** *not-le-F-adm*[*simp*]: ‹*cont u* $\Longrightarrow$ *adm* ($\lambda x.\ \neg\ u\ x \sqsubseteq_F P$)›
by (*simp add*: *adm-def cont2contlubE ch2ch-cont failure-refine-def*
limproc-is-thelub F-LUB subset-iff) *blast*

lemma *not-le-T-adm*[*simp*]: ‹*cont u* $\Longrightarrow$ *adm* ($\lambda x.\ \neg\ u\ x \sqsubseteq_T P$)›
by (*simp add*: *adm-def cont2contlubE ch2ch-cont trace-refine-def*
limproc-is-thelub T-LUB subset-iff) *blast*

lemma *not-le-D-adm*[*simp*]: ‹*cont u* $\Longrightarrow$ *adm* ($\lambda x.\ \neg\ u\ x \sqsubseteq_D P$)›
by (*simp add*: *adm-def cont2contlubE ch2ch-cont divergence-refine-def*

limproc-is-thelub D-LUB subset-iff) *blast*

lemma *not-le-FD-adm*[*simp*]: ‹*cont u* ⟹ *adm* (λ*x*. ¬ *u x* ⊑$_{FD}$ *P*)›
by (*simp add*: *adm-def cont2contlubE ch2ch-cont failure-divergence-refine-def failure-refine-def divergence-refine-def limproc-is-thelub F-LUB D-LUB subset-iff*) *blast*

lemma *not-le-DT-adm*[*simp*]: ‹*cont u* ⟹ *adm* (λ*x*. ¬ *u x* ⊑$_{DT}$ *P*)›
by (*simp add*: *adm-def cont2contlubE ch2ch-cont trace-divergence-refine-def trace-refine-def divergence-refine-def limproc-is-thelub D-LUB T-LUB subset-iff*) *blast*

lemma *initials-refusal*:
‹(*t*, *UNIV*) ∈ ℱ *P*› **if** *assms*: ‹*t* ∈ 𝒯 *P*› ‹*tF t*› ‹(*t*, (*P after*$_{\mathcal{T}}$ *t*)^0) ∈ ℱ *P*›
proof (*rule ccontr*)
assume ‹(*t*, *UNIV*) ∉ ℱ *P*›
from *is-processT5-S7'*[*OF assms*(*3*), *of UNIV*, *simplified*, *OF this*]
obtain *e* **where** ‹*e* ∉ (*P after*$_{\mathcal{T}}$ *t*)^0 ∧ *t* @ [*e*] ∈ 𝒯 *P*› **..**
thus *False* **by** (*simp add*: *initials-def T-After*$_{trace}$*-eq assms*(*1*, *2*))
qed

lemma *leF-ev-initialE'* :
assumes ‹*STOP* ⊓ (⊓*a* ∈ *UNIV* → *P a*) ⊑$_F$ *Q*› ‹*Q* ≠ *STOP*› **obtains** *a* **where** ‹*ev a* ∈ *Q*^{0}›
proof −
from ‹*Q* ≠ *STOP*› *initials-empty-iff-STOP* **obtain** *e* **where** ‹*e* ∈ *Q*^{0}› **by** *blast*
with ‹*STOP* ⊓ (⊓*a* ∈ *UNIV* → *P a*) ⊑$_F$ *Q*› **obtain** *a* **where** ‹*e* = *ev a*›
by (*auto simp add*: *failure-refine-def F-Ndet F-STOP F-Mndetprefix'* *intro*: *T-F dest*!: *initials-memD*)
with ‹*e* ∈ *Q*^{0}› *that* **show** *thesis* **by** *blast*
qed

corollary *leF-ev-initialE* :
assumes ‹⊓*a* ∈ *UNIV* → *P a* ⊑$_F$ *Q*› **obtains** *a* **where** ‹*ev a* ∈ *Q*^{0}›
proof −
from ‹⊓*a* ∈ *UNIV* → *P a* ⊑$_F$ *Q*› *Ndet-F-self-right trans-F*
have ‹*STOP* ⊓ (⊓*a* ∈ *UNIV* → *P a*) ⊑$_F$ *Q*› **by** *blast*
moreover from ‹⊓*a* ∈ *UNIV* → *P a* ⊑$_F$ *Q*› **have** ‹*Q* ≠ *STOP*›
by (*auto simp add*: *Process-eq-spec failure-refine-def F-Mndetprefix' F-STOP*)
ultimately obtain *a* **where** ‹*ev a* ∈ *Q*^{0}› **by** (*rule leF-ev-initialE'*)
with *that* **show** *thesis* **by** *blast*
qed

lemma *leFD-ev-initialE'* :
‹*STOP* ⊓ (⊓*a* ∈ *UNIV* → *P a*) ⊑$_{FD}$ *Q* ⟹ *Q* ≠ *STOP* ⟹ (⋀*a*. *ev a* ∈ *Q*^{0} ⟹ *thesis*) ⟹ *thesis*›
by (*meson leFD-imp-leF leF-ev-initialE'*)

lemma *leFD-ev-initialE* :
‹⊓*a* ∈ *UNIV* → *P a* ⊑$_{FD}$ *Q* ⟹ (⋀*a*. *ev a* ∈ *Q*^{0} ⟹ *thesis*) ⟹ *thesis*›
by (*meson leFD-imp-leF leF-ev-initialE*)

method *prove-propagation* **uses** *simp base* =
induct rule: *reachable-processes-FD-refinement-propagation-induct*,
solves simp, *solves* ‹*use base in* ‹*simp add*: *simp*››, *solves* ‹*simp add*: *simp*›

The three following results illustrate how powerful are our new rules of induction.

Really ? The second version with ⟦*cont ?F*; *cont ?G*; *adm* (λ*x*. *?P* (*fst x*) (*snd x*)); *?P* ⊥ ⊥; ⋀*x y*. *?P x y* ⟹ *?P* (*?F x*) (*?G y*)⟧ ⟹ *?P* (*fix-syn ?F*) (*fix-syn ?G*) seems easier...

lemma
‹*Q* ∈ $\mathcal{R}_{proc}$ *P* ⟹ *DF* α(*P*) ⊑$_F$ *Q*› **if** *df-P*: ‹*deadlock-free P*›
proof (*unfold DF-def*, *induct rule*: *reachable-processes-fix-ind-F*)
show ‹*cont* (λ*x*. ⊓ *a* ∈ *events-of P* → *x*)› **by** *simp*
next
fix *x*
assume *hyp* : ‹∀ *Q* ∈ $\mathcal{R}_{proc}$ *P*. *x* ⊑$_F$ *Q*›
show ‹*s* ∈ $\mathcal{T}$ *P* ⟹ *tickFree s* ⟹ ⊓*a* ∈ *events-of P* → *x* ⊑$_F$ *P after*$_\mathcal{T}$ *s*› **for** *s*
proof (*unfold failure-refine-def*, *safe*)
show ‹*s* ∈ $\mathcal{T}$ *P* ⟹ *tickFree s* ⟹ (*t*, *X*) ∈ $\mathcal{F}$ (*P after*$_\mathcal{T}$ *s*) ⟹
(*t*, *X*) ∈ $\mathcal{F}$ (⊓*a* ∈ *events-of P* → *x*)› **for** *t X*
proof (*induct t arbitrary*: *s*)
case *Nil*
have ∗ : ‹*deadlock-free* (*P after*$_\mathcal{T}$ *s*)›
by (*metis After*$_{trace}$*.simps*(*1*) ‹*s* ∈ $\mathcal{T}$ *P*› *deadlock-free-After*$_{trace}$*-characterization df-P*)
have ‹([], *X* ∪ (− *initials* (*P after*$_\mathcal{T}$ *s*))) ∈ $\mathcal{F}$ (*P after*$_\mathcal{T}$ *s*)›
by (*metis ComplD Nil.prems T-After*$_{trace}$*-eq is-processT5-S7'*
mem-Collect-eq initials-After$_{trace}$ *self-append-conv2*)
from *this*[*THEN set-rev-mp*, *OF* ∗[*unfolded deadlock-free-F failure-refine-def*]]
show ‹([], *X*) ∈ $\mathcal{F}$ (⊓*a* ∈ *events-of P* → *x*)›
apply (*subst* (*asm*) *F-DF*, *simp add*: *F-Mndetprefix write0-def F-Mprefix image-iff*)
using *Nil.prems*(*1*) *deadlock-free-implies-non-terminating events-of-iff-reachable-ev*
reachable-ev-iff-in-initials-After$_{trace}$*-for-some-tickFree-T that* **by** *blast*
next

```
    case (Cons e t)
    from Cons.prems(1, 2) have * : ‹P after_T s ∈ R_proc P›
      by (auto simp add: reachable-processes-is)
    obtain e' where ** : ‹e = ev e'› ‹e' ∈ events-of P›
      by (meson Cons.prems(1, 3) initials-memI F-T set-mp
            deadlock-free-implies-non-terminating df-P imageE
            non-terminating-is-right initials-After_trace-subset-events-of)
    from Cons.prems F-After_trace-eq **(1)
    have ‹(t, X) ∈ F (P after_T (s @ [e]))› by (simp add: F-After_trace)
    also have ‹... ⊆ F x›
      apply (rule hyp[rule-format, unfolded failure-refine-def,
                of ‹P after_T (s @ [e])›, simplified])
      apply (simp add: After_trace-snoc **(1) After_tick-def)
      apply (rule reachable-after[OF * ])
      using Cons.prems(3) initials-memI F-T **(1) by blast
    finally have ‹(t, X) ∈ F x› .
    with ** show ?case by (auto simp add: F-Mndetprefix write0-def F-Mprefix)
  qed
 qed
qed


lemma
 ‹Q ∈ R_proc P ⟹ DF_SKIPS α(P) UNIV ⊑_F Q› if df_SKIP-P: ‹deadlock-free_SKIPS
P›
proof (unfold DF_SKIPS-def, induct rule: reachable-processes-fix-ind-F)
 show ‹cont (λx. (⊓a ∈ events-of P → x) ⊓ SKIPS UNIV)› by simp
next
 fix x
 assume hyp : ‹∀ Q ∈ R_proc P. x ⊑_F Q›
 show ‹s ∈ T P ⟹ tickFree s ⟹ (⊓a ∈ events-of P → x) ⊓ SKIPS UNIV ⊑_F
P after_T s› for s
 proof (unfold failure-refine-def, safe)
   show ‹s ∈ T P ⟹ tickFree s ⟹ (t, X) ∈ F (P after_T s) ⟹
        (t, X) ∈ F ((⊓a ∈ events-of P → x) ⊓ SKIPS UNIV)› for t X
   proof (induct t arbitrary: s)
     case Nil
     have * : ‹deadlock-free_SKIPS (P after_T s)›
     by (metis After_trace.simps(1) Nil.prems(1, 2) deadlock-free_SKIPS-After_trace-characterization
df_SKIP-P)
     have ‹([], X ∪ (− initials (P after_T s))) ∈ F (P after_T s)›
       by (metis (no-types, opaque-lifting) ComplD initials-memI
             Nil.prems(3) append-eq-Cons-conv is-processT5-S7')
      from this[THEN set-rev-mp, OF *[unfolded deadlock-free_SKIPS-def fail-
ure-refine-def]]
     show ‹([], X) ∈ F ((⊓a ∈ events-of P → x) ⊓ SKIPS UNIV)›
      apply (subst (asm) F-DF_SKIPS, simp add: F-Ndet F-SKIPS F-Mndetprefix
write0-def F-Mprefix image-iff)
       using Nil.prems(1, 2) events-of-iff-reachable-ev
```

reachable-ev-iff-in-initials-After$_{trace}$-for-some-tickFree-T **by** *blast*
next
case (*Cons e t*)
from *Cons.prems*(*1*, *2*) **have** ∗ : ‹*P after$_{\mathcal{T}}$ s* ∈ $\mathcal{R}_{proc}$ *P*›
by (*auto simp add*: *reachable-processes-is*)
have ‹*deadlock-free$_{SKIPS}$* (*P after$_{\mathcal{T}}$ s*)›
by (*metis After$_{trace}$.simps*(*1*) *Cons.prems*(*1*, *2*) *deadlock-free$_{SKIPS}$-After$_{trace}$-characterization df$_{SKIP}$-P*)
then consider *a* **where** ‹*e* = *ev a*› ‹*a* ∈ *events-of P*› | *r* **where** ‹*e* = ✓(*r*)›
unfolding *deadlock-free$_{SKIPS}$-def failure-refine-def*
apply (*subst* (*asm*) *F-DF$_{SKIPS}$*, *simp add*: *subset-iff image-iff*)
by (*metis reachable-ev-iff-in-initials-After$_{trace}$-for-some-tickFree-T Cons.prems initials-memI F-T events-of-iff-reachable-ev list.distinct*(*1*) *list.sel*(*1*))
thus *?case*
proof *cases*
fix *a* **assume** ∗∗ : ‹*e* = *ev a*› ‹*a* ∈ *events-of P*›
from *Cons.prems F-After$_{trace}$* ∗∗(*1*)
have ‹(*t*, *X*) ∈ $\mathcal{F}$ (*P after$_{\mathcal{T}}$* (*s* @ [*e*]))› **by** (*simp add*: *F-After$_{trace}$-eq*)
also have ‹. . . ⊆ $\mathcal{F}$ *x*›
apply (*rule hyp*[*rule-format*, *unfolded failure-refine-def*, *of* ‹*P after$_{\mathcal{T}}$* (*s* @ [*e*])›, *simplified*])
apply (*simp add*: *After$_{trace}$-snoc* ∗∗(*1*) *After$_{tick}$-def*)
apply (*rule reachable-after*[*OF* ∗])
using *Cons.prems*(*3*) *initials-memI F-T* ∗∗(*1*) **by** *blast*
finally have ‹(*t*, *X*) ∈ $\mathcal{F}$ *x*› **.**
with ∗∗ **show** *?case* **by** (*auto simp add*: *F-Ndet F-Mndetprefix write0-def F-Mprefix*)
next
fix *r* **assume** ‹*e* = ✓(*r*)›
hence ‹*t* = []›
by (*metis Cons.prems*(*3*) *F-imp-front-tickFree event$_{ptick}$.disc*(*2*) *front-tickFree-Cons-iff*)
thus *?case* **by** (*simp add*: ‹*e* = ✓(*r*)› *F-Ndet F-SKIPS*)
qed
qed
qed
qed

context fixes *P* :: ‹('*a*, '*r*) *process$_{ptick}$*› **begin**

theorem *deadlock-free-iff-empty-ticks-of-and-deadlock-free$_{SKIPS}$* :
‹*deadlock-free P* ⟷ ✓*s*(*P*) = {} ∧ *deadlock-free$_{SKIPS}$ P*›
proof (*intro iffI conjI*)
from *deadlock-free-implies-non-terminating tickFree-traces-iff-empty-ticks-of*
show ‹*deadlock-free P* ⟹ ✓*s*(*P*) = {}› **by** *fast*
next
show ‹*deadlock-free P* ⟹ *deadlock-free$_{SKIPS}$ P*›
by (*fact deadlock-free-imp-deadlock-free$_{SKIPS}$*)
next

have ‹$Q \in \mathcal{R}_{proc}\ P \Longrightarrow DF_{SKIPS}\ UNIV\ UNIV \sqsubseteq_F Q \Longrightarrow DF\ UNIV \sqsubseteq_F Q$› **if** ‹$\checkmark s(P) = \{\}$› **for** Q
proof (*unfold DF-def* DF_{SKIPS}*-def*, *induct arbitrary*: Q *rule*: *cont-parallel-fix-ind*)
show ‹*cont* ($\lambda x.\ (\sqcap a{\in}UNIV \to x) \sqcap SKIPS\ UNIV$)› **by** *simp*
next
show ‹*cont* ($\lambda x.\ \sqcap a{\in}UNIV \to x$)› **by** *simp*
next
show ‹*adm* ($\lambda x.\ \forall Q.\ Q \in \mathcal{R}_{proc}\ P \longrightarrow fst\ x \sqsubseteq_F Q \longrightarrow snd\ x \sqsubseteq_F Q$)› **by** *simp*
next
show ‹$\bigsqcap Q.\ \bot \sqsubseteq_F Q$› **by** *simp*
next
fix $x\ y\ Q$ **assume** *hyp* : ‹$Q \in \mathcal{R}_{proc}\ P \Longrightarrow x \sqsubseteq_F Q \Longrightarrow y \sqsubseteq_F Q$› **for** Q
assume ‹$Q \in \mathcal{R}_{proc}\ P$› ‹$(\sqcap a \in UNIV \to x) \sqcap SKIPS\ UNIV \sqsubseteq_F Q$›
from ‹$\checkmark s(P) = \{\}$› ‹$Q \in \mathcal{R}_{proc}\ P$› **have** ‹$\checkmark s(Q) = \{\}$›
using *ticks-of-reachable-processes-subset* **by** *auto*

from ‹$\checkmark s(Q) = \{\}$› *initial-tick-imp-in-ticks-of* **have** ‹$\{r.\ \checkmark(r) \in Q^0\} = \{\}$› **by** *force*
moreover from ‹$(\sqcap a \in UNIV \to x) \sqcap SKIPS\ UNIV \sqsubseteq_F Q$›
have ‹$Q^0 \neq \{\}$› **by** (*auto simp add*: *initials-empty-iff-STOP Process-eq-spec failure-refine-def*
F-Ndet F-Mndetprefix′ F-SKIPS subset-iff F-STOP)
ultimately have ‹$\{a.\ ev\ a \in Q^0\} \neq \{\}$›
by (*metis empty-Collect-eq equals0I* $event_{ptick}$*.exhaust*)
hence ‹$\sqcap a \in UNIV \to y \sqsubseteq_F \sqcap a \in \{a.\ ev\ a \in Q^0\} \to y$›
by (*metis Mndetprefix-F-subset top-greatest*)
moreover have ‹$\ldots \sqsubseteq_F \sqcap a \in \{a.\ ev\ a \in Q^0\} \to Q\ after\ a$›
proof (*rule mono-Mndetprefix-F*, *clarify*, *rule hyp*)
show ‹$ev\ a \in Q^0 \Longrightarrow Q\ after\ a \in \mathcal{R}_{proc}\ P$› **for** a
by (*simp add*: ‹$Q \in \mathcal{R}_{proc}\ P$› *reachable-after*)
next
show ‹$ev\ a \in Q^0 \Longrightarrow x \sqsubseteq_F Q\ after\ a$› **for** a
by (*frule mono-After-F*[*OF* - ‹$(\sqcap a \in UNIV \to x) \sqcap SKIPS\ UNIV \sqsubseteq_F Q$›])
(*simp add*: *After-Ndet initials-Mndetprefix image-iff After-Mndetprefix*)
qed
moreover have ‹$\ldots \sqsubseteq_F Q$›
proof (*unfold failure-refine-def*, *safe*)
show ‹$(t,\ X) \in \mathcal{F}\ (\sqcap a \in \{a.\ ev\ a \in Q^0\} \to Q\ after\ a)$› **if** ‹$(t,\ X) \in \mathcal{F}\ Q$› **for** $t\ X$
proof (*cases* t)
from *initial-tick-imp-in-ticks-of* ‹$\checkmark s(Q) = \{\}$› **have** ‹*range tick* $\subseteq -\ Q^0$› **by** *force*
assume ‹$t = []$›
with ‹$(t,\ X) \in \mathcal{F}\ Q$› **have** ‹$([],\ X \cup -\ Q^0) \in \mathcal{F}\ Q$›
by (*metis ComplD initials-memI is-processT5-S7′ self-append-conv2*)
with ‹$(\sqcap a \in UNIV \to x) \sqcap SKIPS\ UNIV \sqsubseteq_F Q$›
show ‹$(t,\ X) \in \mathcal{F}\ (\sqcap a \in \{a.\ ev\ a \in Q^0\} \to Q\ after\ a)$›
by (*simp add*: *failure-refine-def F-Ndet F-SKIPS* ‹$t = []$› *F-Mndetprefix′ subset-iff*)

(*meson Compl-iff UnI1 UnI2* ‹*range tick* ⊆ − Q^0› *list.distinct*(*1*) *range-subsetD*)
next
from ‹(*t*, *X*) ∈ $\mathcal{F}$ *Q*› ‹(⊓*a* ∈ *UNIV* → *x*) ⊓ *SKIPS UNIV* ⊑$_F$ *Q*› ‹{*a*. *ev a* ∈ Q^0} ≠ {}› ‹✓*s*(*Q*) = {}›
show ‹*t* = *e* # *t*′ ⟹ (*t*, *X*) ∈ $\mathcal{F}$ (⊓*a* ∈ {*a*. *ev a* ∈ Q^0} → *Q after a*)› **for** *e t*′
by (*simp add*: *failure-refine-def F-Ndet F-SKIPS F-Mndetprefix*′ *F-After subset-iff*)
(*metis F-T empty-iff event*$_{ptick}$*.exhaust initial-tick-imp-in-ticks-of initials-memI*)
qed
qed
ultimately show ‹⊓*a* ∈ *UNIV* → *y* ⊑$_F$ *Q*› **using** *trans-F* **by** *blast*
qed
thus ‹✓*s*(*P*) = {} ∧ *deadlock-free*$_{SKIPS}$ *P* ⟹ *deadlock-free P*›
by (*simp add*: *reachable-self deadlock-free*$_{SKIPS}$*-def deadlock-free-F*)
qed

lemma *reachable-processes-DF-UNIV-leF-imp-DF-events-of-leF* :
‹*Q* ∈ $\mathcal{R}_{proc}$ *P* ⟹ *DF UNIV* ⊑$_F$ *Q* ⟹ *DF* α(*P*) ⊑$_F$ *Q*› **for** *Q*
proof (*unfold DF-def*, *induct arbitrary*: *Q rule*: *cont-parallel-fix-ind*)
show ‹*cont* (λ*x*. ⊓*a* ∈ *UNIV* → *x*)› **by** *simp*
next
show ‹*cont* (λ*x*. ⊓*a* ∈ α(*P*) → *x*)› **by** *simp*
next
show ‹*adm* (λ*x*. ∀ *Q*. *Q* ∈ $\mathcal{R}_{proc}$ *P* ⟶ *fst x* ⊑$_F$ *Q* ⟶ *snd x* ⊑$_F$ *Q*)› **by** *simp*
next
show ‹⋀*Q*. ⊥ ⊑$_F$ *Q*› **by** *simp*
next
fix *x y Q* **assume** *hyp* : ‹*Q* ∈ $\mathcal{R}_{proc}$ *P* ⟹ *x* ⊑$_F$ *Q* ⟹ *y* ⊑$_F$ *Q*› **for** *Q*
assume ‹*Q* ∈ $\mathcal{R}_{proc}$ *P*› **and** ‹⊓*a* ∈ *UNIV* → *x* ⊑$_F$ *Q*›

from ‹⊓*a* ∈ *UNIV* → *x* ⊑$_F$ *Q*› **obtain** *a* **where** ‹*ev a* ∈ Q^0› **by** (*elim leF-ev-initialE*)

have ‹⊓*a* ∈ α(*P*) → *y* ⊑$_F$ ⊓*a* ∈ {*a*. *ev a* ∈ Q^0} → *y*›
by (*metis* (*mono-tags*, *lifting*) *Mndetprefix-F-subset* ‹*Q* ∈ $\mathcal{R}_{proc}$ *P*›
‹*ev a* ∈ Q^0› *empty-iff events-of-reachable-processes-subset*
initial-ev-imp-in-events-of mem-Collect-eq subset-iff)
moreover have ‹... ⊑$_F$ ⊓*a* ∈ {*a*. *ev a* ∈ Q^0} → *Q after a*›
proof (*rule mono-Mndetprefix-F*, *clarify*)
show ‹*ev a* ∈ Q^0 ⟹ *y* ⊑$_F$ *Q after a*› **for** *a*
by (*metis mono-After-F After-Mndetprefix UNIV-I* ‹*Q* ∈ $\mathcal{R}_{proc}$ *P*›
‹⊓*a*∈*UNIV* → *x* ⊑$_F$ *Q*› *hyp reachable-processes.simps*)
qed
moreover have ‹... ⊑$_F$ *Q*›
proof (*unfold failure-refine-def*, *safe*)

show ‹$(t,\ X) \in \mathcal{F}\ (\sqcap a \in \{a.\ ev\ a \in Q^0\} \rightarrow Q\ after\ a)$› **if** ‹$(t,\ X) \in \mathcal{F}\ Q$› **for** $t\ X$
proof (*cases t*)
assume ‹$t = []$›
with ‹$(t,\ X) \in \mathcal{F}\ Q$› **have** ‹$([],\ X \cup (-\ Q^0)) \in \mathcal{F}\ Q$›
by (*metis ComplD initials-memI is-processT5-S7′ self-append-conv2*)
from ‹$\sqcap a \in UNIV \rightarrow x \sqsubseteq_F Q$› ‹$([],\ X \cup (-\ Q^0)) \in \mathcal{F}\ Q$› **have** ‹$\neg\ range\ ev \subseteq X \cup -\ Q^0$›
by (*simp add*: *failure-refine-def F-Mndetprefix′ subset-iff*) *blast*
then obtain *b* **where** ‹$ev\ b \in Q^0$› ‹$ev\ b \notin X$› **by** *auto*
with ‹$t = []$› ‹$ev\ a \in Q^0$› **show** ‹$(t,\ X) \in \mathcal{F}\ (\sqcap a \in \{a.\ ev\ a \in Q^0\} \rightarrow Q\ after\ a)$›
by (*auto simp add*: *F-Mndetprefix′*)
next
from ‹$\sqcap a \in UNIV \rightarrow x \sqsubseteq_F Q$› ‹$(t,\ X) \in \mathcal{F}\ Q$› ‹$ev\ a \in Q^0$›
show ‹$t = e\ \#\ t' \Longrightarrow (t,\ X) \in \mathcal{F}\ (\sqcap a \in \{a.\ ev\ a \in Q^0\} \rightarrow Q\ after\ a)$› **for** $e\ t'$
by (*auto simp add*: *failure-refine-def F-Mndetprefix′ F-After intro*!: *F-T initials-memI*)
qed
qed
ultimately show ‹$\sqcap a \in \alpha(P) \rightarrow y \sqsubseteq_F Q$› **using** *trans-F* **by** *blast*
qed

lemma *reachable-processes-CHAOS-UNIV-leF-imp-CHAOS-events-of-leF* :
‹$Q \in \mathcal{R}_{proc}\ P \Longrightarrow CHAOS\ UNIV \sqsubseteq_F Q \Longrightarrow CHAOS\ \alpha(P) \sqsubseteq_F Q$› **for** *Q*
proof (*unfold CHAOS-def*, *induct arbitrary*: *Q rule*: *cont-parallel-fix-ind*)
show ‹$cont\ (\lambda x.\ STOP \sqcap (\Box a \in UNIV \rightarrow x))$› **by** *simp*
next
show ‹$cont\ (\lambda x.\ STOP \sqcap (\Box a \in \alpha(P) \rightarrow x))$› **by** *simp*
next
show ‹$adm\ (\lambda x.\ \forall Q.\ Q \in \mathcal{R}_{proc}\ P \longrightarrow fst\ x \sqsubseteq_F Q \longrightarrow snd\ x \sqsubseteq_F Q)$› **by** *simp*
next
show ‹$\bigwedge Q.\ \bot \sqsubseteq_F Q$› **by** *simp*
next
fix *x y Q* **assume** *hyp* : ‹$Q \in \mathcal{R}_{proc}\ P \Longrightarrow x \sqsubseteq_F Q \Longrightarrow y \sqsubseteq_F Q$› **for** *Q*
assume ‹$Q \in \mathcal{R}_{proc}\ P$› **and** ‹$STOP \sqcap (\Box a{\in}UNIV \rightarrow x) \sqsubseteq_F Q$›

from ‹$Q \in \mathcal{R}_{proc}\ P$› *events-of-reachable-processes-subset initial-ev-imp-in-events-of*
have ‹$STOP \sqcap (\Box a \in \alpha(P) \rightarrow y) \sqsubseteq_F STOP \sqcap (\Box a \in \{a.\ ev\ a \in Q^0\} \rightarrow y)$›
by (*fastforce simp add*: *failure-refine-def Ndet-projs Mprefix-projs STOP-projs*)
moreover have ‹$\ldots \sqsubseteq_F STOP \sqcap (\Box a \in \{a.\ ev\ a \in Q^0\} \rightarrow Q\ after\ a)$›
proof (*intro mono-Ndet-F*[*OF idem-F*] *mono-Mprefix-F*, *clarify*)
show ‹$ev\ a \in Q^0 \Longrightarrow y \sqsubseteq_F Q\ after\ a$› **for** *a*
by (*frule mono-After-F*[*OF* - ‹$STOP \sqcap (\Box a{\in}UNIV \rightarrow x) \sqsubseteq_F Q$›])
(*simp add*: *After-Ndet initials-Mprefix After-Mprefix*
‹$Q \in \mathcal{R}_{proc}\ P$› *hyp reachable-after*)
qed

moreover have ‹... $\sqsubseteq_F$ Q›
proof (*unfold failure-refine-def*, *safe*)
from ‹$STOP \sqcap (\Box a{\in}UNIV \rightarrow x) \sqsubseteq_F Q$›
show ‹$(t, X) \in \mathcal{F}\ Q \Longrightarrow (t, X) \in \mathcal{F}\ (STOP \sqcap (\Box a \in \{a.\ ev\ a \in Q^0\} \rightarrow Q$ *after* $a)$)› **for** $t\ X$
by (*auto simp add*: *refine-defs F-Ndet F-Mprefix F-After intro*!: *F-T initials-memI*)
qed
ultimately show ‹$STOP \sqcap (\Box a \in \alpha(P) \rightarrow y) \sqsubseteq_F Q$› **using** *trans-F* **by** *blast*
qed

lemma *reachable-processes-CHAOS$_{SKIPS}$-UNIV-UNIV-leF-imp-CHAOS-events-of-ticks-of-leFD*
:
‹$Q \in \mathcal{R}_{proc}\ P \Longrightarrow CHAOS_{SKIPS}\ UNIV\ UNIV \sqsubseteq_{FD} Q \Longrightarrow CHAOS_{SKIPS}$
$\alpha(P)\ \checkmark s(P) \sqsubseteq_{FD} Q$› **for** Q
proof (*unfold CHAOS$_{SKIPS}$-def*, *induct arbitrary*: Q *rule*: *cont-parallel-fix-ind*)
show ‹*cont* $(\lambda x.\ SKIPS\ UNIV \sqcap STOP \sqcap (\Box a \in UNIV \rightarrow x))$› **by** *simp*
next
show ‹*cont* $(\lambda x.\ SKIPS\ \checkmark s(P) \sqcap STOP \sqcap (\Box a \in \alpha(P) \rightarrow x))$› **by** *simp*
next
show ‹*adm* $(\lambda x.\ \forall Q.\ Q \in \mathcal{R}_{proc}\ P \longrightarrow fst\ x \sqsubseteq_{FD} Q \longrightarrow snd\ x \sqsubseteq_{FD} Q)$› **by** *simp*
next
show ‹$\bigwedge Q.\ \bot \sqsubseteq_{FD} Q$› **by** *simp*
next
fix $x\ y\ Q$ **assume** *hyp* : ‹$Q \in \mathcal{R}_{proc}\ P \Longrightarrow x \sqsubseteq_{FD} Q \Longrightarrow y \sqsubseteq_{FD} Q$› **for** Q
assume ‹$Q \in \mathcal{R}_{proc}\ P$› **and** ‹$SKIPS\ UNIV \sqcap STOP \sqcap (\Box a \in UNIV \rightarrow x) \sqsubseteq_{FD} Q$›

from ‹$Q \in \mathcal{R}_{proc}\ P$› *ticks-of-reachable-processes-subset initial-tick-imp-in-ticks-of*
have ‹$SKIPS\ \checkmark s(P) \sqcap STOP \sqsubseteq_{FD} SKIPS\ \{r.\ \checkmark(r) \in Q^0\} \sqcap STOP$›
by (*fastforce simp add*: *refine-defs Ndet-projs STOP-projs SKIPS-projs*)
moreover from ‹$Q \in \mathcal{R}_{proc}\ P$› *events-of-reachable-processes-subset initial-ev-imp-in-events-of*
have ‹$STOP \sqcap (\Box a \in \alpha(P) \rightarrow y) \sqsubseteq_{FD} STOP \sqcap (\Box a \in \{a.\ ev\ a \in Q^0\} \rightarrow y)$›
by (*fastforce simp add*: *refine-defs Ndet-projs STOP-projs Mprefix-projs*)
ultimately have ‹$SKIPS\ \checkmark s(P) \sqcap STOP \sqcap (\Box a \in \alpha(P) \rightarrow y) \sqsubseteq_{FD}$
$SKIPS\ \{r.\ \checkmark(r) \in Q^0\} \sqcap STOP \sqcap (\Box a \in \{a.\ ev\ a \in Q^0\} \rightarrow y)$›
by (*auto simp add*: *refine-defs Ndet-projs*)
moreover have ‹... $\sqsubseteq_{FD}$ $SKIPS\ \{r.\ \checkmark(r) \in Q^0\} \sqcap STOP \sqcap (\Box a \in \{a.\ ev\ a \in Q^0\} \rightarrow Q$ *after* $a)$›
proof (*intro mono-Ndet-FD*[*OF idem-FD*] *mono-Mprefix-FD*, *clarify*)
show ‹$ev\ a \in Q^0 \Longrightarrow y \sqsubseteq_{FD} Q$ *after* a› **for** a
by (*frule mono-After-FD*[*OF* - ‹$SKIPS\ UNIV \sqcap STOP \sqcap (\Box a \in UNIV \rightarrow x) \sqsubseteq_{FD} Q$›])
(*simp add*: *After-Ndet initials-Mprefix initials-Ndet image-iff*
After-Mprefix ‹$Q \in \mathcal{R}_{proc}\ P$› *hyp reachable-after*)
qed
moreover have ‹... $\sqsubseteq_{FD}$ Q›

proof (*unfold refine-defs, safe*)
from ‹*SKIPS UNIV* ⊓ *STOP* ⊓ (□*a* ∈ *UNIV* → *x*) ⊑$_{FD}$ *Q*›
show ‹*t* ∈ $\mathcal{D}$ *Q* ⟹ *t* ∈ $\mathcal{D}$ (*SKIPS* {*r*. ✓(*r*) ∈ Q^0} ⊓ *STOP* ⊓ (□*a* ∈ {*a*. *ev a* ∈ Q^0} → *Q after a*))› **for** *t*
by (*auto simp add*: *refine-defs D-Ndet D-Mprefix D-STOP D-After D-SKIPS*)
(*use D-T initials-memI* **in** *blast*)
next
from ‹*SKIPS UNIV* ⊓ *STOP* ⊓ (□*a* ∈ *UNIV* → *x*) ⊑$_{FD}$ *Q*›
show ‹(*t*, *X*) ∈ $\mathcal{F}$ *Q* ⟹ (*t*, *X*) ∈ $\mathcal{F}$ (*SKIPS* {*r*. ✓(*r*) ∈ Q^0} ⊓ *STOP* ⊓ (□*a* ∈ {*a*. *ev a* ∈ Q^0} → *Q after a*))› **for** *t X*
by (*cases t*, *simp-all add*: *refine-defs F-Ndet F-Mprefix F-After F-SKIPS F-STOP*)
(*use F-T initials-memI* **in** *blast*)
qed
ultimately show ‹*SKIPS* ✓*s*(*P*) ⊓ *STOP* ⊓ (□*a* ∈ α(*P*) → *y*) ⊑$_{FD}$ *Q*› **by** *force*
qed

theorem *deadlock-free-iff-DF-events-of-leF* :
‹*deadlock-free P* ⟷ α(*P*) ≠ {} ∧ *DF* α(*P*) ⊑$_F$ *P*›
proof (*intro iffI conjI*)
show ‹α(*P*) ≠ {} ∧ *DF* α(*P*) ⊑$_F$ *P* ⟹ *deadlock-free P*›
by (*meson DF-Univ-freeness deadlock-free-F order-refl trans-F*)
next
show ‹*deadlock-free P* ⟹ α(*P*) ≠ {}› **by** (*simp add*: *nonempty-events-of-if-deadlock-free*)
next
show ‹*deadlock-free P* ⟹ *DF* α(*P*) ⊑$_F$ *P*›
by (*simp add*: *deadlock-free-F reachable-self*
reachable-processes-DF-UNIV-leF-imp-DF-events-of-leF)
qed

corollary *deadlock-free-iff-DF-events-of-leFD* :
‹*deadlock-free P* ⟷ α(*P*) ≠ {} ∧ *DF* α(*P*) ⊑$_{FD}$ *P*›
by (*metis deadlock-free-iff-DF-events-of-leF deadlock-free-def div-free-DF*
divergence-refine-def failure-divergence-refine-def)

corollary *deadlock-free-iff-DF-strict-events-of-leF* :
‹*deadlock-free P* ⟷ **α**(*P*) ≠ {} ∧ *DF* **α**(*P*) ⊑$_F$ *P*›
by (*metis anti-mono-events-of-F deadlock-free-iff-DF-events-of-leF*
deadlock-free-implies-div-free events-of-DF events-of-is-strict-events-of-or-UNIV
order-class.order-eq-iff strict-events-of-subset-events-of)

corollary *deadlock-free-iff-DF-strict-events-of-leFD* :
‹*deadlock-free P* ⟷ **α**(*P*) ≠ {} ∧ *DF* **α**(*P*) ⊑$_{FD}$ *P*›
by (*metis DF-Univ-freeness deadlock-free-iff-DF-events-of-leFD*
deadlock-free-implies-div-free events-of-is-strict-events-of-or-UNIV)

theorem *lifelock-free-iff-CHAOS-events-of-leF* :
‹*lifelock-free* $P \longleftrightarrow$ *CHAOS* $\alpha(P) \sqsubseteq_F P$›
proof (*intro iffI conjI*)
show ‹*CHAOS* $\alpha(P) \sqsubseteq_F P \Longrightarrow$ *lifelock-free* P›
by (*meson CHAOS-subset-FD lifelock-free-def non-terminating-F non-terminating-FD top-greatest trans-F*)
next
show ‹*lifelock-free* $P \Longrightarrow$ *CHAOS* $\alpha(P) \sqsubseteq_F P$›
by (*simp add*: *leFD-imp-leF lifelock-free-def reachable-self reachable-processes-CHAOS-UNIV-leF-imp-CHAOS-events-of-leF*)
qed

corollary *lifelock-free-iff-CHAOS-strict-events-of-leF* :
‹*lifelock-free* $P \longleftrightarrow$ *CHAOS* $\boldsymbol{\alpha}(P) \sqsubseteq_F P$›
by (*metis anti-mono-ticks-of-F bot.extremum-uniqueI empty-not-UNIV events-of-is-strict-events-of-or-UNIV lifelock-free-iff-CHAOS-events-of-leF ticks-CHAOS ticks-of-is-strict-ticks-of-or-UNIV*)

corollary *lifelock-free-iff-CHAOS-events-of-leFD* :
‹*lifelock-free* $P \longleftrightarrow$ *CHAOS* $\alpha(P) \sqsubseteq_{FD} P$›
by (*metis div-free-CHAOS divergence-refine-def failure-divergence-refine-def lifelock-free-def lifelock-free-iff-CHAOS-events-of-leF*)

corollary *lifelock-free-iff-CHAOS-strict-events-of-leFD* :
‹*lifelock-free* $P \longleftrightarrow$ *CHAOS* $\boldsymbol{\alpha}(P) \sqsubseteq_{FD} P$›
by (*metis anti-mono-events-of-F antisym events-of-CHAOS leFD-imp-leF lifelock-free-iff-CHAOS-events-of-leFD lifelock-free-iff-CHAOS-strict-events-of-leF strict-events-of-subset-events-of*)

theorem *lifelock-free*$_{SKIPS}$*-iff-CHAOS*$_{SKIPS}$*-events-of-ticks-of-FD* :
‹*lifelock-free*$_{SKIPS}$ $P \longleftrightarrow$ *CHAOS*$_{SKIPS}$ $\alpha(P)$ ✓$s(P) \sqsubseteq_{FD} P$›
proof (*intro iffI*)
show ‹*CHAOS*$_{SKIPS}$ $\alpha(P)$ ✓$s(P) \sqsubseteq_{FD} P \Longrightarrow$ *lifelock-free*$_{SKIPS}$ P›
by (*metis events-of-is-strict-events-of-or-UNIV lifelock-free*$_{SKIPS}$*-def lifelock-free*$_{SKIPS}$*-iff-div-free ticks-of-is-strict-ticks-of-or-UNIV*)
next
show ‹*lifelock-free*$_{SKIPS}$ $P \Longrightarrow$ *CHAOS*$_{SKIPS}$ $\alpha(P)$ ✓$s(P) \sqsubseteq_{FD} P$›
by (*simp add*: *lifelock-free*$_{SKIPS}$*-def reachable-self reachable-processes-CHAOS*$_{SKIPS}$*-UNIV-UNIV-leF-imp-CHAOS-events-of-ticks-of-leFD*)
qed

corollary *lifelock-free*$_{SKIPS}$*-iff-CHAOS*$_{SKIPS}$*-strict-events-of-strict-ticks-of-FD* :
‹*lifelock-free*$_{SKIPS}$ $P \longleftrightarrow$ *CHAOS*$_{SKIPS}$ $\boldsymbol{\alpha}(P)$ ✓$\mathbf{s}(P) \sqsubseteq_{FD} P$›

by (*metis* (*no-types*) *anti-mono-events-of-FD anti-mono-ticks-of-FD events-of-CHAOS*$_{SKIPS}$ *events-of-is-strict-events-of-or-UNIV lifelock-free*$_{SKIPS}$*-iff-CHAOS*$_{SKIPS}$*-events-of-ticks-of-FD lifelock-free*$_{SKIPS}$*-iff-div-free ticks-CHAOS*$_{SKIPS}$ *ticks-of-is-strict-ticks-of-or-UNIV top.extremum-uniqueI*)

theorem *deadlock-free*$_{SKIPS}$*-iff-DF*$_{SKIPS}$*-events-of-ticks-of-leF* :
‹*deadlock-free*$_{SKIPS}$ $P \longleftrightarrow$ (*if* $\alpha(P) = \{\} \wedge \checkmark s(P) = \{\}$ *then False*
else if $\checkmark s(P) = \{\}$ *then DF* $\alpha(P) \sqsubseteq_F P$
else if $\alpha(P) = \{\}$ *then SKIPS* $\checkmark s(P) \sqsubseteq_F P$
else DF$_{SKIPS}$ $\alpha(P)$ $\checkmark s(P) \sqsubseteq_F P$)›
(**is** ‹- $\longleftrightarrow$ *?rhs*›)
proof (*split if-split*, *intro conjI impI*)
from *deadlock-free-iff-empty-ticks-of-and-deadlock-free*$_{SKIPS}$ *nonempty-events-of-if-deadlock-free*
show ‹$\alpha(P) = \{\} \wedge \checkmark s(P) = \{\} \Longrightarrow$ *deadlock-free*$_{SKIPS}$ $P \longleftrightarrow$ *False*› **by** *blast*
next
assume ‹$\neg$ $(\alpha(P) = \{\} \wedge \checkmark s(P) = \{\})$›
show ‹*deadlock-free*$_{SKIPS}$ $P \longleftrightarrow$
(*if* $\checkmark s(P) = \{\}$ *then DF* $\alpha(P) \sqsubseteq_F P$
else if $\alpha(P) = \{\}$ *then SKIPS* $\checkmark s(P) \sqsubseteq_F P$ *else DF*$_{SKIPS}$ $\alpha(P)$ $\checkmark s(P)$ $\sqsubseteq_F P$)›
proof (*split if-split*, *intro conjI impI*)
from ‹$\neg$ $(\alpha(P) = \{\} \wedge \checkmark s(P) = \{\})$› *deadlock-free-iff-DF-events-of-leF*
deadlock-free-iff-empty-ticks-of-and-deadlock-free$_{SKIPS}$
show ‹$\checkmark s(P) = \{\} \Longrightarrow$ *deadlock-free*$_{SKIPS}$ $P \longleftrightarrow$ *DF* $\alpha(P) \sqsubseteq_F P$› **by** *blast*
next
show ‹*deadlock-free*$_{SKIPS}$ $P \longleftrightarrow$
(*if* $\alpha(P) = \{\}$ *then SKIPS* $\checkmark s(P) \sqsubseteq_F P$ *else DF*$_{SKIPS}$ $\alpha(P)$ $\checkmark s(P) \sqsubseteq_F P$)› **if** ‹$\checkmark s(P) \neq \{\}$›
proof (*split if-split*, *intro conjI impI*)
assume ‹$\alpha(P) = \{\}$›
with *initial-ev-imp-in-events-of* **have** ‹*range ev* $\subseteq -$ P^0› **by** *force*
show ‹*deadlock-free*$_{SKIPS}$ $P \longleftrightarrow$ *SKIPS* $\checkmark s(P) \sqsubseteq_F P$›
proof (*rule iffI*)
show ‹*SKIPS* $\checkmark s(P) \sqsubseteq_F P \Longrightarrow$ *deadlock-free*$_{SKIPS}$ P›
by (*metis deadlock-free*$_{SKIPS}$*-SKIPS deadlock-free*$_{SKIPS}$*-def* ‹$\checkmark s(P) \neq \{\}$› *trans-F*)
next
show ‹*SKIPS* $\checkmark s(P) \sqsubseteq_F P$› **if** ‹*deadlock-free*$_{SKIPS}$ P›
proof (*unfold failure-refine-def*, *safe*)
show ‹$(t, X) \in \mathcal{F}$ (*SKIPS* $\checkmark s(P)$)› **if** ‹$(t, X) \in \mathcal{F}$ P› **for** t X
proof (*cases t*)
assume ‹$t = []$›
with ‹$(t, X) \in \mathcal{F}$ P› **have** ‹$(t, X \cup - P^0) \in \mathcal{F}$ P›
by (*metis ComplD initials-memI is-processT5-S7′ self-append-conv2*)
with ‹*deadlock-free*$_{SKIPS}$ P› **have** ‹$(t, X \cup - P^0) \in \mathcal{F}$ (*DF*$_{SKIPS}$ *UNIV UNIV*)›
by (*auto simp add*: *deadlock-free*$_{SKIPS}$*-def failure-refine-def*)

```
            with ‹range ev ⊆ − P⁰› ‹✓s(P) ≠ {}› show ‹(t, X) ∈ F (SKIPS ✓s(P))›
              by (subst (asm) DF_SKIPS-unfold)
                (auto simp add: F-Ndet F-Mndetprefix′ F-SKIPS ‹t = []›
                      intro: initial-tick-imp-in-ticks-of)
          next
            fix e t′ assume ‹t = e # t′›
            with F-T ‹(t, X) ∈ F P› ‹α(P) = {}› obtain r where ‹r ∈ ✓s(P)› ‹e = ✓(r)› ‹t′ = []›
          by (metis F-imp-front-tickFree empty-iff event_ptick.exhaust events-of-memI
                    front-tickFree-Cons-iff initial-tick-imp-in-ticks-of
                    initials-memI is-ev-def list.set-intros(1))
            thus ‹(t, X) ∈ F (SKIPS ✓s(P))›
              by (simp add: F-Mndetprefix′ F-SKIPS ‹✓s(P) ≠ {}› ‹t = e # t′›)
          qed
        qed
      qed
    next
      show ‹deadlock-free_SKIPS P ⟷ DF_SKIPS α(P) ✓s(P) ⊑_F P› if ‹α(P) ≠ {}›
      proof (rule iffI)
        show ‹DF_SKIPS α(P) ✓s(P) ⊑_F P ⟹ deadlock-free_SKIPS P›
          by (meson DF_SKIPS-subset ‹✓s(P) ≠ {}› ‹α(P) ≠ {}›
                deadlock-free_SKIPS-def leFD-imp-leF subset-UNIV that trans-F)
      next
        have ‹Q ∈ R_proc P ⟹ DF_SKIPS UNIV UNIV ⊑_F Q ⟹ DF_SKIPS α(P) ✓s(P) ⊑_F Q› for Q
        proof (unfold DF_SKIPS-def, induct arbitrary: Q rule: cont-parallel-fix-ind)
          show ‹cont (λx. (□a ∈ UNIV → x) □ SKIPS UNIV)› by simp
        next
          show ‹cont (λx. (□a ∈ α(P) → x) □ SKIPS ✓s(P))› by simp
        next
          show ‹adm (λx. ∀ Q. Q ∈ R_proc P ⟶ fst x ⊑_F Q ⟶ snd x ⊑_F Q)› by simp
        next
          show ‹⋀Q. ⊥ ⊑_F Q› by simp
        next
          fix x y Q assume hyp : ‹Q ∈ R_proc P ⟹ x ⊑_F Q ⟹ y ⊑_F Q› for Q
          assume ‹Q ∈ R_proc P› and ‹(□a ∈ UNIV → x) □ SKIPS UNIV ⊑_F Q›
          consider ‹{a. ev a ∈ Q⁰} = {}› ‹{r. ✓(r) ∈ Q⁰} = {}›
            | ‹{a. ev a ∈ Q⁰} = {}› ‹{r. ✓(r) ∈ Q⁰} ≠ {}›
            | ‹{a. ev a ∈ Q⁰} ≠ {}› by blast

          thus ‹(□a ∈ α(P) → y) □ SKIPS ✓s(P) ⊑_F Q›
          proof cases
            assume ‹{a. ev a ∈ Q⁰} = {}› ‹{r. ✓(r) ∈ Q⁰} = {}›
          hence ‹Q⁰ = {}› by (metis Collect-empty-eq all-not-in-conv event_ptick.exhaust)
            hence ‹Q = STOP› by (simp add: initials-empty-iff-STOP)
            with ‹(□a ∈ UNIV → x) □ SKIPS UNIV ⊑_F Q› have False
```

by (*auto simp add*: *failure-refine-def Process-eq-spec*
F-STOP F-Ndet F-Mndetprefix′ F-SKIPS)
thus ‹$(\sqcap a \in \alpha(P) \to y) \sqcap SKIPS\ \checkmark s(P) \sqsubseteq_F Q$› **..**
next
assume ‹$\{a.\ ev\ a \in Q^0\} = \{\}$› ‹$\{r.\ \checkmark(r) \in Q^0\} \neq \{\}$›
have ‹$SKIPS\ \{r.\ \checkmark(r) \in Q^0\} \sqsubseteq_F Q$›
proof (*unfold failure-refine-def*, *safe*)
show ‹$(t,\ X) \in \mathcal{F}\ (SKIPS\ \{r.\ \checkmark(r) \in Q^0\})$› **if** ‹$(t,\ X) \in \mathcal{F}\ Q$› **for** $t\ X$
proof (*cases t*)
assume ‹$t = []$›
with ‹$(t,\ X) \in \mathcal{F}\ Q$› **have** ‹$([],\ X \cup -\ Q^0) \in \mathcal{F}\ Q$›
by (*metis ComplD initials-memI is-processT5-S7′ self-append-conv2*)
with ‹$t = []$› ‹$(\sqcap a \in UNIV \to x) \sqcap SKIPS\ UNIV \sqsubseteq_F Q$› ‹$\{a.\ ev\ a \in Q^0\} = \{\}$›
show ‹$(t,\ X) \in \mathcal{F}\ (SKIPS\ \{r.\ \checkmark(r) \in Q^0\})$›
by (*simp add*: *failure-refine-def F-Ndet F-Mndetprefix′ F-SKIPS subset-iff*)
(*metis Compl-iff Un-iff neq-Nil-conv*)
next
from ‹$(\sqcap a \in UNIV \to x) \sqcap SKIPS\ UNIV \sqsubseteq_F Q$› ‹$\{a.\ ev\ a \in Q^0\} = \{\}$›
‹$(t,\ X) \in \mathcal{F}\ Q$› *F-T initials-memI*
show ‹$t = e\ \#\ t' \Longrightarrow (t,\ X) \in \mathcal{F}\ (SKIPS\ \{r.\ \checkmark(r) \in Q^0\})$› **for** $e\ t'$
by (*simp add*: *failure-refine-def F-Ndet F-Mndetprefix′ F-SKIPS*) *blast*
qed
qed
moreover from ‹$Q \in \mathcal{R}_{proc}\ P$› ‹$\{r.\ \checkmark(r) \in Q^0\} \neq \{\}$› *initial-tick-imp-in-ticks-of*
ticks-of-reachable-processes-subset
have ‹$SKIPS\ \checkmark s(P) \sqsubseteq_F SKIPS\ \{r.\ \checkmark(r) \in Q^0\}$›
by (*force simp add*: *SKIPS-F-SKIPS-iff* ‹$\checkmark s(P) \neq \{\}$›)
ultimately show ‹$(\sqcap a \in \alpha(P) \to y) \sqcap SKIPS\ \checkmark s(P) \sqsubseteq_F Q$›
using *Ndet-F-self-right trans-F* **by** *blast*
next
assume ‹$\{a.\ ev\ a \in Q^0\} \neq \{\}$›
from ‹$Q \in \mathcal{R}_{proc}\ P$› *initial-tick-imp-in-ticks-of ticks-of-reachable-processes-subset*
have ‹$\{r.\ \checkmark(r) \in Q^0\} \subseteq \checkmark s(P)$› **by** *force*
from ‹$Q \in \mathcal{R}_{proc}\ P$› *events-of-reachable-processes-subset initial-ev-imp-in-events-of*
have ‹$(\sqcap a \in \alpha(P) \to y) \sqcap SKIPS\ \checkmark s(P) \sqsubseteq_F (\sqcap a \in \{a.\ ev\ a \in Q^0\} \to y) \sqcap SKIPS\ \checkmark s(P)$›
by (*force intro*: *mono-Ndet-F*[*OF Mndetprefix-F-subset*[*OF* ‹$\{a.\ ev\ a \in Q^0\} \neq \{\}$›] *idem-F*])
moreover have ‹$\ldots \sqsubseteq_F (\sqcap a \in \{a.\ ev\ a \in Q^0\} \to Q\ after\ a) \sqcap SKIPS\ \checkmark s(P)$›
proof (*rule mono-Ndet-F*[*OF mono-Mndetprefix-F idem-F*], *clarify*)
show ‹$ev\ a \in Q^0 \Longrightarrow y \sqsubseteq_F Q\ after\ a$› **for** a
by (*frule mono-After-F*[*OF* - ‹$(\sqcap a \in UNIV \to x) \sqcap SKIPS\ UNIV \sqsubseteq_F Q$›])
(*simp add*: *After-Ndet initials-Mndetprefix image-iff*

After-Mndetprefix reachable-after ‹$Q \in \mathcal{R}_{proc}\ P$› *hyp*)
qed
moreover have ‹$\ldots \sqsubseteq_F\ Q$›
proof (*unfold failure-refine-def*, *safe*)
show ‹$(t,\ X) \in \mathcal{F}\ ((\sqcap a \in \{a.\ ev\ a \in Q^0\} \rightarrow Q\ after\ a) \sqcap SKIPS\ \checkmark s(P))$› **if** ‹$(t,\ X) \in \mathcal{F}\ Q$› **for** $t\ X$
proof (*cases t*)
assume ‹$t = []$›
with ‹$(t,\ X) \in \mathcal{F}\ Q$› **have** ‹$([],\ X \cup -\ Q^0) \in \mathcal{F}\ Q$›
by (*metis ComplD eq-Nil-appendI initials-memI' is-processT5-S7'*)
with ‹$(\sqcap a \in UNIV \rightarrow x) \sqcap SKIPS\ UNIV \sqsubseteq_F Q$› ‹$\{r.\ \checkmark(r) \in Q^0\} \subseteq \checkmark s(P)$›
show ‹$(t,\ X) \in \mathcal{F}\ ((\sqcap a \in \{a.\ ev\ a \in Q^0\} \rightarrow Q\ after\ a) \sqcap SKIPS\ \checkmark s(P))$›
by (*simp add*: *failure-refine-def F-Ndet F-Mndetprefix'* ‹$t = []$› *F-SKIPS* ‹$\checkmark s(P) \neq \{\}$› *subset-iff*)
(*meson ComplI UnI1 UnI2 list.distinct*(*1*))
next
from ‹$(t,\ X) \in \mathcal{F}\ Q$› ‹$(\sqcap a \in UNIV \rightarrow x) \sqcap SKIPS\ UNIV \sqsubseteq_F Q$› ‹$\{r.\ \checkmark(r) \in Q^0\} \subseteq \checkmark s(P)$›
show ‹$t = e\ \#\ t' \Longrightarrow (t,\ X) \in \mathcal{F}\ ((\sqcap a \in \{a.\ ev\ a \in Q^0\} \rightarrow Q\ after\ a) \sqcap SKIPS\ \checkmark s(P))$› **for** $e\ t'$
by (*simp add*: *failure-refine-def F-Ndet F-Mndetprefix' F-SKIPS F-After* ‹$\checkmark s(P) \neq \{\}$› *subset-iff*)
(*metis F-T initials-memI list.distinct*(*1*) *list.inject*)
qed
qed
ultimately show ‹$(\sqcap a \in \alpha(P) \rightarrow y) \sqcap SKIPS\ \checkmark s(P) \sqsubseteq_F Q$› **using** *trans-F* **by** *blast*
qed
qed
thus ‹$deadlock\text{-}free_{SKIPS}\ P \Longrightarrow DF_{SKIPS}\ \alpha(P)\ \checkmark s(P) \sqsubseteq_F P$›
by (*simp add*: $deadlock\text{-}free_{SKIPS}$*-def reachable-self*)
qed
qed
qed
qed

corollary $deadlock\text{-}free_{SKIPS}$*-iff-*$DF_{SKIPS}$*-events-of-ticks-of-leFD* :
‹$deadlock\text{-}free_{SKIPS}\ P \longleftrightarrow (\ if\ \alpha(P) = \{\} \wedge \checkmark s(P) = \{\}\ then\ False$
$else\ \ if\ \checkmark s(P) = \{\}\ then\ DF\ \alpha(P) \sqsubseteq_{FD} P$
$else\ \ if\ \alpha(P) = \{\}\ then\ SKIPS\ \checkmark s(P) \sqsubseteq_{FD} P$
$else\ DF_{SKIPS}\ \alpha(P)\ \checkmark s(P) \sqsubseteq_{FD} P)$›
by (*metis* $deadlock\text{-}free_{SKIPS}$*-iff-*$DF_{SKIPS}$*-events-of-ticks-of-leF D-SKIPS* $deadlock\text{-}free_{SKIPS}$*-FD*
div-free-DF div-free-DF_{SKIPS} *divergence-refine-def failure-divergence-refine-def*)

corollary $deadlock\text{-}free_{SKIPS}$*-iff-*$DF_{SKIPS}$*-strict-events-of-strict-ticks-of-leF* :
‹$deadlock\text{-}free_{SKIPS}\ P \longleftrightarrow (\ if\ \boldsymbol{\alpha}(P) = \{\} \wedge \checkmark \mathbf{s}(P) = \{\}\ then\ False$

else *if* ✓**s**$(P) = \{\}$ *then* $DF\ \boldsymbol{\alpha}(P) \sqsubseteq_F P$
else *if* $\boldsymbol{\alpha}(P) = \{\}$ *then* $SKIPS$ ✓**s**$(P) \sqsubseteq_F P$
else $DF_{SKIPS}\ \boldsymbol{\alpha}(P)$ ✓**s**$(P) \sqsubseteq_F P$⟩
by (*simp add*: *deadlock-free*$_{SKIPS}$-*iff-DF*$_{SKIPS}$-*events-of-ticks-of-leF*
events-of-is-strict-events-of-or-UNIV ticks-of-is-strict-ticks-of-or-UNIV)
(*meson deadlock-free-iff-DF-strict-events-of-leF DF*$_{SKIPS}$-*subset deadlock-free*$_{SKIPS}$-*SKIPS*
deadlock-free$_{SKIPS}$-*def deadlock-free*$_{SKIPS}$-*implies-div-free trans-F*
deadlock-free-implies-div-free failure-divergence-refine-def subset-UNIV)

corollary *deadlock-free*$_{SKIPS}$-*iff-DF*$_{SKIPS}$-*strict-events-of-strict-ticks-of-leFD* :
⟨*deadlock-free*$_{SKIPS}$ $P \longleftrightarrow$ (*if* $\boldsymbol{\alpha}(P) = \{\} \wedge$ ✓**s**$(P) = \{\}$ *then False*
else *if* ✓**s**$(P) = \{\}$ *then* $DF\ \boldsymbol{\alpha}(P) \sqsubseteq_{FD} P$
else *if* $\boldsymbol{\alpha}(P) = \{\}$ *then* $SKIPS$ ✓**s**$(P) \sqsubseteq_{FD} P$
else $DF_{SKIPS}\ \boldsymbol{\alpha}(P)$ ✓**s**$(P) \sqsubseteq_{FD} P$)⟩
by (*metis deadlock-free*$_{SKIPS}$-*iff-DF*$_{SKIPS}$-*events-of-ticks-of-leFD*
deadlock-free$_{SKIPS}$-*iff-DF*$_{SKIPS}$-*strict-events-of-strict-ticks-of-leF*
deadlock-free$_{SKIPS}$-*implies-div-free events-of-is-strict-events-of-or-UNIV*
leFD-imp-leF ticks-of-is-strict-ticks-of-or-UNIV)

8.1.5 Big results

As consequences, we have very powerful results, and especially a "data independence" deadlock freeness theorem.

lemma *deadlock-free-is-right*:
⟨*deadlock-free* $P \longleftrightarrow (\forall\, t \in \mathcal{T}\ P.\ tF\ t \wedge (t,\ \ \ UNIV) \notin \mathcal{F}\ P)$⟩
⟨*deadlock-free* $P \longleftrightarrow (\forall\, t \in \mathcal{T}\ P.\ tF\ t \wedge (t,\ ev\ `\ UNIV) \notin \mathcal{F}\ P)$⟩
proof −
from *deadlock-free-iff-empty-ticks-of-and-deadlock-free*$_{SKIPS}$
deadlock-free$_{SKIPS}$-*is-right tickFree-traces-iff-empty-ticks-of*
show ⟨*deadlock-free* $P \longleftrightarrow (\forall\, t \in \mathcal{T}\ P.\ tF\ t \wedge (t,\ UNIV) \notin \mathcal{F}\ P)$⟩ **by** *blast*
thus ⟨*deadlock-free* $P \longleftrightarrow (\forall\, t \in \mathcal{T}\ P.\ tF\ t \wedge (t,\ ev\ `\ UNIV) \notin \mathcal{F}\ P)$⟩
by (*auto intro*: *is-processT4*)
(*metis After*$_{trace}$.*simps*(*1*) *deadlock-free-After*$_{tick}$-*characterization*
F-imp-R-After$_{trace}$ *deadlock-free-After*$_{trace}$-*characterization*)
qed

end

— We may probably prove *deadlock-free* $P = (\forall\, t{\in}\mathcal{T}\ P.\ tF\ t \wedge (t,\ ev\ `\ \alpha(P)) \notin \mathcal{F}\ P)$

theorem *data-independence-deadlock-free-Sync*:
fixes $P\ Q$ :: ⟨('*a*, '*r*) *process*$_{ptick}$⟩
assumes *df-P* : ⟨*deadlock-free P*⟩ **and** *df-Q* : ⟨*deadlock-free Q*⟩
and *hyp* : ⟨*events-of Q* $\cap\ S = \{\} \vee (\exists\, y.$ *events-of Q* $\cap\ S = \{y\} \wedge$ *events-of P* $\cap\ S \subseteq \{y\}$)⟩
shows ⟨*deadlock-free* $(P\ [\![S]\!]\ Q)$⟩

proof −
have *non-BOT* : ‹$P \neq \bot$› ‹$Q \neq \bot$›
by (*simp-all add*: *BOT-iff-Nil-D deadlock-free-implies-div-free df-P df-Q*)

have *nonempty-events* : ‹*events-of* $P \neq \{\}$› ‹*events-of* $Q \neq \{\}$›
by (*simp-all add*: *df-P df-Q nonempty-events-of-if-deadlock-free*)

obtain *a b* **where** *initial* : ‹*ev* $a \in P^0$› ‹*ev* $b \in Q^0$›
by (*meson deadlock-free-initial-evE df-P df-Q*)

have ‹*ev* $a \in (P \llbracket S \rrbracket Q)^0 \vee$ *ev* $b \in (P \llbracket S \rrbracket Q)^0$›
by (*simp add*: *initials-Sync non-BOT image-iff*)
(*metis events-of-iff-reachable-ev reachable-ev.intros*(*1*)
Int-iff initial empty-iff hyp insert-iff subset-singleton-iff)
hence *nonempty-events-Sync*: ‹$\alpha(P \llbracket S \rrbracket Q) \neq \{\}$›
by (*metis events-of-iff-reachable-ev empty-iff reachable-ev.intros*(*1*))

have $*$: ‹*DF* $(\alpha(P) \cup \alpha(Q)) \sqsubseteq_{FD}$ *DF* $\alpha(P) \llbracket S \rrbracket$ *DF* $\alpha(Q)$›
by (*simp add*: *DF-FD-DF-Sync-DF hyp nonempty-events*)

also have ‹$\ldots \sqsubseteq_{FD} P \llbracket S \rrbracket Q$›
by (*meson deadlock-free-iff-DF-events-of-leFD df-P df-Q mono-Sync-FD*)

ultimately show ‹*deadlock-free* $(P \llbracket S \rrbracket Q)$›
by (*meson DF-Univ-freeness Un-empty nonempty-events*(*1*) *trans-FD*)
qed

lemma *data-independence-deadlock-free-Sync-bis*:
‹$\llbracket$*deadlock-free P*; *deadlock-free Q*; $\alpha(Q) \cap S = \{\}\rrbracket \Longrightarrow$
deadlock-free $(P \llbracket S \rrbracket Q)$› **for** P :: ‹$('a, 'r)$ $process_{ptick}$›
by (*simp add*: *data-independence-deadlock-free-Sync*)

We can't expect much better without hypothesis on the processes P and Q. We can easily build the following counter example.

lemma ‹$\exists P\ Q\ S.$ *deadlock-free P* $\wedge$ *deadlock-free Q* $\wedge$
$(\exists y\ z.$ *events-of Q* $\cap S \subseteq \{y, z\} \wedge$ *events-of P* $\cap S \subseteq \{y, z\}) \wedge$
$\neg$ *deadlock-free* $(P \llbracket S ::$ *nat set*$\rrbracket Q)$›
proof (*intro exI conjI*)
show ‹*deadlock-free* (*DF* $\{0\}$)› **and** ‹*deadlock-free* (*DF* $\{Suc\ 0\}$)›
by (*metis DF-Univ-freeness empty-not-insert idem-FD*)+
next
show ‹*events-of* (*DF* $\{Suc\ 0\}) \cap \{0, Suc\ 0\} \subseteq \{0, Suc\ 0\}$›
and ‹*events-of* (*DF* $\{0\}) \cap \{0, Suc\ 0\} \subseteq \{0, Suc\ 0\}$› **by** *simp-all*
next
have ‹*DF* $\{0\} \llbracket\{0, Suc\ 0\}\rrbracket$ *DF* $\{Suc\ 0\} = STOP$›
by (*simp add*: *initials-empty-iff-STOP*[*symmetric*]
initials-Sync initials-DF BOT-iff-Nil-D div-free-DF)
from *this*[*THEN arg-cong*[**where** $f = deadlock\text{-}free$]]
show ‹$\neg$ *deadlock-free* (*DF* $\{0\} \llbracket\{0, Suc\ 0\}\rrbracket$ *DF* $\{Suc\ 0\})$›

by (*simp add*: *non-deadlock-free-STOP*)
qed

end

find-theorems *name*: *data-independence-deadlock-free-Sync-bis*

— Think about a *deadlock-free*$_{SKIPS}$ version.

8.1.6 Results with other references Processes

RUN **and** *non-terminating*

lemma *non-terminating-STOP* [*simp*] : ‹*non-terminating STOP*›
unfolding *non-terminating-def* **by** *simp*

lemma *not-non-terminating-SKIP* [*simp*]: ‹¬ *non-terminating* (*SKIP r*)›
unfolding *non-terminating-is-right* **by** (*simp add*: *T-SKIP*)

lemma *not-non-terminating-BOT* [*simp*] : ‹¬ *non-terminating* ⊥›
unfolding *non-terminating-is-right T-BOT*
using *front-tickFree-single non-tickFree-tick* **by** (*metis mem-Collect-eq*)

context *AfterExt* **begin**

lemma *non-terminating-iff-RUN-events-T*:
‹*non-terminating P* ⟷ *RUN* $\alpha(P)$ $\sqsubseteq_T$ *P*› **for** *P* :: ‹('*a*, '*r*) *process*$_{ptick}$›
proof (*intro iffI*)
have *RUN-subset-T*: ‹(*A* :: '*a set*) ⊆ *B* ⟹ *RUN B* $\sqsubseteq_T$ *RUN A*› **for** *A B*
by (*drule RUN-subset-DT*, *unfold trace-divergence-refine-def*, *elim conjE*, *assumption*)
show ‹*RUN* (*events-of P*) $\sqsubseteq_T$ *P* ⟹ *non-terminating P*›
unfolding *non-terminating-def* **by** (*meson RUN-subset-T UNIV-I subsetI trans-T*)
next
assume *non-terminating-P*: ‹*non-terminating P*›
have ‹*Q* ∈ $\mathcal{R}_{proc}$ *P* ⟹ *RUN* (*events-of P*) $\sqsubseteq_T$ *Q*› **for** *Q*
proof (*unfold RUN-def*, *induct rule*: *reachable-processes-fix-ind-T*)
show ‹*cont* (λ*x*. □*e* ∈ *events-of P* → *x*)› **by** *simp*
next
fix *x s*
assume *hyp*: ‹∀ *Q* ∈ $\mathcal{R}_{proc}$ *P*. *x* $\sqsubseteq_T$ *Q*›
show ‹*s* ∈ $\mathcal{T}$ *P* ⟹ *tickFree s* ⟹ □*e* ∈ *events-of P* → *x* $\sqsubseteq_T$ *P after*$_{\mathcal{T}}$ *s*›
proof (*unfold trace-refine-def*, *safe*)
show ‹*s* ∈ $\mathcal{T}$ *P* ⟹ *tickFree s* ⟹ *t* ∈ $\mathcal{T}$ (*P after*$_{\mathcal{T}}$ *s*) ⟹
t ∈ $\mathcal{T}$ (□*e* ∈ *events-of P* → *x*)› **for** *t*
proof (*induct t arbitrary*: *s*)
show ‹[] ∈ $\mathcal{T}$ (□*e*∈*events-of P* → *x*)› **by** *simp*
next

case (*Cons e t*)
from *Cons.prems*(*1*, *2*) **have** ∗ : ‹*P after*$_{\mathcal{T}}$ *s* ∈ $\mathcal{R}_{proc}$ *P*›
by (*auto simp add*: *reachable-processes-is*)
obtain *e*′ **where** ∗∗ : ‹*e* = *ev e*′› ‹*e*′ ∈ *events-of P*›
by (*meson Cons.prems*(*1*, *3*) *initials-memI imageE*
non-terminating-P initials-After$_{trace}$*-subset-events-of set-mp*)
from *Cons.prems T-After*$_{trace}$ ∗∗(*1*)
have ‹*t* ∈ $\mathcal{T}$ (*P after*$_{\mathcal{T}}$ (*s* @ [*e*]))› **by** (*simp add*: *T-After*$_{trace}$*-eq*)
also have ‹. . . ⊆ $\mathcal{T}$ *x*›
apply (*rule hyp*[*rule-format*, *unfolded trace-refine-def*,
of ‹*P after*$_{\mathcal{T}}$ (*s* @ [*e*])›, *simplified*])
apply (*simp add*: *After*$_{trace}$*-snoc* ∗∗(*1*) *After*$_{tick}$*-def*)
apply (*rule reachable-after*[*OF* ∗])
using *Cons.prems*(*3*) *initials-memI F-T* ∗∗(*1*) **by** *blast*
finally have ‹*t* ∈ $\mathcal{T}$ *x*› **.**
with ∗∗ **show** *?case* **by** (*auto simp add*: *T-Mprefix*)
qed
qed
qed
with *reachable-self* **show** ‹*RUN* α(*P*) ⊑$_T$ *P*› **by** *blast*
qed

lemma *lifelock-free-F*: ‹*lifelock-free P* ⟷ *CHAOS UNIV* ⊑$_F$ *P*›
by (*simp add*: *lifelock-free-is-non-terminating non-terminating-F*)

end

Chapter 9

Bonus: powerful new Laws

9.1 Powerful Results about *Sync*

lemma *add-complementary-events-of-in-failure*:
‹$(t, X) \in \mathcal{F}\ P \Longrightarrow (t, X \cup ev\ `\ (-\ \alpha(P))) \in \mathcal{F}\ P$›
by (*erule is-processT5*) (*auto simp add*: *events-of-def*, *metis F-T in-set-conv-decomp*)

lemma *add-complementary-initials-in-refusal*: ‹$X \in \mathcal{R}\ P \Longrightarrow X \cup -\ P^0 \in \mathcal{R}\ P$›
unfolding *Refusals-iff* **by** (*erule is-processT5*) (*auto simp add*: *initials-def F-T*)

lemma *TickRightSync*:
‹$\checkmark(r) \in S \Longrightarrow ftF\ u \Longrightarrow t\ setinterleaves\ ((u, [\checkmark(r)]), S) \Longrightarrow t = u \wedge last\ u = \checkmark(r)$›
by (*simp add*: *TickLeftSync setinterleaving-sym*)

theorem *Sync-is-Sync-restricted-superset-events*:
fixes $S\ A :: \langle 'a\ set\rangle$ **and** $P\ Q :: \langle('a, 'r)\ process_{ptick}\rangle$
assumes *superset* : ‹$\alpha(P) \cup \alpha(Q) \subseteq A$›
defines ‹$S' \equiv S \cap A$›
shows ‹$P\ [\![S]\!]\ Q = P\ [\![S']\!]\ Q$›
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹$s \in \mathcal{D}\ (P\ [\![S]\!]\ Q) \Longrightarrow s \in \mathcal{D}\ (P\ [\![S']\!]\ Q)$› **for** *s*
by (*simp add*: *D-Sync S'-def*)
(*metis Un-UNIV-left Un-commute equals0D events-of-is-strict-events-of-or-UNIV inf-top.right-neutral sup.orderE superset*)
next
show ‹$s \in \mathcal{D}\ (P\ [\![S']\!]\ Q) \Longrightarrow s \in \mathcal{D}\ (P\ [\![S]\!]\ Q)$› **for** *s*
by (*simp add*: *D-Sync S'-def*)
(*metis Un-UNIV-left Un-commute equals0D events-of-is-strict-events-of-or-UNIV inf-top.right-neutral sup.orderE superset*)
next
let *?S* = ‹$range\ tick \cup ev\ `\ S :: ('a, 'r)\ event_{ptick}\ set$›
and *?S'* = ‹$range\ tick \cup ev\ `\ S' :: ('a, 'r)\ event_{ptick}\ set$›

fix s X
assume *same-div* : $\langle \mathcal{D}\ (P\ [\![S]\!]\ Q) = \mathcal{D}\ (P\ [\![S']\!]\ Q)\rangle$
assume $\langle (s,\ X) \in \mathcal{F}\ (P\ [\![S']\!]\ Q)\rangle$
then consider $\langle s \in \mathcal{D}\ (P\ [\![S']\!]\ Q)\rangle$
| *s-P X-P s-Q X-Q* **where** $\langle (rev\ s\text{-}P,\ X\text{-}P) \in \mathcal{F}\ P\rangle$ $\langle (rev\ s\text{-}Q,\ X\text{-}Q) \in \mathcal{F}\ Q\rangle$
$\langle s\ setinterleaves\ ((rev\ s\text{-}P,\ rev\ s\text{-}Q),\ ?S')\rangle$
$\langle X = (X\text{-}P \cup X\text{-}Q) \cap\ ?S' \cup X\text{-}P \cap X\text{-}Q\rangle$
by (*simp add*: *F-Sync D-Sync*) (*metis rev-rev-ident*)
thus $\langle (s,\ X) \in \mathcal{F}\ (P\ [\![S]\!]\ Q)\rangle$
proof *cases*
from *same-div D-F* **show** $\langle s \in \mathcal{D}\ (P\ [\![S']\!]\ Q) \Longrightarrow (s,\ X) \in \mathcal{F}\ (P\ [\![S]\!]\ Q)\rangle$ **by** *blast*
next
fix *s-P s-Q* **and** *X-P X-Q* :: $\langle ('a,\ 'r)\ event_{ptick}\ set\rangle$
let *?X-P'* = $\langle X\text{-}P \cup ev\ `\ (-\ \alpha(P))\rangle$ **and** *?X-Q'* = $\langle X\text{-}Q \cup ev\ `\ (-\ \alpha(Q))\rangle$
assume *assms* : $\langle (rev\ s\text{-}P,\ X\text{-}P) \in \mathcal{F}\ P\rangle$ $\langle (rev\ s\text{-}Q,\ X\text{-}Q) \in \mathcal{F}\ Q\rangle$
$\langle s\ setinterleaves\ ((rev\ s\text{-}P,\ rev\ s\text{-}Q),\ ?S')\rangle$
$\langle X = (X\text{-}P \cup X\text{-}Q) \cap\ ?S' \cup X\text{-}P \cap X\text{-}Q\rangle$

from *assms*(*1*, *2*)[*THEN F-T*] **and** *assms*(*3*)
have *assms-3-bis* : $\langle s\ setinterleaves\ ((rev\ s\text{-}P,\ rev\ s\text{-}Q),\ ?S)\rangle$
proof (*induct* $\langle (s\text{-}P,\ ?S,\ s\text{-}Q)\rangle$ *arbitrary*: *s-P s-Q s rule*: *setinterleaving.induct*)
case *1*
thus $\langle s\ setinterleaves\ ((rev\ [],\ rev\ []),\ ?S)\rangle$ **by** *simp*
next
case (*2 y s-Q*)
from *2.prems*(*3*)[*THEN doubleReverse*] **obtain** s' b
where $*$: $\langle y = ev\ b\rangle$ $\langle b \notin S'\rangle$ $\langle s = rev\ (y\ \#\ s')\rangle$
$\langle s'\ setinterleaves\ (([],\ s\text{-}Q),\ ?S')\rangle$
by (*simp add*: *image-iff split*: *if-split-asm*) (*metis* $event_{ptick}$.*exhaust*)
from *2.prems*(*2*)[*unfolded* $\langle y = ev\ b\rangle$]
have $\langle b \in \alpha(Q)\rangle$ **by** (*force simp add*: *events-of-def*)
with $\langle b \notin S'\rangle$ *superset* **have** $\langle b \notin S\rangle$ **by** (*simp add*: *S'-def subset-iff*)
from *2.prems*(*2*)[*simplified*, *THEN is-processT3-TR*] **have** $\langle rev\ s\text{-}Q \in \mathcal{T}\ Q\rangle$ **by** *auto*
have $\langle y \notin ?S\rangle$ **by** (*simp add*: $*$(*1*) $\langle b \notin S\rangle$ *image-iff*)
have $\langle rev\ s'\ setinterleaves\ ((rev\ [],\ rev\ s\text{-}Q),\ ?S)\rangle$
by (*fact 2.hyps*[*OF* $\langle y \notin ?S\rangle$ *2.prems*(*1*) $\langle rev\ s\text{-}Q \in \mathcal{T}\ Q\rangle$ $*$(*4*)[*THEN doubleReverse*]])
from *this*[*THEN addNonSync*, *OF* $\langle y \notin ?S\rangle$]
show $\langle s\ setinterleaves\ ((rev\ [],\ rev\ (y\ \#\ s\text{-}Q)),\ ?S)\rangle$
by (*simp add*: $\langle s = rev\ (y\ \#\ s')\rangle$)
next
case (*3 x s-P*)
from *3.prems*(*3*)[*THEN doubleReverse*] **obtain** s' a
where $*$: $\langle x = ev\ a\rangle$ $\langle a \notin S'\rangle$ $\langle s = rev\ (x\ \#\ s')\rangle$
$\langle s'\ setinterleaves\ ((s\text{-}P,\ []),\ ?S')\rangle$
by (*simp add*: *image-iff split*: *if-split-asm*) (*metis* $event_{ptick}$.*exhaust*)
from *3.prems*(*1*)[*unfolded* $\langle x = ev\ a\rangle$]

have ‹$a \in \alpha(P)$› **by** (*force simp add*: *events-of-def*)
with ‹$a \notin S'$› *superset* **have** ‹$a \notin S$› **by** (*simp add*: *S'-def subset-iff*)
from *3.prems(1)*[*simplified*, *THEN is-processT3-TR*] **have** ‹*rev s-P* $\in \mathcal{T}$ *P*› **by** *auto*
have ‹$x \notin$ *?S*› **by** (*simp add*: $*(1)$ ‹$a \notin S$› *image-iff*)
have ‹*rev s'* *setinterleaves* ((*rev s-P*, *rev* []), *?S*)›
by (*fact 3.hyps*[*OF* ‹$x \notin$ *?S*› ‹*rev s-P* $\in \mathcal{T}$ *P*› *3.prems(2)* $*(4)$[*THEN doubleReverse*]])
from *this*[*THEN addNonSync*, *OF* ‹$x \notin$ *?S*›]
show ‹*s* *setinterleaves* ((*rev* (x # *s-P*), *rev* []), *?S*)›
by (*simp add*: ‹$s = rev\ (x \# s')$›)
next
case (*4 x s-P y s-Q*)
from *4.prems(1)*[*simplified*, *THEN is-processT3-TR*] **have** ‹*rev s-P* $\in \mathcal{T}$ *P*› **by** *auto*
from *4.prems(2)*[*simplified*, *THEN is-processT3-TR*] **have** ‹*rev s-Q* $\in \mathcal{T}$ *Q*› **by** *auto*
from *4.prems(1)* **have** ‹$x \in$ *range tick* $\cup$ *ev* ‘ $\alpha(P)$›
by (*cases x*; *force simp add*: *events-of-def image-iff split*: $event_{ptick}.split$)
from *4.prems(2)* **have** ‹$y \in$ *range tick* $\cup$ *ev* ‘ $\alpha(Q)$›
by (*cases y*; *force simp add*: *events-of-def image-iff split*: $event_{ptick}.split$)
consider ‹$x \in$ *?S*› **and** ‹$y \in$ *?S*› | ‹$x \in$ *?S*› **and** ‹$y \notin$ *?S*›
| ‹$x \notin$ *?S*› **and** ‹$y \in$ *?S*› | ‹$x \notin$ *?S*› **and** ‹$y \notin$ *?S*› **by** *blast*
thus ‹*s* *setinterleaves* ((*rev* (x # *s-P*), *rev* (y # *s-Q*)), *?S*)›
proof *cases*
assume ‹$x \in$ *?S*› **and** ‹$y \in$ *?S*›
with ‹$x \in$ *range tick* $\cup$ *ev* ‘ $\alpha(P)$› ‹$y \in$ *range tick* $\cup$ *ev* ‘ $\alpha(Q)$› *superset*
have ‹$x \in$ *?S'*› **and** ‹$y \in$ *?S'*› **by** (*auto simp add*: *S'-def*)
with *4.prems(3)*[*THEN doubleReverse*] **obtain** s'
where $* :$ ‹$x = y$› ‹$s = rev\ (x \# s')$› ‹s' *setinterleaves* ((*s-P*, *s-Q*), *?S'*)›
by (*simp split*: *if-split-asm*) *blast*

from *4.hyps(1)*[*OF* ‹$x \in$ *?S*› ‹$y \in$ *?S*› ‹$x = y$› ‹*rev s-P* $\in \mathcal{T}$ *P*› ‹*rev s-Q* $\in \mathcal{T}$ *Q*›
‹s' *setinterleaves* ((*s-P*, *s-Q*), *?S'*)›[*THEN doubleReverse*]]
have ‹*rev s'* *setinterleaves* ((*rev s-P*, *rev s-Q*), *?S*)› **.**
from *this*[*THEN doubleReverse*] ‹$x \in$ *?S*›
have ‹(x # s') *setinterleaves* ((x # *s-P*, x # *s-Q*), *?S*)› **by** *simp*
from *this*[*THEN doubleReverse*] **show** *?case* **by** (*simp add*: $*(1, 2)$)
next
assume ‹$x \in$ *?S*› **and** ‹$y \notin$ *?S*›
with ‹$x \in$ *range tick* $\cup$ *ev* ‘ $\alpha(P)$› ‹$y \in$ *range tick* $\cup$ *ev* ‘ $\alpha(Q)$› *superset*
have ‹$x \in$ *?S'*› **and** ‹$y \notin$ *?S'*› **by** (*auto simp add*: *S'-def*)
with *4.prems(3)*[*THEN doubleReverse*, *simplified*] **obtain** s'
where $* :$ ‹$s = rev\ (y \# s')$› ‹s' *setinterleaves* ((x # *s-P*, *s-Q*), *?S'*)›
by (*simp split*: *if-split-asm*) *blast*
from *4.hyps(2)*[*OF* ‹$x \in$ *?S*› ‹$y \notin$ *?S*› ‹*rev* (x # *s-P*) $\in \mathcal{T}$ *P*› ‹*rev s-Q* $\in \mathcal{T}$ *Q*›
‹s' *setinterleaves* ((x # *s-P*, *s-Q*), *?S'*)›[*THEN doubleReverse*]]

have ‹*rev s′ setinterleaves* ((*rev* (*x* # *s-P*), *rev s-Q*), *?S*)› **.**
from *this*[*THEN doubleReverse*] ‹*x* ∈ *?S*› ‹*y* ∉ *?S*›
have ‹(*y* # *s′*) *setinterleaves* ((*x* # *s-P*, *y* # *s-Q*), *?S*)› **by** *simp*
from *this*[*THEN doubleReverse*] **show** *?case* **by** (*simp add*: ‹*s* = *rev* (*y* # *s′*)›)
next
assume ‹*x* ∉ *?S*› **and** ‹*y* ∈ *?S*›
with ‹*x* ∈ *range tick* ∪ *ev* ‘ *α*(*P*)› ‹*y* ∈ *range tick* ∪ *ev* ‘ *α*(*Q*)› *superset*
have ‹*x* ∉ *?S′*› **and** ‹*y* ∈ *?S′*› **by** (*auto simp add*: *S′-def*)
with *4*.*prems*(*3*)[*THEN doubleReverse*] **obtain** *s′*
where ∗ : ‹*s* = *rev* (*x* # *s′*)› ‹*s′ setinterleaves* ((*s-P*, *y* # *s-Q*), *?S′*)›
by (*simp split*: *if-split-asm*) *blast*
from *4*.*hyps*(*5*)[*OF* ‹*x* ∉ *?S*› - ‹*rev s-P* ∈ *T P*› ‹*rev* (*y* # *s-Q*) ∈ *T Q*›
‹*s′ setinterleaves* ((*s-P*, *y* # *s-Q*), *?S′*)›[*THEN doubleReverse*]]
‹*y* ∈ *?S*›
have ‹*rev s′ setinterleaves* ((*rev s-P*, *rev* (*y* # *s-Q*)), *?S*)› **by** *simp*
from *this*[*THEN doubleReverse*] ‹*x* ∉ *?S*› ‹*y* ∈ *?S*›
have ‹(*x* # *s′*) *setinterleaves* ((*x* # *s-P*, *y* # *s-Q*), *?S*)› **by** *simp*
from *this*[*THEN doubleReverse*] **show** *?case* **by** (*simp add*: ‹*s* = *rev* (*x* # *s′*)›)
next
assume ‹*x* ∉ *?S*› **and** ‹*y* ∉ *?S*›
with ‹*x* ∈ *range tick* ∪ *ev* ‘ *α*(*P*)› ‹*y* ∈ *range tick* ∪ *ev* ‘ *α*(*Q*)›
have ‹*x* ∉ *?S′*› **and** ‹*y* ∉ *?S′*› **by** (*auto simp add*: *S′-def*)
with *4*.*prems*(*3*)[*THEN doubleReverse*, *simplified*] **consider**
s′ **where** ‹*s* = *rev* (*x* # *s′*)› ‹*s′ setinterleaves* ((*s-P*, *y* # *s-Q*), *?S′*)›
| *s′* **where** ‹*s* = *rev* (*y* # *s′*)› ‹*s′ setinterleaves* ((*x* # *s-P*, *s-Q*), *?S′*)›
by (*simp split*: *if-split-asm*) *blast*
thus *?case*
proof *cases*
fix *s′* **assume** ‹*s* = *rev* (*x* # *s′*)› ‹*s′ setinterleaves* ((*s-P*, *y* # *s-Q*), *?S′*)›
from *4*.*hyps*(*3*)[*OF* ‹*x* ∉ *?S*› ‹*y* ∉ *?S*› ‹*rev s-P* ∈ *T P*› ‹*rev* (*y* # *s-Q*) ∈ *T Q*›
‹*s′ setinterleaves* ((*s-P*, *y* # *s-Q*), *?S′*)›[*THEN doubleReverse*]]
have ‹*rev s′ setinterleaves* ((*rev s-P*, *rev* (*y* # *s-Q*)), *?S*)› **.**
from *this*[*THEN doubleReverse*] ‹*x* ∉ *?S*› ‹*y* ∉ *?S*›
have ‹(*x* # *s′*) *setinterleaves* ((*x* # *s-P*, *y* # *s-Q*), *?S*)› **by** *simp*
from *this*[*THEN doubleReverse*] **show** *?case* **by** (*simp add*: ‹*s* = *rev* (*x* # *s′*)›)
next
fix *s′* **assume** ‹*s* = *rev* (*y* # *s′*)› ‹*s′ setinterleaves* ((*x* # *s-P*, *s-Q*), *?S′*)›
from *4*.*hyps*(*4*)[*OF* ‹*x* ∉ *?S*› ‹*y* ∉ *?S*› ‹*rev* (*x* # *s-P*) ∈ *T P*› ‹*rev s-Q* ∈ *T Q*›
‹*s′ setinterleaves* ((*x* # *s-P*, *s-Q*), *?S′*)›[*THEN doubleReverse*]]
have ‹*rev s′ setinterleaves* ((*rev* (*x* # *s-P*), *rev s-Q*), *?S*)› **.**
from *this*[*THEN doubleReverse*] ‹*x* ∉ *?S*› ‹*y* ∉ *?S*›
have ‹(*y* # *s′*) *setinterleaves* ((*x* # *s-P*, *y* # *s-Q*), *?S*)› **by** *simp*
from *this*[*THEN doubleReverse*] **show** *?case* **by** (*simp add*: ‹*s* = *rev* (*y* # *s′*)›)

```
      qed
    qed
  qed

  from add-complementary-events-of-in-failure[OF assms(1)]
  have ‹(rev s-P, ?X-P') ∈ ℱ P› .
  moreover from add-complementary-events-of-in-failure[OF assms(2)]
  have ‹(rev s-Q, ?X-Q') ∈ ℱ Q› .
  ultimately have ‹(s, (?X-P' ∪ ?X-Q') ∩ ?S ∪ ?X-P' ∩ ?X-Q') ∈ ℱ (P ⟦S⟧
Q)›
    using assms-3-bis by (simp add: F-Sync) blast
  moreover have ‹X ⊆ (?X-P' ∪ ?X-Q') ∩ ?S ∪ ?X-P' ∩ ?X-Q'›
    by (auto simp add: assms(4) S'-def image-iff)
  ultimately show ‹(s, X) ∈ ℱ (P ⟦S⟧ Q)› by (rule is-processT4)
 qed
next
 let ?S = ‹range tick ∪ ev ‘ S :: ('a, 'r) event_ptick set›
 and ?S' = ‹range tick ∪ ev ‘ S' :: ('a, 'r) event_ptick set›
 fix s X
 assume same-div : ‹𝒟 (P ⟦S⟧ Q) = 𝒟 (P ⟦S'⟧ Q)›
 assume ‹(s, X) ∈ ℱ (P ⟦S⟧ Q)›
 then consider ‹s ∈ 𝒟 (P ⟦S⟧ Q)›
  | s-P X-P s-Q X-Q where ‹(rev s-P, X-P) ∈ ℱ P› ‹(rev s-Q, X-Q) ∈ ℱ Q›
                          ‹s setinterleaves ((rev s-P, rev s-Q), ?S)›
                          ‹X = (X-P ∪ X-Q) ∩ ?S ∪ X-P ∩ X-Q›
  by (simp add: F-Sync D-Sync) (metis rev-rev-ident)
 thus ‹(s, X) ∈ ℱ (P ⟦S'⟧ Q)›
 proof cases
   from same-div D-F show ‹s ∈ 𝒟 (P ⟦S⟧ Q) ⟹ (s, X) ∈ ℱ (P ⟦S'⟧ Q)› by
blast
 next
  fix s-P s-Q and X-P X-Q :: ‹('a, 'r) event_ptick set›
  let ?X-P' = ‹X-P ∪ ev ‘ (− α(P))› and ?X-Q' = ‹X-Q ∪ ev ‘ (− α(Q))›
  assume assms : ‹(rev s-P, X-P) ∈ ℱ P› ‹(rev s-Q, X-Q) ∈ ℱ Q›
                 ‹s setinterleaves ((rev s-P, rev s-Q), ?S)›
                 ‹X = (X-P ∪ X-Q) ∩ ?S ∪ X-P ∩ X-Q›

  from assms(1, 2)[THEN F-T] and assms(3)
  have assms-3-bis : ‹s setinterleaves ((rev s-P, rev s-Q), ?S')›
  proof (induct ‹(s-P, ?S', s-Q)› arbitrary: s-P s-Q s rule: setinterleaving.induct)
    case 1
    thus ‹s setinterleaves ((rev [], rev []), ?S')› by simp
  next
    case (2 y s-Q)
    from 2.prems(3)[THEN doubleReverse] obtain s' b
      where * : ‹y = ev b› ‹b ∉ S› ‹s = rev (y # s')›
                ‹s' setinterleaves (([], s-Q), ?S)›
      by (simp add: image-iff split: if-split-asm) (metis event_ptick.exhaust)
    from 2.prems(2)[unfolded ‹y = ev b›]
```

have ‹$b \in \alpha(Q)$› **by** (*force simp add*: *events-of-def*)
with ‹$b \notin S$› **have** ‹$b \notin S'$› **by** (*simp add*: S'*-def*)
from *2.prems(2)*[*simplified*, *THEN is-processT3-TR*] **have** ‹*rev s-Q* $\in \mathcal{T}$ *Q*› **by** *auto*
have ‹$y \notin$ *?S'*› **by** (*simp add*: $*(1)$ ‹$b \notin S'$› *image-iff*)
have ‹*rev* s' *setinterleaves* ((*rev* [], *rev s-Q*), *?S'*)›
by (*fact 2.hyps*[*OF* ‹$y \notin$ *?S'*› *2.prems(1)* ‹*rev s-Q* $\in \mathcal{T}$ *Q*› $*(4)$[*THEN doubleReverse*]])
from *this*[*THEN addNonSync*, *OF* ‹$y \notin$ *?S'*›]
show ‹*s setinterleaves* ((*rev* [], *rev* (y # *s-Q*)), *?S'*)›
by (*simp add*: ‹s = *rev* (y # s')›)
next
case (*3 x s-P*)
from *3.prems(3)*[*THEN doubleReverse*] **obtain** s' a
where $*$: ‹x = *ev* a› ‹$a \notin S$› ‹s = *rev* (x # s')›
‹s' *setinterleaves* ((*s-P*, []), *?S*)›
by (*simp add*: *image-iff split*: *if-split-asm*) (*metis* $event_{ptick}$*.exhaust*)
from *3.prems(1)*[*unfolded* ‹x = *ev* a›]
have ‹$a \in \alpha(P)$› **by** (*force simp add*: *events-of-def*)
with ‹$a \notin S$› **have** ‹$a \notin S'$› **by** (*simp add*: S'*-def*)
from *3.prems(1)*[*simplified*, *THEN is-processT3-TR*] **have** ‹*rev s-P* $\in \mathcal{T}$ *P*› **by** *auto*
have ‹$x \notin$ *?S'*› **by** (*simp add*: $*(1)$ ‹$a \notin S'$› *image-iff*)
have ‹*rev* s' *setinterleaves* ((*rev s-P*, *rev* []), *?S'*)›
by (*fact 3.hyps*[*OF* ‹$x \notin$ *?S'*› ‹*rev s-P* $\in \mathcal{T}$ *P*› *3.prems(2)* $*(4)$[*THEN doubleReverse*]])
from *this*[*THEN addNonSync*, *OF* ‹$x \notin$ *?S'*›]
show ‹*s setinterleaves* ((*rev* (x # *s-P*), *rev* []), *?S'*)›
by (*simp add*: ‹s = *rev* (x # s')›)
next
case (*4 x s-P y s-Q*)
from *4.prems(1)*[*simplified*, *THEN is-processT3-TR*] **have** ‹*rev s-P* $\in \mathcal{T}$ *P*› **by** *auto*
from *4.prems(2)*[*simplified*, *THEN is-processT3-TR*] **have** ‹*rev s-Q* $\in \mathcal{T}$ *Q*› **by** *auto*
from *4.prems(1)* **have** ‹$x \in$ *range tick* $\cup$ *ev* ‘ $\alpha(P)$›
by (*cases x*; *force simp add*: *events-of-def image-iff split*: $event_{ptick}$*.split*)
from *4.prems(2)* **have** ‹$y \in$ *range tick* $\cup$ *ev* ‘ $\alpha(Q)$›
by (*cases y*; *force simp add*: *events-of-def image-iff split*: $event_{ptick}$*.split*)
consider ‹$x \in$ *?S'*› **and** ‹$y \in$ *?S'*› | ‹$x \in$ *?S'*› **and** ‹$y \notin$ *?S'*›
| ‹$x \notin$ *?S'*› **and** ‹$y \in$ *?S'*› | ‹$x \notin$ *?S'*› **and** ‹$y \notin$ *?S'*› **by** *blast*
thus ‹*s setinterleaves* ((*rev* (x # *s-P*), *rev* (y # *s-Q*)), *?S'*)›
proof *cases*
assume ‹$x \in$ *?S'*› **and** ‹$y \in$ *?S'*›
with ‹$x \in$ *range tick* $\cup$ *ev* ‘ $\alpha(P)$› ‹$y \in$ *range tick* $\cup$ *ev* ‘ $\alpha(Q)$›
have ‹$x \in$ *?S*› **and** ‹$y \in$ *?S*› **by** (*auto simp add*: S'*-def*)
with *4.prems(3)*[*THEN doubleReverse*] **obtain** s'
where $*$: ‹x = y› ‹s = *rev* (x # s')› ‹s' *setinterleaves* ((*s-P*, *s-Q*), *?S*)›
by (*simp split*: *if-split-asm*) *blast*

```
      from 4.hyps(1)[OF ‹x ∈ ?S'› ‹y ∈ ?S'› ‹x = y› ‹rev s-P ∈ T P› ‹rev s-Q
∈ T Q›
                        ‹s' setinterleaves ((s-P, s-Q), ?S)›[THEN doubleReverse]]
      have ‹rev s' setinterleaves ((rev s-P, rev s-Q), ?S')› .
      from this[THEN doubleReverse] ‹x ∈ ?S'›
      have ‹(x # s') setinterleaves ((x # s-P, x # s-Q), ?S')› by simp
      from this[THEN doubleReverse] show ?case by (simp add: ∗(1, 2))
    next
      assume ‹x ∈ ?S'› and ‹y ∉ ?S'›
      with ‹x ∈ range tick ∪ ev ‘ α(P)› ‹y ∈ range tick ∪ ev ‘ α(Q)› superset
      have ‹x ∈ ?S› and ‹y ∉ ?S› by (auto simp add: S'-def)
      with 4.prems(3)[THEN doubleReverse, simplified] obtain s'
      where ∗ : ‹s = rev (y # s')› ‹s' setinterleaves ((x # s-P, s-Q), ?S)›
        by (simp split: if-split-asm) blast
      from 4.hyps(2)[OF ‹x ∈ ?S'› ‹y ∉ ?S'› ‹rev (x # s-P) ∈ T P› ‹rev s-Q ∈
T Q›
                        ‹s' setinterleaves ((x # s-P, s-Q), ?S)›[THEN doubleReverse]]
      have ‹rev s' setinterleaves ((rev (x # s-P), rev s-Q), ?S')› .
      from this[THEN doubleReverse] ‹x ∈ ?S'› ‹y ∉ ?S'›
      have ‹(y # s') setinterleaves ((x # s-P, y # s-Q), ?S')› by simp
      from this[THEN doubleReverse] show ?case by (simp add: ‹s = rev (y #
s')›)
    next
      assume ‹x ∉ ?S'› and ‹y ∈ ?S'›
      with ‹x ∈ range tick ∪ ev ‘ α(P)› ‹y ∈ range tick ∪ ev ‘ α(Q)› superset
      have ‹x ∉ ?S› and ‹y ∈ ?S› by (auto simp add: S'-def)
      with 4.prems(3)[THEN doubleReverse] obtain s'
      where ∗ : ‹s = rev (x # s')› ‹s' setinterleaves ((s-P, y # s-Q), ?S)›
        by (simp split: if-split-asm) blast
      from 4.hyps(5)[OF ‹x ∉ ?S'› - ‹rev s-P ∈ T P› ‹rev (y # s-Q) ∈ T Q›
                        ‹s' setinterleaves ((s-P, y # s-Q), ?S)›[THEN doubleReverse]]
‹y ∈ ?S'›
      have ‹rev s' setinterleaves ((rev s-P, rev (y # s-Q)), ?S')› by simp
      from this[THEN doubleReverse] ‹x ∉ ?S'› ‹y ∈ ?S'›
      have ‹(x # s') setinterleaves ((x # s-P, y # s-Q), ?S')› by simp
      from this[THEN doubleReverse] show ?case by (simp add: ‹s = rev (x #
s')›)
    next
      assume ‹x ∉ ?S'› and ‹y ∉ ?S'›
      with ‹x ∈ range tick ∪ ev ‘ α(P)› ‹y ∈ range tick ∪ ev ‘ α(Q)› superset
      have ‹x ∉ ?S› and ‹y ∉ ?S› by (auto simp add: S'-def)
      with 4.prems(3)[THEN doubleReverse, simplified] consider
        s' where ‹s = rev (x # s')› ‹s' setinterleaves ((s-P, y # s-Q), ?S)›
      | s' where ‹s = rev (y # s')› ‹s' setinterleaves ((x # s-P, s-Q), ?S)›
        by (simp split: if-split-asm) blast
      thus ?case
      proof cases
        fix s' assume ‹s = rev (x # s')› ‹s' setinterleaves ((s-P, y # s-Q), ?S)›
```

```
      from 4.hyps(3)[OF ‹x ∉ ?S'› ‹y ∉ ?S'› ‹rev s-P ∈ T P› ‹rev (y # s-Q)
∈ T Q›
                        ‹s' setinterleaves ((s-P, y # s-Q), ?S)›[THEN doubleReverse]]
      have ‹rev s' setinterleaves ((rev s-P, rev (y # s-Q)), ?S')› .
      from this[THEN doubleReverse] ‹x ∉ ?S'› ‹y ∉ ?S'›
      have ‹(x # s') setinterleaves ((x # s-P, y # s-Q), ?S')› by simp
      from this[THEN doubleReverse] show ?case by (simp add: ‹s = rev (x #
s')›)
    next
      fix s' assume ‹s = rev (y # s')› ‹s' setinterleaves ((x # s-P, s-Q), ?S)›
      from 4.hyps(4)[OF ‹x ∉ ?S'› ‹y ∉ ?S'› ‹rev (x # s-P) ∈ T P› ‹rev s-Q
∈ T Q›
                        ‹s' setinterleaves ((x # s-P, s-Q), ?S)›[THEN doubleReverse]]
      have ‹rev s' setinterleaves ((rev (x # s-P), rev s-Q), ?S')› .
      from this[THEN doubleReverse] ‹x ∉ ?S'› ‹y ∉ ?S'›
      have ‹(y # s') setinterleaves ((x # s-P, y # s-Q), ?S')› by simp
      from this[THEN doubleReverse] show ?case by (simp add: ‹s = rev (y #
s')›)
    qed
  qed
qed

  from add-complementary-events-of-in-failure[OF assms(1)]
  have ‹(rev s-P, ?X-P') ∈ F P› .
  moreover from add-complementary-events-of-in-failure[OF assms(2)]
  have ‹(rev s-Q, ?X-Q') ∈ F Q› .
  ultimately have ‹(s, (?X-P' ∪ ?X-Q') ∩ ?S' ∪ ?X-P' ∩ ?X-Q') ∈ F (P ⟦S'⟧
Q)›
    using assms-3-bis by (simp add: F-Sync) blast
 moreover from superset have ‹X ⊆ (?X-P' ∪ ?X-Q') ∩ ?S' ∪ ?X-P' ∩ ?X-Q'›
    by (auto simp add: assms(4) S'-def image-iff)
  ultimately show ‹(s, X) ∈ F (P ⟦S'⟧ Q)› by (rule is-processT4)
 qed
qed

corollary Sync-is-Sync-restricted-events : ‹P ⟦S⟧ Q = P ⟦S ∩ (α(P) ∪ α(Q))⟧ Q›
 by (simp add: Sync-is-Sync-restricted-superset-events)
```

This version is closer to the intuition that we may have, but the first one would be more useful if we don't want to compute the events of a process but know a superset approximation.

```
corollary ‹deadlock-free P ⟹ deadlock-free Q ⟹
        S ∩ (α(P) ∪ α(Q)) = {} ⟹ deadlock-free (P ⟦S⟧ Q)›
 by (subst Sync-is-Sync-restricted-events) (simp add: Inter-deadlock-free)
```

9.2 Powerful Results about *Renaming*

In this section we will provide laws about the *Renaming* operator. In the first subsection we will give slight generalizations of previous results, but in the other we prove some very powerful theorems.

9.2.1 Some Generalizations

For *Renaming*, we can obtain generalizations of the following results:
Renaming (*Mprefix A P*) *f g* = $\Box y \in f$ ‘ $A \rightarrow \sqcap a \in \{x \in A.\ y = f\, x\}$. *Renaming* (*P a*) *f g*
Renaming (*Mndetprefix A P*) *f g* = $\sqcap b \in f$ ‘ $A \rightarrow \sqcap a \in \{a \in A.\ b = f\, a\}$. *Renaming* (*P a*) *f g*

lemma *Renaming-Mprefix-Sliding*:
‹*Renaming* (($\Box a \in A \rightarrow P\ a$) $\triangleright$ *Q*) *f g* =
($\Box y \in f$ ‘ $A \rightarrow \sqcap a \in \{x \in A.\ y = f\, x\}$. *Renaming* (*P a*) *f g*) $\triangleright$ *Renaming Q f g*›
unfolding *Sliding-def*
by (*simp add*: *Renaming-Det Renaming-distrib-Ndet Renaming-Mprefix*)

9.2.2 *Renaming* and (\)

When *f* is one to one, *Renaming* (*P* \ *S*) *f* will behave like we expect it to do.

lemma *strict-mono-map*: ‹*strict-mono g* $\Longrightarrow$ *strict-mono* (λi. *map f* (*g i*))›
unfolding *strict-mono-def less-eq-list-def less-list-def prefix-def* **by** *fastforce*

lemma *trace-hide-map-map-event*$_{ptick}$:
‹*inj-on* (*map-event*$_{ptick}$ *f g*) (*set s* $\cup$ *ev* ‘ *S*) $\Longrightarrow$
trace-hide (*map* (*map-event*$_{ptick}$ *f g*) *s*) (*ev* ‘ *f* ‘ *S*) =
map (*map-event*$_{ptick}$ *f g*) (*trace-hide s* (*ev* ‘ *S*))›
proof (*induct s*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons e s*)
hence $*$: ‹*trace-hide* (*map* (*map-event*$_{ptick}$ *f g*) *s*) (*ev* ‘ *f* ‘ *S*) =
map (*map-event*$_{ptick}$ *f g*) (*trace-hide s* (*ev* ‘ *S*))› **by** *fastforce*
from *Cons.prems*[*unfolded inj-on-def*, *rule-format*, *of e*, *simplified*] **show** *?case*
by (*simp add*: $*$, *simp add*: *image-iff split*: *event*$_{ptick}$.*split*)
(*metis event*$_{ptick}$.*simps*(*9*))
qed

theorem *bij-Renaming-Hiding*: ‹*Renaming* (*P* \ *S*) *f g* = *Renaming P f g* \ *f* ‘ *S*›
(**is** ‹*?lhs* = *?rhs*›) **if** *bij-f*: ‹*bij f*› **and** *bij-g* : ‹*bij g*›
proof −
have *inj-on-map-event*$_{ptick}$: ‹*inj-on* (*map-event*$_{ptick}$ *f g*) *X*› **for** *X*
by (*metis bij-betw-imp-inj-on bij-f bij-g event*$_{ptick}$*.inj-map inj-eq inj-onI*)
have *inj-on-map-event*$_{ptick}$*-inv* : ‹*inj-on* (*map-event*$_{ptick}$ (*inv f*) (*inv g*)) *X*› **for** *X*
by (*metis bij-betw-def bij-f bij-g event*$_{ptick}$*.inj-map inj-on-inverseI inv-f-eq surj-imp-inj-inv*)
show ‹*?lhs* = *?rhs*›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹*s* ∈ $\mathcal{D}$ *?lhs*›
then obtain *s1 s2* **where** ∗ : ‹*tF s1*› ‹*ftF s2*›
‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *s2*› ‹*s1* ∈ $\mathcal{D}$ (*P* \ *S*)›
by (*simp add*: *D-Renaming*) *blast*
from ∗(*4*) **obtain** *t u*
where ∗∗ : ‹*ftF u*› ‹*tF t*› ‹*s1* = *trace-hide t* (*ev* ‘ *S*) @ *u*›
‹*t* ∈ $\mathcal{D}$ *P* ∨ (∃ *g*. *isInfHiddenRun g P S* ∧ *t* ∈ *range g*)›
by (*simp add*: *D-Hiding*) *blast*
from ∗∗(*4*) **show** ‹*s* ∈ $\mathcal{D}$ *?rhs*›
proof (*elim disjE*)
assume ‹*t* ∈ $\mathcal{D}$ *P*›
hence ‹*ftF* (*map* (*map-event*$_{ptick}$ *f g*) *u* @ *s2*) ∧ *tF* (*map* (*map-event*$_{ptick}$ *f g*) *t*) ∧
s = *trace-hide* (*map* (*map-event*$_{ptick}$ *f g*) *t*) (*ev* ‘ *f* ‘ *S*) @ *map* (*map-event*$_{ptick}$ *f g*) *u* @ *s2* ∧
map (*map-event*$_{ptick}$ *f g*) *t* ∈ $\mathcal{D}$ (*Renaming P f g*)›
apply (*simp add*: ∗(*3*) ∗∗(*2*, *3*) *map-event*$_{ptick}$*-tickFree*, *intro conjI*)
apply (*metis* ∗(*1*, *2*) ∗∗(*1*) ∗∗(*3*) *front-tickFree-append-iff*
map-event$_{ptick}$*-front-tickFree map-event*$_{ptick}$*-tickFree tick-Free-append-iff*)
apply (*rule trace-hide-map-map-event*$_{ptick}$[*symmetric*])
apply (*meson inj-def inj-onCI inj-on-map-event*$_{ptick}$)
apply (*simp add*: *D-Renaming*)
using ∗∗(*2*) *front-tickFree-Nil* **by** *blast*
thus ‹*s* ∈ $\mathcal{D}$ *?rhs*› **by** (*simp add*: *D-Hiding*) *blast*
next
assume ‹∃ *h*. *isInfHiddenRun h P S* ∧ *t* ∈ *range h*›
then obtain *h* **where** ‹*isInfHiddenRun h P S*› ‹*t* ∈ *range h*› **by** *blast*
hence ‹*ftF* (*map* (*map-event*$_{ptick}$ *f g*) *u* @ *s2*) ∧
tF (*map* (*map-event*$_{ptick}$ *f g*) *t*) ∧
s = *trace-hide* (*map* (*map-event*$_{ptick}$ *f g*) *t*) (*ev* ‘ *f* ‘ *S*) @ *map* (*map-event*$_{ptick}$ *f g*) *u* @ *s2* ∧
isInfHiddenRun (λ*i*. *map* (*map-event*$_{ptick}$ *f g*) (*h i*)) (*Renaming P f g*) (*f* ‘ *S*) ∧
map (*map-event*$_{ptick}$ *f g*) *t* ∈ *range* (λ*i*. *map* (*map-event*$_{ptick}$ *f g*) (*h i*))›
apply (*simp add*: ∗(*3*) ∗∗(*2*, *3*) *map-event*$_{ptick}$*-tickFree*, *intro conjI*)

apply (*metis* ∗(*1*, *2*) ∗∗(*3*) *front-tickFree-append map-event*$_{ptick}$*-tickFree tickFree-append-iff*)
apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$, *symmetric*])
apply (*solves* ‹*rule strict-mono-map*, *simp*›)
apply (*solves* ‹*auto simp add*: *T-Renaming*›)
apply (*subst* (*1 2*) *trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
by *metis blast*
thus ‹*s* ∈ 𝒟 *?rhs*› **by** (*simp add*: *D-Hiding*) *blast*
qed
next
fix *s*
assume ‹*s* ∈ 𝒟 *?rhs*›
then obtain *t u*
where ∗ : ‹*ftF u*› ‹*tF t*› ‹*s* = *trace-hide t* (*ev* ‘ *f* ‘ *S*) @ *u*›
‹*t* ∈ 𝒟 (*Renaming P f g*) ∨
(∃ *h*. *isInfHiddenRun h* (*Renaming P f g*) (*f* ‘ *S*) ∧ *t* ∈ *range h*)›
by (*simp add*: *D-Hiding*) *blast*
from ∗(*4*) **show** ‹*s* ∈ 𝒟 *?lhs*›
proof (*elim disjE*)
assume ‹*t* ∈ 𝒟 (*Renaming P f g*)›
then obtain *t1 t2* **where** ∗∗ : ‹*tF t1*› ‹*ftF t2*›
‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *t1* @ *t2*› ‹*t1* ∈ 𝒟 *P*›
by (*simp add*: *D-Renaming*) *blast*
have ‹*tF* (*trace-hide t1* (*ev* ‘ *S*)) ∧
ftF (*trace-hide t2* (*ev* ‘ *f* ‘ *S*) @ *u*) ∧
trace-hide (*map* (*map-event*$_{ptick}$ *f g*) *t1*) (*ev* ‘ *f* ‘ *S*) @ *trace-hide t2* (*ev* ‘ *f* ‘ *S*) @ *u* =
map (*map-event*$_{ptick}$ *f g*) (*trace-hide t1* (*ev* ‘ *S*)) @ *trace-hide t2* (*ev* ‘ *f* ‘ *S*) @ *u* ∧
trace-hide t1 (*ev* ‘ *S*) ∈ 𝒟 (*P* \ *S*)›
apply (*simp*, *intro conjI*)
using ∗∗(*1*) *Hiding-tickFree* **apply** *blast*
using ∗(*1*, *2*) ∗∗(*3*) *Hiding-tickFree front-tickFree-append tickFree-append-iff* **apply** *blast*
apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
using ∗∗(*4*) *mem-D-imp-mem-D-Hiding* **by** *blast*
thus ‹*s* ∈ 𝒟 *?lhs*› **by** (*simp add*: *D-Renaming* ∗(*3*) ∗∗(*3*)) *blast*
next
have *inv-S*: ‹*S* = *inv f* ‘ *f* ‘ *S*› **by** (*simp add*: *bij-is-inj bij-f*)
have *inj-inv-f*: ‹*inj* (*inv f*)›
by (*simp add*: *bij-imp-bij-inv bij-is-inj bij-f*)
have ∗∗ : ‹*map* (*map-event*$_{ptick}$ (*inv f*) (*inv g*) ∘ *map-event*$_{ptick}$ *f g*) *v* = *v*› **for** *v*
by (*induct v*, *simp-all*)
(*metis UNIV-I bij-betw-inv-into-left bij-f bij-g event*$_{ptick}$*.exhaust event*$_{ptick}$*.map*(*2*) *event*$_{ptick}$*.simps*(*9*))
assume ‹∃ *h*. *isInfHiddenRun h* (*Renaming P f g*) (*f* ‘ *S*) ∧ *t* ∈ *range h*›
then obtain *h*

where $***$: ‹*isInfHiddenRun h* (*Renaming P f g*) (*f ‘ S*)› ‹*t* ∈ *range h*› **by** *blast*

then consider *t1* **where** ‹*t1* ∈ $\mathcal{T}$ *P*› ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *t1*›
| *t1 t2* **where** ‹*tF t1*› ‹*ftF t2*›
‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *t1* @ *t2*› ‹*t1* ∈ $\mathcal{D}$ *P*›
by (*simp add*: *T-Renaming*) *blast*

thus ‹*s* ∈ $\mathcal{D}$ *?lhs*›

proof *cases*

fix *t1* **assume** $****$: ‹*t1* ∈ $\mathcal{T}$ *P*› ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *t1*›

have $*****$: ‹*t1* = *map* (*map-event*$_{ptick}$ (*inv f*) (*inv g*)) *t*› **by** (*simp add*: $****(2)$ $**$)

have $******$: ‹*trace-hide t1* (*ev ‘ S*) = *trace-hide t1* (*ev ‘ S*) ∧
isInfHiddenRun (λ*i*. *map* (*map-event*$_{ptick}$ (*inv f*) (*inv g*)) (*h i*)) *P S* ∧
t1 ∈ *range* (λ*i*. *map* (*map-event*$_{ptick}$ (*inv f*) (*inv g*)) (*h i*))›

apply (*simp add*: $***(1)$ *strict-mono-map*, *intro conjI*)

apply (*subst Renaming-inv*[**where** *f* = *f* **and** *g* = *g*, *symmetric*])

apply (*solves* ‹*simp add*: *bij-is-inj bij-f*›)

apply (*solves* ‹*simp add*: *bij-is-inj bij-g*›)

using $***(1)$ **apply** (*subst T-Renaming*, *blast*)

apply (*subst* (*1 2*) *inv-S*, *subst* (*1 2*) *trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$*-inv*])

apply (*metis* $***(1)$)

using $***(2)$ $*****$ **by** *blast*

have ‹*tF* (*trace-hide t1* (*ev ‘ S*)) ∧ *ftF t1* ∧
trace-hide (*map* (*map-event*$_{ptick}$ *f g*) *t1*) (*ev ‘ f ‘ S*) @ *u* =
map (*map-event*$_{ptick}$ *f g*) (*trace-hide t1* (*ev ‘ S*)) @ *u* ∧
trace-hide t1 (*ev ‘ S*) ∈ $\mathcal{D}$ (*P* \ *S*)›

apply (*simp*, *intro conjI*)

using $*(2)$ $****(2)$ *map-event*$_{ptick}$*-tickFree Hiding-tickFree* **apply** *blast*

using $****(1)$ *is-processT2-TR* **apply** *blast*

apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])

apply (*simp add*: *D-Renaming D-Hiding*)

using $*(2)$ $*****$ $******$ *map-event*$_{ptick}$*-tickFree front-tickFree-Nil* **by** *blast*

with $*(1)$ **show** ‹*s* ∈ $\mathcal{D}$ *?lhs*› **by** (*simp add*: *D-Renaming* $*(3)$ $****(2)$) *blast*

next

fix *t1 t2* **assume** $****$: ‹*tF t1*› ‹*ftF t2*›
‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *t1* @ *t2*› ‹*t1* ∈ $\mathcal{D}$ *P*›

have ‹*tF* (*trace-hide t1* (*ev ‘ S*)) ∧
ftF (*trace-hide t2* (*ev ‘ f ‘ S*) @ *u*) ∧
trace-hide (*map* (*map-event*$_{ptick}$ *f g*) *t1*) (*ev ‘ f ‘ S*) @ *trace-hide t2* (*ev ‘ f ‘ S*) @ *u* =
map (*map-event*$_{ptick}$ *f g*) (*trace-hide t1* (*ev ‘ S*)) @ *trace-hide t2* (*ev ‘ f ‘ S*) @ *u* ∧
trace-hide t1 (*ev ‘ S*) ∈ $\mathcal{D}$ (*P* \ *S*)›

apply (*simp*, *intro conjI*)

using $****(1)$ *Hiding-tickFree* **apply** *blast*
using $*(1, 2)$ $****(3)$ *Hiding-tickFree front-tickFree-append tickFree-append-iff* **apply** *blast*
apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
using $****(4)$ *mem-D-imp-mem-D-Hiding* **by** *blast*
thus ‹$s \in \mathcal{D}$ *?lhs*› **by** (*simp add*: *D-Renaming* $*(3)$ $****(3)$) *blast*
qed
qed
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹$(s, X) \in \mathcal{F}$ *?lhs*›
then consider ‹$s \in \mathcal{D}$ *?lhs*›
| *s1* **where** ‹(*s1*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ (*P* \ *S*)› ‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1*›
by (*simp add*: *F-Renaming D-Renaming*) *blast*
thus ‹$(s, X) \in \mathcal{F}$ *?rhs*›
proof *cases*
from *D-F same-div* **show** ‹$s \in \mathcal{D}$ *?lhs* ⟹ $(s, X) \in \mathcal{F}$ *?rhs*› **by** *blast*
next
fix *s1* **assume** $*$: ‹(*s1*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ (*P* \ *S*)›
‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1*›
from *this*(*1*) **consider** ‹*s1* ∈ $\mathcal{D}$ (*P* \ *S*)›
| *t* **where** ‹*s1* = *trace-hide t* (*ev* ‘ *S*)› ‹(*t*, *map-event*$_{ptick}$ *f g* −‘ *X* ∪ *ev* ‘ *S*) ∈ $\mathcal{F}$ *P*›
by (*simp add*: *F-Hiding D-Hiding*) *blast*
thus ‹$(s, X) \in \mathcal{F}$ *?rhs*›
proof *cases*
assume ‹*s1* ∈ $\mathcal{D}$ (*P* \ *S*)›
then obtain *t u*
where $**$: ‹*ftF u*› ‹*tF t*› ‹*s1* = *trace-hide t* (*ev* ‘ *S*) @ *u*›
‹*t* ∈ $\mathcal{D}$ *P* ∨ (∃ *g*. *isInfHiddenRun g P S* ∧ *t* ∈ *range g*)›
by (*simp add*: *D-Hiding*) *blast*
have $***$: ‹*ftF* (*map* (*map-event*$_{ptick}$ *f g*) *u*) ∧ *tF* (*map* (*map-event*$_{ptick}$ *f g*) *t*) ∧
map (*map-event*$_{ptick}$ *f g*) (*trace-hide t* (*ev* ‘ *S*)) @ *map* (*map-event*$_{ptick}$ *f g*) *u* =
trace-hide (*map* (*map-event*$_{ptick}$ *f g*) *t*) (*ev* ‘ *f* ‘ *S*) @ (*map* (*map-event*$_{ptick}$ *f g*) *u*)›
by (*simp add*: *map-event*$_{ptick}$*-front-tickFree map-event*$_{ptick}$*-tickFree* $**$(*1*, *2*))
(*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$, *symmetric*])
from $**(4)$ **show** ‹$(s, X) \in \mathcal{F}$ *?rhs*›
proof (*elim disjE exE*)
assume ‹$t \in \mathcal{D}$ *P*›
hence $: ‹*map* (*map-event*$_{ptick}$ *f g*) *t* ∈ $\mathcal{D}$ (*Renaming P f g*)›
apply (*simp add*: *D-Renaming*)
using $**(2)$ *front-tickFree-Nil* **by** *blast*

show ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
by (*simp add*: *F-Hiding*) (*metis* $ ∗(*2*) ∗∗(*3*) ∗∗∗ *map-append*)
next
fix *h* **assume** ‹*isInfHiddenRun h P S* ∧ *t* ∈ *range h*›
hence $: ‹*isInfHiddenRun* (λ*i*. *map* (*map-event*$_{ptick}$ *f g*) (*h i*)) (*Renaming P f g*) (*f* ‘ *S*) ∧
map (*map-event*$_{ptick}$ *f g*) *t* ∈ *range* (λ*i*. *map* (*map-event*$_{ptick}$ *f g*) (*h i*))›
apply (*subst* (*1 2*) *trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
by (*simp add*: *strict-mono-map T-Renaming image-iff*) (*metis* (*mono-tags*, *lifting*))
show ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
apply (*simp add*: *F-Hiding*)

by (*smt* (*verit*, *del-insts*) $ ∗(*2*) ∗∗(*3*) ∗∗∗ *map-append*)
qed
next
fix *t* **assume** ∗∗ : ‹*s1* = *trace-hide t* (*ev* ‘ *S*)›
‹(*t*, *map-event*$_{ptick}$ *f g* −‘ *X* ∪ *ev* ‘ *S*) ∈ $\mathcal{F}$ *P*›
have ∗∗∗ : ‹*map-event*$_{ptick}$ *f g* −‘ *X* ∪ *map-event*$_{ptick}$ *f g* −‘ *ev* ‘ *f* ‘ *S* = *map-event*$_{ptick}$ *f g* −‘ *X* ∪ *ev* ‘ *S*›
by (*simp add*: *set-eq-iff map-event*$_{ptick}$*-eq-ev-iff image-iff*) (*metis bij-f bij-pointE*)
have ‹*map* (*map-event*$_{ptick}$ *f g*) (*trace-hide t* (*ev* ‘ *S*)) =
trace-hide (*map* (*map-event*$_{ptick}$ *f g*) *t*) (*ev* ‘ *f* ‘ *S*) ∧
(*map* (*map-event*$_{ptick}$ *f g*) *t*, *X* ∪ *ev* ‘ *f* ‘ *S*) ∈ $\mathcal{F}$ (*Renaming P f g*)›
apply (*intro conjI*)
apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$, *symmetric*])
apply (*simp add*: *F-Renaming*)
using ∗∗(*2*) ∗∗∗ **by** *auto*
show ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
apply (*simp add*: *F-Hiding* ∗(*2*) ∗∗(*1*))
using ‹*?this*› **by** *blast*
qed
qed
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
then consider ‹$s \in \mathcal{D}$ *?rhs*›
| *t* **where** ‹*s* = *trace-hide t* (*ev* ‘ *f* ‘ *S*)› ‹(*t*, *X* ∪ *ev* ‘ *f* ‘ *S*) ∈ $\mathcal{F}$ (*Renaming P f g*)›
by (*simp add*: *F-Hiding D-Hiding*) *blast*
thus ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
proof *cases*
from *D-F same-div* **show** ‹$s \in \mathcal{D}$ *?rhs* ⟹ $(s,\ X) \in \mathcal{F}$ *?lhs*› **by** *blast*
next
fix *t* **assume** ‹*s* = *trace-hide t* (*ev* ‘ *f* ‘ *S*)› ‹(*t*, *X* ∪ *ev* ‘ *f* ‘ *S*) ∈ $\mathcal{F}$ (*Renaming*

$P\ f\ g)$›
then obtain t
where $*$: ‹s = *trace-hide* t (*ev* ‘ f ‘ S)›
‹(t, $X \cup$ *ev* ‘ f ‘ S) $\in \mathcal{F}$ (*Renaming* $P\ f\ g$)› **by** *blast*
have $**$: ‹*map-event*$_{ptick}$ $f\ g$ −‘ $X \cup$ *map-event*$_{ptick}$ $f\ g$ −‘ *ev* ‘ f ‘ S = *map-event*$_{ptick}$ $f\ g$ −‘ $X \cup$ *ev* ‘ S›
by (*simp add*: *set-eq-iff map-event*$_{ptick}$*-eq-ev-iff image-iff*) (*metis bij-f bij-pointE*)
have ‹($\exists$ *s1*. (*s1*, *map-event*$_{ptick}$ $f\ g$ −‘ $X \cup$ *map-event*$_{ptick}$ $f\ g$ −‘ *ev* ‘ f ‘ S) $\in \mathcal{F}$ $P \wedge$ t = *map* (*map-event*$_{ptick}$ $f\ g$) *s1*) $\vee$
($\exists$ *s1 s2*. *tF s1* $\wedge$ *ftF s2* $\wedge$ t = *map* (*map-event*$_{ptick}$ $f\ g$) *s1* @ *s2* $\wedge$ *s1* $\in \mathcal{D}$ P)›
using $*$(*2*) **by** (*auto simp add*: *F-Renaming*)
thus ‹(s, X) $\in \mathcal{F}$ *?lhs*›
proof (*elim disjE exE conjE*)
fix *s1*
assume ‹(*s1*, *map-event*$_{ptick}$ $f\ g$ −‘ $X \cup$ *map-event*$_{ptick}$ $f\ g$ −‘ *ev* ‘ f ‘ S) $\in \mathcal{F}$ P› ‹t = *map* (*map-event*$_{ptick}$ $f\ g$) *s1*›
hence ‹(*trace-hide s1* (*ev* ‘ S), *map-event*$_{ptick}$ $f\ g$ −‘ X) $\in \mathcal{F}$ ($P \setminus S$) $\wedge$
s = *map* (*map-event*$_{ptick}$ $f\ g$) (*trace-hide s1* (*ev* ‘ S))›
apply (*simp add*: $*$(*1*) *F-Hiding* $**$, *intro conjI*)
by *blast* (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
show ‹(s, X) $\in \mathcal{F}$ *?lhs*›
apply (*simp add*: *F-Renaming*)
using ‹*?this*› **by** *blast*
next
fix *s1 s2*
assume ‹*tF s1*› ‹*ftF s2*› ‹t = *map* (*map-event*$_{ptick}$ $f\ g$) *s1* @ *s2*› ‹*s1* $\in \mathcal{D}$ P›
hence ‹*tF* (*trace-hide s1* (*ev* ‘ S)) $\wedge$
ftF (*trace-hide s2* (*ev* ‘ f ‘ S)) $\wedge$
s = *map* (*map-event*$_{ptick}$ $f\ g$) (*trace-hide s1* (*ev* ‘ S)) @ *trace-hide s2* (*ev* ‘ f ‘ S) $\wedge$
trace-hide s1 (*ev* ‘ S) $\in \mathcal{D}$ ($P \setminus S$)›
apply (*simp add*: *F-Renaming* $*$(*1*), *intro conjI*)
using *Hiding-tickFree* **apply** *blast*
using *Hiding-front-tickFree* **apply** *blast*
apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
using *mem-D-imp-mem-D-Hiding* **by** *blast*
show ‹(s, X) $\in \mathcal{F}$ *?lhs*›
apply (*simp add*: *F-Renaming*)
using ‹*?this*› **by** *blast*
qed
qed
qed
qed

9.2.3 *Renaming* and *Sync*

Idem for the synchronization: when f is one to one, *Renaming* $(P \ [\![S]\!] \ Q)$ will behave as expected.

lemma *map-antecedent-if-subset-rangeE* :
assumes ‹*set u* ⊆ *range f*›
obtains *t* **where** ‹*u* = *map f t*›
— In particular, when f is surjective or bijective.
proof −
from ‹*set u* ⊆ *range f*› **have** ‹∃ *t*. *u* = *map f t*›
proof (*induct u*)
show ‹∃ *t*. [] = *map f t*› **by** *simp*
next
fix *a u*
assume *prem* : ‹*set* (*a* # *u*) ⊆ *range f*›
and *hyp* : ‹*set u* ⊆ *range f* ⟹ ∃ *t*. *u* = *map f t*›
then obtain *t* **where** ∗ : ‹*u* = *map f t*›
by (*meson set-subset-Cons subset-trans*)
from *prem* **obtain** *x* **where** ∗∗ : ‹*f x* = *a*› **by** *auto*
show ‹∃ *t*. *a* # *u* = *map f t*›
proof (*intro exI*)
show ‹*a* # *u* = *map f* (*x* # *t*)› **by** (*simp add*: ∗ ∗∗)
qed
qed
with *that* **show** *thesis* **by** *blast*
qed

lemma *bij-map-setinterleaving-iff-setinterleaving* :
‹*map f r setinterleaves* ((*map f t*, *map f u*), *f* ‘ *S*) ⟷
r setinterleaves ((*t*, *u*), *S*)› **if** *bij-f* : ‹*bij f*›
proof (*induct* ‹(*t*, *S*, *u*)› *arbitrary*: *t u r rule*: *setinterleaving.induct*)
case *1*
thus *?case* **by** *simp*
next
case (*2 y u*)
show *?case*
proof (*cases* ‹*y* ∈ *S*›)
show ‹*y* ∈ *S* ⟹ *?case*› **by** *simp*
next
assume ‹*y* ∉ *S*›
hence ‹*f y* ∉ *f* ‘ *S*› **by** (*metis bij-betw-imp-inj-on inj-image-mem-iff bij-f*)
with *2.hyps*[*OF* ‹*y* ∉ *S*›, *of* ‹*tl r*›] **show** *?case*
by (*cases r*; *simp add*: ‹*y* ∉ *S*›) (*metis bij-pointE bij-f*)
qed
next
case (*3 x t*)
show *?case*
proof (*cases* ‹*x* ∈ *S*›)

```
    show ‹x ∈ S ⟹ ?case› by simp
  next
    assume ‹x ∉ S›
    hence ‹f x ∉ f ‘ S› by (metis bij-betw-imp-inj-on inj-image-mem-iff bij-f)
    with 3.hyps[OF ‹x ∉ S›, of ‹tl r›] show ?case
      by (cases r; simp add: ‹x ∉ S›) (metis bij-pointE bij-f)
  qed
next
  case (4 x t y u)
  have ∗ : ‹x ≠ y ⟹ f x ≠ f y› by (metis bij-pointE bij-f)
  have ∗∗ : ‹f z ∈ f ‘ S ⟷ z ∈ S› for z
    by (meson bij-betw-def inj-image-mem-iff bij-f)
  show ?case
  proof (cases ‹x ∈ S›; cases ‹y ∈ S›)
    from 4.hyps(1)[of ‹tl r›] show ‹x ∈ S ⟹ y ∈ S ⟹ ?case›
      by (cases r; simp add: ∗) (metis bij-pointE bij-f)
  next
    from 4.hyps(2)[of ‹tl r›] show ‹x ∈ S ⟹ y ∉ S ⟹ ?case›
      by (cases r; simp add: ∗∗) (metis bij-pointE bij-f)
  next
    from 4.hyps(5)[of ‹tl r›] show ‹x ∉ S ⟹ y ∈ S ⟹ ?case›
      by (cases r; simp add: ∗∗) (metis bij-pointE bij-f)
  next
    from 4.hyps(3, 4)[of ‹tl r›] show ‹x ∉ S ⟹ y ∉ S ⟹ ?case›
      by (cases r; simp add: ∗∗) (metis bij-pointE bij-f)
  qed
qed


theorem bij-Renaming-Sync:
  ‹Renaming (P ⟦S⟧ Q) f g = Renaming P f g ⟦f ‘ S⟧ Renaming Q f g›
  (is ‹?lhs P Q = ?rhs P Q›) if bij-f: ‹bij f› and bij-g : ‹bij g›
proof −
  — Some intermediate results.
  have map-event_ptick-inv-comp-map-event_ptick : ‹map-event_ptick (inv f) (inv g)
∘ map-event_ptick f g = id›
  proof (rule ext)
    show map-event_ptick-inv-comp-map-event_ptick :
      ‹(map-event_ptick (inv f) (inv g) ∘ map-event_ptick f g) t = id t› for t
      by (induct t, simp-all) (metis bij-f bij-inv-eq-iff, metis bij-g bij-inv-eq-iff)
  qed
  have map-event_ptick-comp-map-event_ptick-inv : ‹map-event_ptick f g ∘ map-event_ptick
(inv f) (inv g) = id›
    proof (rule ext)
    show map-event_ptick-inv-comp-map-event_ptick :
      ‹(map-event_ptick f g ∘ map-event_ptick (inv f) (inv g)) t = id t› for t
      by (induct t, simp-all) (metis bij-f bij-inv-eq-iff, metis bij-g bij-inv-eq-iff)
  qed
  from map-event_ptick-inv-comp-map-event_ptick map-event_ptick-comp-map-event_ptick-inv
```

o-bij
have *bij-map-event*$_{ptick}$: ‹*bij* (*map-event*$_{ptick}$ *f g*)› **by** *blast*
have *inv-map-event*$_{ptick}$*-is-map-event*$_{ptick}$*-inv* : ‹*inv* (*map-event*$_{ptick}$ *f g*) = *map-event*$_{ptick}$ (*inv f*) (*inv g*)›
by (*simp add*: *inv-equality map-event*$_{ptick}$*-comp-map-event*$_{ptick}$*-inv map-event*$_{ptick}$*-inv-comp-map-eve pointfree-idE*)
have *sets-S-eq* : ‹*map-event*$_{ptick}$ *f g* ‘ (*range tick* ∪ *ev* ‘ *S*) = *range tick* ∪ *ev* ‘ *f* ‘ *S*›
by (*auto simp add*: *set-eq-iff image-iff tick-eq-map-event*$_{ptick}$*-iff ev-eq-map-event*$_{ptick}$*-iff split*: *event*$_{ptick}$*.split*)
(*metis UNIV-I Un-iff bij-betw-def bij-g image-iff*, *blast*)
have *inj-map-event*$_{ptick}$: ‹*inj* (*map-event*$_{ptick}$ *f g*)›
and *inj-inv-map-event*$_{ptick}$: ‹*inj* (*inv* (*map-event*$_{ptick}$ *f g*))›
by (*use bij-betw-imp-inj-on bij-map-event*$_{ptick}$ **in** *blast*)
(*meson bij-betw-imp-inj-on bij-betw-inv-into bij-map-event*$_{ptick}$)
show ‹*?lhs P Q* = *?rhs P Q*›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)›
then obtain *s1 s2*
where ∗ : ‹*tF s1*› ‹*ftF s2*› ‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *s2*› ‹*s1* ∈ $\mathcal{D}$ (*P* ⟦*S*⟧ *Q*)›
by (*simp add*: *D-Renaming*) *blast*
from ∗(*4*) **obtain** *t u r v*
where ∗∗ : ‹*ftF v*› ‹*tF r* ∨ *v* = []›
‹*s1* = *r* @ *v*› ‹*r setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
‹*t* ∈ $\mathcal{D}$ *P* ∧ *u* ∈ $\mathcal{T}$ *Q* ∨ *t* ∈ $\mathcal{D}$ *Q* ∧ *u* ∈ $\mathcal{T}$ *P*›
by (*simp add*: *D-Sync*) *blast*
{ fix *t u P Q*
assume *assms* : ‹*r setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
‹*t* ∈ $\mathcal{D}$ *P*› ‹*u* ∈ $\mathcal{T}$ *Q*›
have ‹*map* (*map-event*$_{ptick}$ *f g*) *r setinterleaves*
((*map* (*map-event*$_{ptick}$ *f g*) *t*, *map* (*map-event*$_{ptick}$ *f g*) *u*), *range tick* ∪ *ev* ‘ *f* ‘ *S*)›
by (*metis assms*(*1*) *bij-map-setinterleaving-iff-setinterleaving bij-map-event*$_{ptick}$ *sets-S-eq*)
moreover have ‹*map* (*map-event*$_{ptick}$ *f g*) *t* ∈ $\mathcal{D}$ (*Renaming P f g*)›
apply (*cases* ‹*tF t*›; *simp add*: *D-Renaming*)
using *assms*(*2*) *front-tickFree-Nil* **apply** *blast*
by (*metis D-T D-imp-front-tickFree append-T-imp-tickFree assms*(*2*) *front-tickFree-Cons-iff is-processT9 list.simps*(*3*) *map-append nonTickFree-n-frontTickFree map-event*$_{ptick}$*-front-tickFree*)
moreover have ‹*map* (*map-event*$_{ptick}$ *f g*) *u* ∈ $\mathcal{T}$ (*Renaming Q f g*)›
using *assms*(*3*) **by** (*simp add*: *T-Renaming*) *blast*
ultimately have ‹*s* ∈ $\mathcal{D}$ (*?rhs P Q*)›
by (*simp add*: *D-Sync* ∗(*3*) ∗∗(*3*))
(*metis* ∗(*1*, *2*) ∗∗(*3*) *map-event*$_{ptick}$*-tickFree front-tickFree-append tick-Free-append-iff*)
} note ∗∗∗ = *this*

from $**(4,\ 5)$ $***$ **show** ‹$s \in \mathcal{D}\ (?rhs\ P\ Q)$›
apply (*elim disjE*)
using $**(4)$ $***$ **apply** *blast*
using $**(4)$ $***$ **by** (*subst Sync-commute*) *blast*
next
fix *s*
assume ‹$s \in \mathcal{D}\ (?rhs\ P\ Q)$›
then obtain *t u r v*
where $*$: ‹*ftF v*› ‹$tF\ r \vee v = []$› ‹$s = r\ @\ v$›
‹*r setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *f* ‘ *S*)›
‹$t \in \mathcal{D}\ (Renaming\ P\ f\ g) \wedge u \in \mathcal{T}\ (Renaming\ Q\ f\ g) \vee$
$t \in \mathcal{D}\ (Renaming\ Q\ f\ g) \wedge u \in \mathcal{T}\ (Renaming\ P\ f\ g)$›
by (*simp add*: *D-Sync*) *blast*

{ fix *t u P Q*
assume *assms* : ‹*r setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *f* ‘ *S*)›
‹$t \in \mathcal{D}\ (Renaming\ P\ f\ g)$› ‹$u \in \mathcal{T}\ (Renaming\ Q\ f\ g)$›
have ‹*inv* ($map\text{-}event_{ptick}\ f\ g$) ‘ (*range tick* ∪ *ev* ‘ *f* ‘ *S*) =
inv ($map\text{-}event_{ptick}\ f\ g$) ‘ $map\text{-}event_{ptick}\ f\ g$ ‘ (*range tick* ∪ *ev* ‘ *S*)›
using *sets-S-eq* **by** *presburger*
from *bij-map-setinterleaving-iff-setinterleaving*
[*THEN iffD2*, *OF - assms*(*1*), *of* ‹*inv* ($map\text{-}event_{ptick}\ f\ g$)›,
simplified this image-inv-f-f[*OF inj-map-event*$_{ptick}$]]
have $**$: ‹(*map* (*inv* ($map\text{-}event_{ptick}\ f\ g$)) *r*) *setinterleaves*
((*map* (*inv* ($map\text{-}event_{ptick}\ f\ g$)) *t*, *map* (*inv* ($map\text{-}event_{ptick}\ f\ g$))
u), *range tick* ∪ *ev* ‘ *S*)›
using *bij-betw-inv-into bij-map-event*$_{ptick}$ **by** *blast*
from *assms*(*2*) **obtain** *s1 s2*
where ‹$t = map\ (map\text{-}event_{ptick}\ f\ g)\ s1\ @\ s2$› ‹*tF s1*› ‹*ftF s2*› ‹$s1 \in \mathcal{D}\ P$›
by (*auto simp add*: *D-Renaming*)
hence ‹*map* ($map\text{-}event_{ptick}$ (*inv f*) (*inv g*)) $t \in \mathcal{D}$ (*Renaming* (*Renaming P*
f g) (*inv f*) (*inv g*))›
apply (*simp add*: *D-Renaming*)
apply (*rule-tac x* = ‹*map* ($map\text{-}event_{ptick}\ f\ g$) *s1*› **in** *exI*)
apply (*rule-tac x* = ‹*map* ($map\text{-}event_{ptick}$ (*inv f*) (*inv g*)) *s2*› **in** *exI*)
by *simp* (*metis append-Nil2 front-tickFree-Nil map-event*$_{ptick}$*-front-tickFree*
map-event$_{ptick}$*-tickFree*)
hence $***$: ‹*map* (*inv* ($map\text{-}event_{ptick}\ f\ g$)) $t \in \mathcal{D}\ P$›
by (*metis Renaming-inv bij-def bij-f bij-g inv-map-event*$_{ptick}$*-is-map-event*$_{ptick}$*-inv*)
have ‹*map* ($map\text{-}event_{ptick}$ (*inv f*) (*inv g*)) $u \in \mathcal{T}$ (*Renaming* (*Renaming Q*
f g) (*inv f*) (*inv g*))›
using *assms*(*3*) **by** (*subst T-Renaming*, *simp*) *blast*
hence $****$: ‹*map* (*inv* ($map\text{-}event_{ptick}\ f\ g$)) $u \in \mathcal{T}\ Q$›
by (*simp add*: *Renaming-inv bij-f bij-g bij-is-inj inv-map-event*$_{ptick}$*-is-map-event*$_{ptick}$*-inv*)
have $*****$: ‹*map* ($map\text{-}event_{ptick}\ f\ g \circ inv\ (map\text{-}event_{ptick}\ f\ g)$) $r = r$›
by (*metis* (*no-types*, *lifting*) *bij-betw-imp-inj-on bij-betw-inv-into bij-map-event*$_{ptick}$
inj-iff list.map-comp list.map-id)
have ‹$s \in \mathcal{D}\ (?lhs\ P\ Q)$›

proof (*cases* ‹*tF r*›)
assume ‹*tF r*›
have $: ‹*r* @ *v* = *map* (*map-event*$_{ptick}$ *f g*) (*map* (*inv* (*map-event*$_{ptick}$ *f g*)) *r*) @ *v*›
by (*simp add*: *****)
show ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)›
apply (*simp add*: *D-Renaming D-Sync* ∗(*3*))
by (*metis* $ ∗(*1*) ** *** **** *map-event*$_{ptick}$*-tickFree* ‹*tF r*›
append.right-neutral append-same-eq front-tickFree-Nil)
next
assume ‹¬ *tF r*›
then obtain *r′ res* **where** $: ‹*r* = *r′* @ [✓(*res*)]› ‹*tF r′*›
by (*metis D-imp-front-tickFree assms butlast-snoc front-tickFree-charn*
front-tickFree-single ftf-Sync is-processT2-TR list.distinct(*1*)
nonTickFree-n-frontTickFree self-append-conv2)
then obtain *t′ u′*
where $$: ‹*t* = *t′* @ [✓(*res*)]› ‹*u* = *u′* @ [✓(*res*)]›
by (*metis D-imp-front-tickFree SyncWithTick-imp-NTF T-imp-front-tickFree assms*)
hence $$$: ‹(*map* (*inv* (*map-event*$_{ptick}$ *f g*)) *r′*) *setinterleaves*
((*map* (*inv* (*map-event*$_{ptick}$ *f g*)) *t′*, *map* (*inv* (*map-event*$_{ptick}$ *f g*)) *u′*),
range tick ∪ *ev* ‘ *S*)›
by (*metis* (*no-types*) $(*1*) *append-same-eq assms*(*1*) *bij-betw-imp-surj-on*
bij-map-setinterleaving-iff-setinterleaving bij-map-event$_{ptick}$
ftf-Sync32 inj-imp-bij-betw-inv inj-map-event$_{ptick}$ *sets-S-eq*)
from *** $$(*1*) **have** *** : ‹*map* (*inv* (*map-event*$_{ptick}$ *f g*)) *t′* ∈ $\mathcal{D}$ *P*›
by *simp* (*use inv-map-event*$_{ptick}$*-is-map-event*$_{ptick}$*-inv is-processT9* **in** *force*)
from **** $$(*2*) **have** **** : ‹*map* (*inv* (*map-event*$_{ptick}$ *f g*)) *u′* ∈ $\mathcal{T}$ *Q*›
using *is-processT3-TR prefixI* **by** *simp blast*
have $$$$: ‹*r* = *map* (*map-event*$_{ptick}$ *f g*) (*map* (*inv* (*map-event*$_{ptick}$ *f g*)) *r′*) @ [✓(*res*)]›
using $ ***** **by** *auto*
show ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)›
by (*simp add*: *D-Renaming D-Sync* ∗(*3*) $$$)
(*metis* $(*1*) $(*2*) $$$ $$$$ ∗(*2*) *** **** *map-event*$_{ptick}$*-tickFree* ‹¬ *tF r*›
append.right-neutral append-same-eq front-tickFree-Nil front-tickFree-single)
qed
} **note** ** = *this*
show ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)› **by** (*metis* ∗(*4*, *5*) ** *Sync-commute*)
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ (*?lhs P Q*) = $\mathcal{D}$ (*?rhs P Q*)›
assume ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?lhs P Q*)›
then consider ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)›
| *s1* **where** ‹(*s1*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ (*P* ⟦*S*⟧ *Q*)› ‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1*›

by (*simp add*: *F-Renaming D-Renaming*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?rhs P Q*)›
proof *cases*
from *same-div D-F* **show** ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ (*?rhs P Q*)› **by** *blast*
next
fix *s1* **assume** ∗ : ‹(*s1*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ (*P* ⟦*S*⟧ *Q*)›
‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1*›
from ∗(*1*) **consider** ‹*s1* ∈ $\mathcal{D}$ (*P* ⟦*S*⟧ *Q*)›
| *t-P t-Q X-P X-Q*
where ‹(*t-P*, *X-P*) ∈ $\mathcal{F}$ *P*› ‹(*t-Q*, *X-Q*) ∈ $\mathcal{F}$ *Q*›
‹*s1 setinterleaves* ((*t-P*, *t-Q*), *range tick* ∪ *ev* ‘ *S*)›
‹*map-event*$_{ptick}$ *f g* −‘ *X* = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
by (*auto simp add*: *F-Sync D-Sync*)
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?rhs P Q*)›
proof *cases*
assume ‹*s1* ∈ $\mathcal{D}$ (*P* ⟦*S*⟧ *Q*)›
hence ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)›
apply (*cases* ‹*tF s1*›; *simp add*: *D-Renaming* ∗(*2*))
using *front-tickFree-Nil* **apply** *blast*
by (*metis* (*no-types*, *lifting*) *map-event*$_{ptick}$*-front-tickFree butlast-snoc front-tickFree-iff-tickFree-butlast*
front-tickFree-single map-butlast nonTickFree-n-frontTickFree process-charn)
with *same-div D-F* **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?rhs P Q*)› **by** *blast*
next
fix *t-P t-Q X-P X-Q*
assume ∗∗ : ‹(*t-P*, *X-P*) ∈ $\mathcal{F}$ *P*› ‹(*t-Q*, *X-Q*) ∈ $\mathcal{F}$ *Q*›
‹*s1 setinterleaves* ((*t-P*, *t-Q*), *range tick* ∪ *ev* ‘ *S*)›
‹*map-event*$_{ptick}$ *f g* −‘ *X* = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
have ‹(*map* (*map-event*$_{ptick}$ *f g*) *t-P*, (*map-event*$_{ptick}$ *f g*) ‘ *X-P*) ∈ $\mathcal{F}$ (*Renaming P f g*)›
by (*simp add*: *F-Renaming*)
(*metis* ∗∗(*1*) *bij-betw-def bij-map-event*$_{ptick}$ *inj-vimage-image-eq*)
moreover have ‹(*map* (*map-event*$_{ptick}$ *f g*) *t-Q*, (*map-event*$_{ptick}$ *f g*) ‘ *X-Q*) ∈ $\mathcal{F}$ (*Renaming Q f g*)›
by (*simp add*: *F-Renaming*)
(*metis* ∗∗(*2*) *bij-betw-imp-inj-on bij-map-event*$_{ptick}$ *inj-vimage-image-eq*)
moreover have ‹*s setinterleaves* ((*map* (*map-event*$_{ptick}$ *f g*) *t-P*, *map* (*map-event*$_{ptick}$ *f g*) *t-Q*),
range tick ∪ *ev* ‘ *f* ‘ *S*)›
by (*metis* ∗(*2*) ∗∗(*3*) *bij-map-event*$_{ptick}$ *sets-S-eq*
bij-map-setinterleaving-iff-setinterleaving)
moreover have ‹*X* = ((*map-event*$_{ptick}$ *f g*) ‘ *X-P* ∪ (*map-event*$_{ptick}$ *f g*) ‘ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *f* ‘ *S*) ∪
(*map-event*$_{ptick}$ *f g*) ‘ *X-P* ∩ (*map-event*$_{ptick}$ *f g*) ‘ *X-Q*›
apply (*rule inj-image-eq-iff*[*THEN iffD1*, *OF inj-inv-map-event*$_{ptick}$])

apply (*subst bij-vimage-eq-inv-image*[*OF bij-map-event*$_{ptick}$, *symmetric*])
apply (*subst* ∗∗(*4*), *fold image-Un sets-S-eq*)
apply (*subst* (*1 2*) *image-Int*[*OF inj-map-event*$_{ptick}$, *symmetric*])
apply (*fold image-Un*)
apply (*subst image-inv-f-f*[*OF inj-map-event*$_{ptick}$])
by *simp*
ultimately show ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?rhs P Q*)›
by (*simp add*: *F-Sync*) *blast*
qed
qed
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ (*?lhs P Q*) = $\mathcal{D}$ (*?rhs P Q*)›
assume ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?rhs P Q*)›
then consider ‹*s* ∈ $\mathcal{D}$ (*?rhs P Q*)›
| *t-P t-Q X-P X-Q* **where**
‹(*t-P*, *X-P*) ∈ $\mathcal{F}$ (*Renaming P f g*)› ‹(*t-Q*, *X-Q*) ∈ $\mathcal{F}$ (*Renaming Q f g*)›
‹*s setinterleaves* ((*t-P*, *t-Q*), *range tick* ∪ *ev* ‘ *f* ‘ *S*)›
‹*X* = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *f* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
by (*simp add*: *F-Sync D-Sync*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?lhs P Q*)›
proof *cases*
from *same-div D-F* **show** ‹*s* ∈ $\mathcal{D}$ (*?rhs P Q*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ (*?lhs P Q*)› **by** *blast*
next
fix *t-P t-Q X-P X-Q*
assume ∗ : ‹(*t-P*, *X-P*) ∈ $\mathcal{F}$ (*Renaming P f g*)› ‹(*t-Q*, *X-Q*) ∈ $\mathcal{F}$ (*Renaming Q f g*)›
‹*s setinterleaves* ((*t-P*, *t-Q*), *range tick* ∪ *ev* ‘ *f* ‘ *S*)›
‹*X* = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *f* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
from ∗(*1*, *2*) **consider** ‹*t-P* ∈ $\mathcal{D}$ (*Renaming P f g*) ∨ *t-Q* ∈ $\mathcal{D}$ (*Renaming Q f g*)›
| *t-P1 t-Q1* **where** ‹(*t-P1*, *map-event*$_{ptick}$ *f g* −‘ *X-P*) ∈ $\mathcal{F}$ *P*› ‹*t-P* = *map* (*map-event*$_{ptick}$ *f g*) *t-P1*›
‹(*t-Q1*, *map-event*$_{ptick}$ *f g* −‘ *X-Q*) ∈ $\mathcal{F}$ *Q*› ‹*t-Q* = *map* (*map-event*$_{ptick}$ *f g*) *t-Q1*›
by (*simp add*: *F-Renaming D-Renaming*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?lhs P Q*)›
proof *cases*
assume ‹*t-P* ∈ $\mathcal{D}$ (*Renaming P f g*) ∨ *t-Q* ∈ $\mathcal{D}$ (*Renaming Q f g*)›
hence ‹*s* ∈ $\mathcal{D}$ (*?rhs P Q*)›
apply (*simp add*: *D-Sync*)
using ∗(*1*, *2*, *3*) *F-T setinterleaving-sym front-tickFree-Nil* **by** *blast*
with *same-div D-F* **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?lhs P Q*)› **by** *blast*
next
fix *t-P1 t-Q1*
assume ∗∗ : ‹(*t-P1*, *map-event*$_{ptick}$ *f g* −‘ *X-P*) ∈ $\mathcal{F}$ *P*› ‹*t-P* = *map* (*map-event*$_{ptick}$ *f g*) *t-P1*›
‹(*t-Q1*, *map-event*$_{ptick}$ *f g* −‘ *X-Q*) ∈ $\mathcal{F}$ *Q*› ‹*t-Q* = *map*

```
(map-event_ptick f g) t-Q1›
      from **(2, 4) have *** : ‹t-P1 = map (inv (map-event_ptick f g)) t-P›
                               ‹t-Q1 = map (inv (map-event_ptick f g)) t-Q›
        by (simp-all add: inj-map-event_ptick)
       have **** : ‹map (map-event_ptick f g) (map (inv (map-event_ptick f g)) s)
= s›
         by (metis bij-betw-imp-surj bij-map-event_ptick list.map-comp list.map-id
surj-iff)
       from bij-map-setinterleaving-iff-setinterleaving
         [of ‹inv (map-event_ptick f g)› s t-P ‹range tick ∪ ev ‘ f ‘ S› t-Q, simplified
*(3)]
       have ‹map (inv (map-event_ptick f g)) s setinterleaves ((t-P1, t-Q1), range
tick ∪ ev ‘ S)›
       by (metis *** bij-betw-def bij-map-event_ptick inj-imp-bij-betw-inv sets-S-eq)
      moreover have ‹map-event_ptick f g −‘ X = (map-event_ptick f g −‘ X-P ∪
map-event_ptick f g −‘ X-Q) ∩ (range tick ∪ ev ‘ S) ∪
                    map-event_ptick f g −‘ X-P ∩ map-event_ptick f g −‘ X-Q›
        by (metis (no-types, lifting) *(4) inj-map-event_ptick inj-vimage-image-eq
sets-S-eq vimage-Int vimage-Un)
       ultimately show ‹(s, X) ∈ ℱ (?lhs P Q)›
        by (simp add: F-Renaming F-Sync)
          (metis **(1, 3) ****)
     qed
   qed
  qed
qed
```

9.3 (\) and *Mprefix*

We already have a way to distribute the (\) operator on the *Mprefix* operator with $S \cap A = \{\} \Longrightarrow$ *Mprefix S ?P* $\setminus$ *A* $= \Box a{\in}S \rightarrow$ (*?P a* $\setminus$ *A*). But this is only usable when $A \cap S = \{\}$. With the ($\triangleright$) operator, we can now handle the case $A \cap S \neq \{\}$.

9.3.1 (\) and *Mprefix* for disjoint Sets

This is a result similar to *?A* $\cap$ *?S* $= \{\} \Longrightarrow$ *Mprefix ?A ?P* $\setminus$ *?S* $= \Box a{\in}$*?A* $\rightarrow$ (*?P a* $\setminus$ *?S*) when we add a ($\triangleright$) in the expression.

```
theorem Hiding-Mprefix-Sliding-disjoint:
  ‹((□a ∈ A → P a) ▷ Q) \ S = (□a ∈ A → (P a \ S)) ▷ (Q \ S)›
  if disjoint: ‹A ∩ S = {}›
proof (subst Hiding-Mprefix-disjoint[OF disjoint, symmetric])
  show ‹((□a ∈ A → P a) ▷ Q) \ S = (□a ∈ A → P a \ S) ▷ (Q \ S)›
   (is ‹?lhs = ?rhs›)
  proof (subst Process-eq-spec-optimized, safe)
    fix s
    assume ‹s ∈ 𝒟 ?lhs›
```

hence ‹$s \in \mathcal{D}\ (\mathit{Mprefix}\ A\ P \sqcap Q \setminus S)$›
by (*simp add*: *D-Hiding D-Sliding D-Ndet T-Sliding T-Ndet*)
thus ‹$s \in \mathcal{D}$ *?rhs*›
by (*rule set-rev-mp*) (*simp add*: *D-Ndet D-Sliding Hiding-distrib-Ndet*)
next
fix s
assume ‹$s \in \mathcal{D}$ *?rhs*›
hence ‹$s \in \mathcal{D}\ (Q \setminus S) \vee s \in \mathcal{D}\ (\Box a \in A \rightarrow P\ a \setminus S)$›
by (*simp add*: *Hiding-Mprefix-disjoint*[*OF disjoint*]
D-Ndet D-Sliding) *blast*
thus ‹$s \in \mathcal{D}$ *?lhs*›
by (*auto simp add*: *D-Hiding D-Sliding T-Sliding*)
next
fix $s\ X$
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹$(s, X) \in \mathcal{F}$ *?lhs*›
then consider ‹$s \in \mathcal{D}$ *?lhs*›
| ‹$\exists t.\ s = \mathit{trace\text{-}hide}\ t\ (ev\ `\ S) \wedge (t, X \cup ev\ `\ S) \in \mathcal{F}\ (\mathit{Mprefix}\ A\ P \rhd Q)$›
by (*simp add*: *F-Hiding D-Hiding*) *blast*
thus ‹$(s, X) \in \mathcal{F}$ *?rhs*›
proof *cases*
from *same-div D-F* **show** ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow (s, X) \in \mathcal{F}$ *?rhs*› **by** *blast*
next
assume ‹$\exists t.\ s = \mathit{trace\text{-}hide}\ t\ (ev\ `\ S) \wedge$
$(t, X \cup ev\ `\ S) \in \mathcal{F}\ (\mathit{Mprefix}\ A\ P \rhd Q)$›
then obtain t
where $*$: ‹$s = \mathit{trace\text{-}hide}\ t\ (ev\ `\ S)$›
‹$(t, X \cup ev\ `\ S) \in \mathcal{F}\ (\mathit{Mprefix}\ A\ P \rhd Q)$› **by** *blast*
from $*(2)$ **consider** ‹$(t, X \cup ev\ `\ S) \in \mathcal{F}\ Q$›
| ‹$t \neq []$› ‹$(t, X \cup ev\ `\ S) \in \mathcal{F}\ (\mathit{Mprefix}\ A\ P)$›
by (*simp add*: *F-Sliding D-Mprefix*) *blast*
thus ‹$(s, X) \in \mathcal{F}$ *?rhs*›
proof *cases*
have ‹$(t, X \cup ev\ `\ S) \in \mathcal{F}\ Q \Longrightarrow (s, X) \in \mathcal{F}\ (Q \setminus S)$›
by (*auto simp add*: *F-Hiding* $*(1)$)
thus ‹$(t, X \cup ev\ `\ S) \in \mathcal{F}\ Q \Longrightarrow (s, X) \in \mathcal{F}$ *?rhs*›
by (*simp add*: *F-Ndet F-Sliding* $*(1)$)
next
assume *assms* : ‹$t \neq []$› ‹$(t, X \cup ev\ `\ S) \in \mathcal{F}\ (\mathit{Mprefix}\ A\ P)$›
with *disjoint* **have** ‹*trace-hide t* $(ev\ `\ S) \neq []$›
by (*cases t*, *auto simp add*: *F-Mprefix*)
also have ‹$(s, X) \in \mathcal{F}\ (\mathit{Mprefix}\ A\ P \setminus S)$›
using *assms* **by** (*auto simp*: *F-Hiding* $*(1)$)
ultimately show ‹$(s, X) \in \mathcal{F}$ *?rhs*›
by (*simp add*: *F-Sliding* $*(1)$)
qed
qed
next
have $*$: ‹$t \in \mathcal{T}\ (\mathit{Mprefix}\ A\ P) \Longrightarrow \mathit{trace\text{-}hide}\ t\ (ev\ `\ S) = [] \longleftrightarrow t = []$› **for** t

```
    using disjoint by (cases t, auto simp add: T-Mprefix)
  have ** : ‹[] ∉ 𝒟 (Mprefix A P \ S)›
    apply (rule ccontr, simp add: D-Hiding, elim exE conjE disjE)
    by (use * D-T Nil-notin-D-Mprefix in blast)
     (metis (no-types, lifting) * UNIV-I f-inv-into-f old.nat.distinct(2) strict-mono-on-eqD)
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?rhs›
  with ** consider ‹(s, X) ∈ ℱ (Q \ S)›
    | ‹s ≠ []› ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
   by (simp add: Hiding-Mprefix-disjoint[OF disjoint] F-Sliding D-Mprefix) blast
  thus ‹(s, X) ∈ ℱ ?lhs›
  proof cases
    assume ‹(s, X) ∈ ℱ (Q \ S)›
    then consider ‹s ∈ 𝒟 (Q \ S)›
      | ‹∃ t. s = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ Q›
      by (simp add: F-Hiding D-Hiding) blast
    thus ‹(s, X) ∈ ℱ ?lhs›
    proof cases
      assume ‹s ∈ 𝒟 (Q \ S)›
      hence ‹s ∈ 𝒟 ?rhs› by (simp add: D-Ndet D-Sliding)
      with same-div D-F show ‹(s, X) ∈ ℱ ?lhs› by blast
    next
      assume ‹∃ t. s = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ Q›
      thus ‹(s, X) ∈ ℱ ?lhs›
        by (simp add: F-Hiding F-Sliding) blast
    qed
  next
    assume assms : ‹s ≠ []› ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
    then consider ‹s ∈ 𝒟 (Mprefix A P \ S)›
      | ‹∃ t. t ≠ [] ∧ s = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A
P)›
      by (simp add: F-Hiding D-Hiding) (metis filter.simps(1))
    thus ‹(s, X) ∈ ℱ ?lhs›
    proof cases
      assume ‹s ∈ 𝒟 (Mprefix A P \ S)›
      hence ‹s ∈ 𝒟 ?rhs› by (simp add: D-Ndet D-Sliding)
      with same-div D-F show ‹(s, X) ∈ ℱ ?lhs› by blast
    next
      show ‹∃ t. t ≠ [] ∧ s = trace-hide t (ev ‘ S) ∧
              (t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P) ⟹ (s, X) ∈ ℱ ?lhs›
        by (auto simp add: F-Sliding F-Hiding)
    qed
  qed
 qed
qed
```

9.3.2 $(\setminus)$ and *Mprefix* for non-disjoint Sets

Finally the new version, when $A \cap S \neq \{\}$.

lemma ‹$\exists\, A :: nat\ set.\ \exists P\ S.$
$A \cap S = \{\} \wedge \Box a \in A \rightarrow P\ a \setminus S \neq$
$(\Box a \in (A - S) \rightarrow (P\ a \setminus S)) \rhd (\sqcap a \in (A \cap S).\ (P\ a \setminus S))$›
proof (*intro exI*)
show ‹$\{0\} \cap \{Suc\ 0\} = \{\} \wedge$
$\Box a \in \{0\} \rightarrow SKIP\ undefined \setminus \{Suc\ 0\} \neq$
$(\Box a \in (\{0\} - \{Suc\ 0\}) \rightarrow (SKIP\ undefined \setminus \{Suc\ 0\})) \rhd (\sqcap a \in (\{0\} \cap \{Suc\ 0\}).\ (SKIP\ undefined \setminus \{Suc\ 0\}))$›
apply (*simp add*: *write0-def*[*symmetric*] *Hiding-write0-disjoint*)
using *UNIV-I list.discI* **by** (*auto simp add*: *Process-eq-spec write0-def F-Ndet F-Mprefix F-Sliding F-STOP set-eq-iff*)
qed

This is a result similar to $?A \cap ?S \neq \{\} \Longrightarrow Mprefix\ ?A\ ?P \setminus ?S = (\Box a{\in}(?A - ?S) \rightarrow (?P\ a \setminus ?S)) \rhd (\sqcap a{\in}(?A \cap ?S).\ (?P\ a \setminus ?S))$ when we add a $(\rhd)$ in the expression.

lemma *Hiding-Mprefix-Sliding-non-disjoint*:
‹$(\Box a \in A \rightarrow P\ a) \rhd Q \setminus S = (\Box a \in (A - S) \rightarrow (P\ a \setminus S)) \rhd (Q \setminus S) \sqcap (\sqcap a \in (A \cap S).\ (P\ a \setminus S))$›
if *non-disjoint*: ‹$A \cap S \neq \{\}$›
proof (*subst Sliding-distrib-Ndet-left*,
subst Hiding-Mprefix-non-disjoint[*OF non-disjoint*, *symmetric*])
show ‹$Mprefix\ A\ P \rhd Q \setminus S =$
$((\Box a \in (A - S) \rightarrow (P\ a \setminus S)) \rhd (Q \setminus S)) \sqcap (\Box a \in A \rightarrow P\ a \setminus S)$›
(**is** ‹$?lhs = ?rhs$›)
proof (*subst Process-eq-spec-optimized*, *safe*)
fix s
assume ‹$s \in \mathcal{D}\ ?lhs$›
hence ‹$s \in \mathcal{D}\ (Mprefix\ A\ P \sqcap Q \setminus S)$›
by (*simp add*: *D-Hiding D-Sliding D-Ndet T-Sliding T-Ndet*)
thus ‹$s \in \mathcal{D}\ ?rhs$›
by (*rule set-rev-mp*)
(*simp add*: *D-Ndet D-Sliding Hiding-distrib-Ndet subsetI*)
next
fix s
assume ‹$s \in \mathcal{D}\ ?rhs$›
hence ‹$s \in \mathcal{D}\ (Q \setminus S) \vee s \in \mathcal{D}\ (\Box a \in A \rightarrow P\ a \setminus S)$›
by (*simp add*: *Hiding-Mprefix-non-disjoint*[*OF non-disjoint*] *D-Ndet D-Sliding*) *blast*
thus ‹$s \in \mathcal{D}\ ?lhs$›
by (*auto simp add*: *D-Hiding D-Sliding T-Sliding*)
next
fix $s\ X$
assume *same-div* : ‹$\mathcal{D}\ ?lhs = \mathcal{D}\ ?rhs$›
assume ‹$(s, X) \in \mathcal{F}\ ?lhs$›
then consider ‹$s \in \mathcal{D}\ ?lhs$›

```
    |‹∃ t. s = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P ▷ Q)›
    by (simp add: F-Hiding D-Hiding) blast
  thus ‹(s, X) ∈ ℱ ?rhs›
  proof cases
    from same-div D-F show ‹s ∈ 𝒟 ?lhs ⟹ (s, X) ∈ ℱ ?rhs› by blast
  next
    assume ‹∃ t. s = trace-hide t (ev ‘ S) ∧
              (t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P ▷ Q)›
    then obtain t
      where ∗ : ‹s = trace-hide t (ev ‘ S)›
              ‹(t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P ▷ Q)› by blast
    from ∗(2) consider ‹(t, X ∪ ev ‘ S) ∈ ℱ Q›
      | ‹(t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)›
      by (simp add: F-Sliding D-Mprefix) blast
    thus ‹(s, X) ∈ ℱ ?rhs›
    proof cases
      have ‹(t, X ∪ ev ‘ S) ∈ ℱ Q ⟹ (s, X) ∈ ℱ (Q \ S)›
        by (auto simp add: F-Hiding ∗(1))
      thus ‹(t, X ∪ ev ‘ S) ∈ ℱ Q ⟹ (s, X) ∈ ℱ ?rhs›
        by (simp add: F-Ndet F-Sliding ∗(1))
    next
      assume ‹(t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)›
      hence ‹(s, X) ∈ ℱ (Mprefix A P \ S)› by (auto simp: F-Hiding ∗(1))
      thus ‹(s, X) ∈ ℱ ?rhs› by (simp add: F-Ndet ∗(1))
    qed
  qed
next
  fix s X
  have ∗ : ‹s ≠ [] ⟹ (s, X) ∈ ℱ (□a ∈ (A − S) → (P a \ S)) ⟹
                      (s, X) ∈ ℱ (□a ∈ A → P a \ S)›
    by (simp add: Hiding-Mprefix-non-disjoint[OF non-disjoint] F-Sliding)
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?rhs›
  with ∗ consider ‹(s, X) ∈ ℱ (Q \ S)› | ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
    by (auto simp add: F-Ndet F-Sliding)
  thus ‹(s, X) ∈ ℱ ?lhs›
  proof cases
    assume ‹(s, X) ∈ ℱ (Q \ S)›
    then consider ‹s ∈ 𝒟 (Q \ S)›
      | ‹∃ t. s = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ Q›
      by (simp add: F-Hiding D-Hiding) blast
    thus ‹(s, X) ∈ ℱ ?lhs›
    proof cases
      assume ‹s ∈ 𝒟 (Q \ S)›
      hence ‹s ∈ 𝒟 ?rhs› by (simp add: D-Ndet D-Sliding)
      with same-div D-F show ‹(s, X) ∈ ℱ ?lhs› by blast
    next
      assume ‹∃ t. s = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ Q›
      thus ‹(s, X) ∈ ℱ ?lhs›
```

```
        by (simp add: F-Hiding F-Sliding) blast
    qed
  next
    assume ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
    then consider ‹s ∈ 𝒟 (Mprefix A P \ S)›
      | ‹∃ t. s = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)›
      by (simp add: F-Hiding D-Hiding) blast
    thus ‹(s, X) ∈ ℱ ?lhs›
    proof cases
      assume ‹s ∈ 𝒟 (Mprefix A P \ S)›
      hence ‹s ∈ 𝒟 ?rhs› by (simp add: D-Ndet D-Sliding)
      with same-div D-F show ‹(s, X) ∈ ℱ ?lhs› by blast
    next
     assume ‹∃ t. s = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)›
      then obtain t where ∗ : ‹s = trace-hide t (ev ‘ S)›
                              ‹(t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)› by blast
      from ∗(2) non-disjoint have ‹t ≠ []› by (simp add: F-Mprefix) blast
      with ∗ show ‹(s, X) ∈ ℱ ?lhs›
        by (simp add: F-Hiding F-Sliding) blast
    qed
  qed
qed
```

9.4 (▷) behaviour

We already proved several laws for the (▷) operator. Here we give other results in the same spirit as *Hiding-Mprefix-Sliding-disjoint* and *Hiding-Mprefix-Sliding-non-disjoin*

```
lemma Mprefix-Sliding-Mprefix-Sliding:
  ‹(□a ∈ A → P a) ▷ (□b ∈ B → Q b) ▷ R =
   (□ x ∈ (A ∪ B) → (if x ∈ A ∩ B then P x ⊓ Q x else if x ∈ A then P x else Q
x)) ▷ R›
  (is ‹(□a ∈ A → P a) ▷ (□b ∈ B → Q b) ▷ R = ?term ▷ R›)
proof (subst Sliding-def, subst Mprefix-Det-Mprefix)
  have ‹Mprefix B Q ⊓ (Mprefix A P □ Mprefix B Q) ▷ R = Mprefix A P □
Mprefix B Q ▷ R›
    by (metis (no-types, opaque-lifting) Det-assoc Det-commute Ndet-commute
          Ndet-distrib-Det-left Ndet-id Sliding-Det Sliding-assoc Sliding-def)
  thus ‹?term ⊓ Mprefix B Q ▷ R = ?term ▷ R›
    by (simp add: Mprefix-Det-Mprefix Ndet-commute)
qed


lemma Mprefix-Sliding-Seq:
  ‹(□a ∈ A → P a) ▷ P′ ; Q = (□a ∈ A → (P a ; Q)) ▷ (P′ ; Q)›
proof (subst Mprefix-Seq[symmetric])
  show ‹((□a ∈ A → P a) ▷ P′) ; Q =
        ((□a ∈ A → P a) ; Q) ▷ (P′ ; Q)› (is ‹?lhs = ?rhs›)
```

```
  proof (subst Process-eq-spec, safe)
    show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs› for s
      by (simp add: D-Sliding D-Seq T-Sliding) blast
  next
    show ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ?lhs› for s
      by (auto simp add: D-Sliding D-Seq T-Sliding)
  next
    show ‹(s, X) ∈ ℱ ?lhs ⟹ (s, X) ∈ ℱ ?rhs› for s X
      by (cases s; simp add: F-Sliding D-Sliding T-Sliding F-Seq) metis
  next
    show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs› for s X
      by (cases s; simp add: F-Sliding D-Sliding T-Sliding
                             F-Seq D-Seq T-Seq D-Mprefix T-Mprefix)
         (metis event_ptick.simps(4) hd-append list.sel(1), blast)
  qed
qed



lemma Throw-Sliding :
  ‹(□a ∈ A → P a) ▷ P' Θ b ∈ B. Q b =
   (□a ∈ A → (if a ∈ B then Q a else P a Θ b ∈ B. Q b)) ▷ (P' Θ b ∈ B. Q b)›
  (is ‹?lhs = ?rhs›)
proof (subst Process-eq-spec-optimized, safe)
  fix s
  assume ‹s ∈ 𝒟 ?lhs›
  then consider t1 t2 where ‹s = t1 @ t2› ‹t1 ∈ 𝒟 (Mprefix A P ▷ P')›
                            ‹tF t1› ‹set t1 ∩ ev ‘ B = {}› ‹ftF t2›
    | t1 b t2 where ‹s = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (Mprefix A P ▷ P')›
                    ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹t2 ∈ 𝒟 (Q b)›
    by (simp add: D-Throw) blast
  thus ‹s ∈ 𝒟 ?rhs›
  proof cases
    fix t1 t2 assume ∗ : ‹s = t1 @ t2› ‹t1 ∈ 𝒟 (Mprefix A P ▷ P')› ‹tF t1›
                         ‹set t1 ∩ ev ‘ B = {}› ‹ftF t2›
    from ∗(2) consider ‹t1 ∈ 𝒟 (Mprefix A P)› | ‹t1 ∈ 𝒟 P'›
      by (simp add: D-Sliding) blast
    thus ‹s ∈ 𝒟 ?rhs›
    proof cases
      assume ‹t1 ∈ 𝒟 (Mprefix A P)›
      then obtain a t1' where ‹t1 = ev a # t1'› ‹a ∈ A› ‹t1' ∈ 𝒟 (P a)›
        by (auto simp add: D-Mprefix image-iff)
      with ∗(1, 3, 4, 5) show ‹s ∈ 𝒟 ?rhs›
        by (simp add: D-Sliding D-Mprefix D-Throw) (metis image-eqI)
    next
      from ∗(1, 3, 4, 5)  show ‹t1 ∈ 𝒟 P' ⟹ s ∈ 𝒟 ?rhs›
        by (simp add: D-Sliding D-Throw) blast
    qed
  next
```

```
    fix t1 b t2 assume ∗ : ‹s = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (Mprefix A P ▷
P')›
                         ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹t2 ∈ 𝒟 (Q b)›
    from ∗(2) consider ‹t1 @ [ev b] ∈ 𝒯 (Mprefix A P)› | ‹t1 @ [ev b] ∈ 𝒯 P'›
      by (simp add: T-Sliding) blast
    thus ‹s ∈ 𝒟 ?rhs›
    proof cases
      assume ‹t1 @ [ev b] ∈ 𝒯 (Mprefix A P)›
      then obtain a t1'
        where ‹t1 @ [ev b] = ev a # t1'› ‹a ∈ A› ‹t1' ∈ 𝒯 (P a)›
        by (auto simp add: T-Mprefix)
      with ∗(1, 3, 4, 5) show ‹s ∈ 𝒟 ?rhs›
        by (cases t1; simp add: ∗(1) D-Sliding D-Mprefix D-Throw)
           (metis image-eqI)
    next
      from ∗(1, 3, 4, 5) show ‹t1 @ [ev b] ∈ 𝒯 P' ⟹ s ∈ 𝒟 ?rhs›
        by (simp add: D-Sliding D-Mprefix D-Throw) blast
    qed
  qed
next
  fix s
  assume ‹s ∈ 𝒟 ?rhs›
  then consider ‹s ∈ 𝒟 (Throw P' B Q)›
    | ‹s ∈ 𝒟 (□a∈A → (if a ∈ B then Q a else Throw (P a) B Q))›
    by (simp add: D-Sliding) blast
  thus ‹s ∈ 𝒟 ?lhs›
  proof cases
    show ‹s ∈ 𝒟 (Throw P' B Q) ⟹ s ∈ 𝒟 ?lhs›
      by (simp add: D-Throw D-Sliding T-Sliding) blast
  next
    assume ‹s ∈ 𝒟 (□a∈A → (if a ∈ B then Q a else Throw (P a) B Q))›
    then obtain a s'
      where ∗ : ‹s = ev a # s'› ‹a ∈ A›
                ‹s' ∈ 𝒟 (if a ∈ B then Q a else Throw (P a) B Q)›
      by (cases s; simp add: D-Mprefix) blast
    show ‹s ∈ 𝒟 ?lhs›
    proof (cases ‹a ∈ B›)
      from ∗ show ‹a ∈ B ⟹ s ∈ 𝒟 ?lhs›
        by (simp add: D-Throw T-Sliding T-Mprefix disjoint-iff)
           (metis Nil-elem-T emptyE empty-set self-append-conv2)
    next
      assume ‹a ∉ B›
      with ∗(3) consider
        t1 t2 where ‹s' = t1 @ t2› ‹t1 ∈ 𝒟 (P a)› ‹tF t1›
                    ‹set t1 ∩ ev ‘ B = {}› ‹ftF t2›
      | t1 b t2 where ‹s' = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (P a)›
                      ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹t2 ∈ 𝒟 (Q b)›
        by (simp add: D-Throw) blast
      thus ‹s ∈ 𝒟 ?lhs›
```

```
    proof cases
      fix t1 t2 assume ** : ‹s' = t1 @ t2› ‹t1 ∈ 𝒟 (P a)› ‹tF t1›
                           ‹set t1 ∩ ev ‘ B = {}› ‹ftF t2›
      have *** : ‹s = (ev a # t1) @ t2 ∧ set (ev a # t1) ∩ ev ‘ B = {}›
        by (simp add: *(1) **(1, 4) image-iff ‹a ∉ B›)
      from * **(1, 2, 3, 5) show ‹s ∈ 𝒟 ?lhs›
        by (simp add: D-Throw D-Sliding D-Mprefix image-iff)
           (metis *** event_ptick.disc(1) tickFree-Cons-iff)
    next
      fix t1 b t2 assume ** : ‹s' = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (P a)›
                             ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹t2 ∈ 𝒟 (Q b)›
      have *** : ‹s = (ev a # t1 @ [ev b]) @ t2 ∧ set (ev a # t1) ∩ ev ‘ B = {}›
        by (simp add: *(1) **(1, 3) image-iff ‹a ∉ B›)
      from * **(1, 2, 4, 5) show ‹s ∈ 𝒟 ?lhs›
       by (simp add: D-Throw T-Sliding T-Mprefix) (metis *** Cons-eq-appendI)
    qed
  qed
next
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?lhs›
  then consider ‹s ∈ 𝒟 ?lhs›
    | ‹(s, X) ∈ ℱ (Mprefix A P ▷ P')› ‹set s ∩ ev ‘ B = {}›
    | t1 b t2 where ‹s = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (Mprefix A P ▷ P')›
                    ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹(t2, X) ∈ ℱ (Q b)›
    by (auto simp add: F-Throw D-Throw)
  thus ‹(s, X) ∈ ℱ ?rhs›
  proof cases
    from same-div D-F show ‹s ∈ 𝒟 ?lhs ⟹ (s, X) ∈ ℱ ?rhs› by blast
  next
    show ‹(s, X) ∈ ℱ (Mprefix A P ▷ P') ⟹ set s ∩ ev ‘ B = {} ⟹ (s, X) ∈
ℱ ?rhs›
      by (cases s; simp add: F-Sliding F-Mprefix F-Throw) blast
  next
    fix t1 b t2 assume * : ‹s = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (Mprefix A P ▷
P')›
                          ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹(t2, X) ∈ ℱ (Q b)›
    from *(2) consider ‹t1 @ [ev b] ∈ 𝒯 (Mprefix A P)› | ‹t1 @ [ev b] ∈ 𝒯 P'›
      by (simp add: T-Sliding) blast
    thus ‹(s, X) ∈ ℱ ?rhs›
    proof cases
      assume ‹t1 @ [ev b] ∈ 𝒯 (Mprefix A P)›
      then obtain a t1'
        where ‹t1 @ [ev b] = ev a # t1'› ‹a ∈ A› ‹t1' ∈ 𝒯 (P a)›
        by (auto simp add: T-Mprefix)
      with *(1, 3, 4, 5) show ‹(s, X) ∈ ℱ ?rhs›
        by (cases t1; simp add: *(1) F-Sliding F-Mprefix F-Throw) blast
    next
```

```
    from *(1, 3, 4, 5) show ‹t1 @ [ev b] ∈ 𝒯 P' ⟹ (s, X) ∈ ℱ ?rhs›
      by (simp add: F-Sliding F-Mprefix F-Throw) blast
  qed
 qed
next
 fix s X
 assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
 assume ‹(s, X) ∈ ℱ ?rhs›
 then consider ‹s ∈ 𝒟 ?rhs› | ‹(s, X) ∈ ℱ (Throw P' B Q)›
  | ‹s ≠ []› ‹(s, X) ∈ ℱ (□a ∈ A → (if a ∈ B then Q a else Throw (P a) B Q))›
  by (simp add: F-Sliding D-Sliding) blast
 thus ‹(s, X) ∈ ℱ ?lhs›
 proof cases
  from same-div D-F show ‹s ∈ 𝒟 ?rhs ⟹ (s, X) ∈ ℱ ?lhs› by blast
 next
  show ‹(s, X) ∈ ℱ (Throw P' B Q) ⟹ (s, X) ∈ ℱ ?lhs›
   by (simp add: F-Throw F-Sliding D-Sliding T-Sliding) blast
 next
  assume ‹s ≠ []› ‹(s, X) ∈ ℱ (□a ∈ A → (if a ∈ B then Q a else Throw (P a)
B Q))›
  then obtain a s'
   where * : ‹s = ev a # s'› ‹a ∈ A›
         ‹(s', X) ∈ ℱ (if a ∈ B then Q a else Throw (P a) B Q)›
   by (auto simp add: F-Mprefix image-iff)
  show ‹(s, X) ∈ ℱ ?lhs›
  proof (cases ‹a ∈ B›)
   assume ‹a ∈ B›
   have ‹[ev a] ∈ 𝒯 (Mprefix A P ▷ P')›
    by (simp add: T-Sliding T-Mprefix *(2))
   with *(1, 3) ‹a ∈ B› show ‹(s, X) ∈ ℱ ?lhs›
    by (simp add: F-Throw) (metis append-Nil empty-set inf-bot-left)
  next
   assume ‹a ∉ B›
   with *(3) consider ‹s' ∈ 𝒟 (Throw (P a) B Q)›
    | ‹(s', X) ∈ ℱ (P a)› ‹set s' ∩ ev ‘ B = {}›
    | t1 b t2 where ‹s' = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (P a)›
                    ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹(t2, X) ∈ ℱ (Q b)›
    by (auto simp add: F-Throw D-Throw)
   thus ‹(s, X) ∈ ℱ ?lhs›
   proof cases
    assume ‹s' ∈ 𝒟 (Throw (P a) B Q)›
    hence ‹s ∈ 𝒟 ?rhs› by (simp add: *(1, 2) D-Sliding D-Mprefix ‹a ∉ B›)
    with same-div D-F show ‹(s, X) ∈ ℱ ?lhs› by blast
   next
    show ‹(s', X) ∈ ℱ (P a) ⟹ set s' ∩ ev ‘ B = {} ⟹ (s, X) ∈ ℱ ?lhs›
     using *(1, 2) ‹a ∉ B› by (simp add: F-Throw F-Sliding F-Mprefix) blast
   next
    fix t1 b t2 assume ** : ‹s' = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (P a)›
                            ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹(t2, X) ∈ ℱ (Q b)›
```

from ∗∗ **have** ∗∗∗ : ‹(*ev a # t1*) @ [*ev b*] ∈ $\mathcal{T}$ (*Mprefix A P*) ∧
set (*ev a # t1*) ∩ *ev ' B* = {}›
by (*simp add*: *T-Mprefix* ∗(*2*) *image-iff* ‹*a* ∉ *B*›)
from ∗(*1*) ∗∗(*1*, *4*, *5*) ∗∗(*5*) **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*simp add*: *F-Throw T-Sliding*) (*metis* ∗∗∗ *Cons-eq-appendI*)
qed
qed
qed
qed

9.5 Dealing with *SKIP*

lemma *Renaming-Mprefix-Det-SKIP*:
‹*Renaming* ((□ *a* ∈ *A* → *P a*) □ *SKIP r*) *f g* =
(□*y*∈*f ' A* → ⊓ *a* ∈ {*x* ∈ *A*. *y* = *f x*}. *Renaming* (*P a*) *f g*) □ *SKIP* (*g r*)›

by (*simp add*: *Renaming-Det Renaming-Mprefix*)

lemma *Mprefix-Sliding-SKIP-Seq*: ‹((□ *a* ∈ *A* → *P a*) ▷ *SKIP r*) ; *Q* = (□ *a* ∈ *A* → (*P a* ; *Q*)) ▷ *Q*›

by (*simp del*: *Sliding-SKIP add*: *Mprefix-Sliding-Seq*)

lemma *Mprefix-Det-SKIP-Seq*: ‹((□ *a* ∈ *A* → *P a*) □ *SKIP r*) ; *Q* = (□ *a* ∈ *A* → (*P a* ; *Q*)) ▷ *Q*›
by (*fold Sliding-SKIP*) (*fact Mprefix-Sliding-SKIP-Seq*)

lemma *Sliding-Ndet-pseudo-assoc* : ‹(*P* ▷ *Q*) ⊓ *R* = *P* ▷ *Q* ⊓ *R*›
by (*metis Ndet-assoc Ndet-distrib-Det-right Ndet-id Sliding-def*)

lemma *Hiding-Mprefix-Det-SKIP*:
‹(□ *a* ∈ *A* → *P a*) □ *SKIP r* \ *S* =
(*if A* ∩ *S* = {} *then* (□ *a* ∈ *A* → (*P a* \ *S*)) □ *SKIP r*
else ((□ *a* ∈ (*A* − *S*) → (*P a* \ *S*)) □ *SKIP r*) ⊓ (⊓ *a* ∈ (*A* ∩ *S*). (*P a* \ *S*)))›
by (*fold Sliding-SKIP*)
(*simp del*: *Sliding-SKIP add*: *Hiding-Mprefix-Sliding-disjoint*
Hiding-Mprefix-Sliding-non-disjoint Sliding-Ndet-pseudo-assoc)

lemma ‹*s* ≠ [] ⟹ (*s*, *X*) ∈ $\mathcal{F}$ (*P* □ *Q*) ⟷ (*s*, *X*) ∈ $\mathcal{F}$ (*P* ⊓ *Q*)›
by (*simp add*: *F-Det F-Ndet*)

lemma *Mprefix-Det-SKIP-Sync-SKIP* :
‹((□ *a* ∈ *A* → *P a*) □ *SKIP res*) ⟦*S*⟧ *SKIP res'* =
(*if res* = *res'* *then* (□ *a* ∈ (*A* − *S*) → (*P a* ⟦*S*⟧ *SKIP res'*)) □ *SKIP res'*

else $(\Box\ a \in (A - S) \rightarrow (P\ a\ [\![S]\!]\ SKIP\ res')) \sqcap STOP)$›
(**is** ‹*?lhs* = (*if res* = *res' then ?rhs1 else ?rhs2*)›)
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹$s \in \mathcal{D}$ *?lhs*›
then obtain *a t u r v*
where ∗ : ‹*ftF v*› ‹*tF r* ∨ *v* = []› ‹*s* = *r* @ *v*›
‹*r setinterleaves* ((*ev a* # *t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
‹*a* ∈ *A*› ‹*t* ∈ $\mathcal{D}$ (*P a*)› ‹*u* ∈ $\mathcal{T}$ (*SKIP res'*)›
by (*simp add*: *D-Sync D-SKIP D-Det D-Mprefix T-SKIP image-iff*) *blast*
from ∗(*3*, *4*, *5*, *7*) **have** ∗∗ : ‹*a* ∈ *A* − *S*› ‹*s* = *ev a* # *tl r* @ *v*›
‹*tl r setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
by (*auto simp add*: *image-iff T-SKIP split*: *if-split-asm*)
have ‹*tl s* ∈ $\mathcal{D}$ (*P a* [*S*] *SKIP res'*)›
by (*simp add*: *D-Sync*)
(*metis* ∗(*1*, *2*, *6*, *7*) ∗∗(*2*, *3*) *list.sel*(*3*) *tickFree-tl*)
with ∗∗(*1*) **show** ‹*s* ∈ $\mathcal{D}$ (*if res* = *res' then ?rhs1 else ?rhs2*)›
by (*simp add*: *D-Det D-Ndet D-Mprefix* ∗∗(*2*))
next
fix *s*
assume ‹*s* ∈ $\mathcal{D}$ (*if res* = *res' then ?rhs1 else ?rhs2*)›
then obtain *a s'* **where** ∗ : ‹*s* = *ev a* # *s'*› ‹*a* ∈ *A* − *S*› ‹*s'* ∈ $\mathcal{D}$ (*P a* [*S*] *SKIP res'*)›
by (*auto simp add*: *D-Det D-Ndet D-SKIP D-STOP D-Mprefix image-iff split*: *if-split-asm*)
then obtain *t u r v*
where ∗∗ : ‹*ftF v*› ‹*tF r* ∨ *v* = []› ‹*s'* = *r* @ *v*›
‹*r setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
‹*t* ∈ $\mathcal{D}$ (*P a*)› ‹*u* ∈ $\mathcal{T}$ (*SKIP res'*)›
by (*simp add*: *D-Sync D-SKIP*) *blast*
have ‹(*ev a* # *r*) *setinterleaves* ((*ev a* # *t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
using ∗(*2*) ∗∗(*4*, *6*) **by** (*auto simp add*: *T-SKIP*)
with ∗(*2*) ∗∗(*1*, *2*, *3*, *5*, *6*) **show** ‹*s* ∈ $\mathcal{D}$ *?lhs*›
by (*simp add*: *D-Sync D-SKIP D-Det D-Mprefix T-SKIP image-iff*)
(*metis* (*no-types*, *opaque-lifting*) ∗(*1*) *Cons-eq-appendI* $event_{ptick}$.*disc*(*1*) *tickFree-Cons-iff*)
next
fix *s Z*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ (*if res* = *res' then ?rhs1 else ?rhs2*)›
assume ‹(*s*, *Z*) ∈ $\mathcal{F}$ *?lhs*›
then consider ‹*s* ∈ $\mathcal{D}$ *?lhs*›
| *t u X Y* **where** ‹(*t*, *X*) ∈ $\mathcal{F}$ (*Mprefix A P* □ *SKIP res*)› ‹(*u*, *Y*) ∈ $\mathcal{F}$ (*SKIP res'*)›
‹*s setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
‹*Z* = (*X* ∪ *Y*) ∩ (*range tick* ∪ *ev* ‘ *S*) ∪ *X* ∩ *Y*›
by (*simp add*: *F-Sync D-Sync*) *blast*
thus ‹(*s*, *Z*) ∈ $\mathcal{F}$ (*if res* = *res' then ?rhs1 else ?rhs2*)›
proof *cases*
from *same-div D-F* **show** ‹*s* ∈ $\mathcal{D}$ *?lhs* ⟹ (*s*, *Z*) ∈ $\mathcal{F}$ (*if res* = *res' then ?rhs1*

else ?rhs2)› **by** *blast*
next
fix *t u X Y*
assume ∗ : ‹(*t*, *X*) ∈ $\mathcal{F}$ (*Mprefix A P* □ *SKIP res*)› ‹(*u*, *Y*) ∈ $\mathcal{F}$ (*SKIP res′*)›
‹*s setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *S*)›
‹*Z* = (*X* ∪ *Y*) ∩ (*range tick* ∪ *ev* ‘ *S*) ∪ *X* ∩ *Y*›
from ∗(*1*) **consider** ‹*t* = []› | ‹*t* = [✓(*res*)]› | *a t′* **where** ‹*t* = *ev a* # *t′*›
by (*auto simp add*: *F-Det F-SKIP F-Mprefix*)
thus ‹(*s*, *Z*) ∈ $\mathcal{F}$ (*if res* = *res′ then ?rhs1 else ?rhs2*)›
proof *cases*
assume ‹*t* = []›
with ∗(*2*, *3*) **have** ‹*s* = []› **by** (*auto simp add*: *F-SKIP*)
with ‹*t* = []› ∗(*3*)[*simplified* ‹*t* = []›, *THEN emptyLeftProperty*] ∗(*1*, *2*, *3*, *4*)
show ‹(*s*, *Z*) ∈ $\mathcal{F}$ (*if res* = *res′ then ?rhs1 else ?rhs2*)›
by (*auto simp add*: *F-Det F-Ndet F-Mprefix F-STOP F-SKIP D-SKIP T-SKIP*)
next
assume ‹*t* = [✓(*res*)]›
with ∗(*2*, *3*) **have** ‹*res′* = *res* ∧ *s* = [✓(*res*)]›
by (*cases u*; *auto simp add*: *F-SKIP split*: *if-split-asm*)
with ‹*t* = [✓(*res*)]› **show** ‹(*s*, *Z*) ∈ $\mathcal{F}$ (*if res* = *res′ then ?rhs1 else ?rhs2*)›
by (*simp add*: *F-Det F-Ndet F-STOP F-SKIP*)
next
fix *a t′* **assume** ‹*t* = *ev a* # *t′*›
with ∗(*1*, *2*, *3*) *empty-setinterleaving* **obtain** *s′*
where ∗∗ : ‹*s′ setinterleaves* ((*t′*, *u*), *range tick* ∪ *ev* ‘ *S*)›
‹*s* = *ev a* # *s′*› ‹(*t′*, *X*) ∈ $\mathcal{F}$ (*P a*)› ‹*a* ∈ *A* − *S*›
by (*auto simp add*: *F-SKIP F-Det F-Mprefix image-iff split*: *if-split-asm*)
from ∗(*2*, *4*) ∗∗(*1*, *3*) **have** ‹(*s′*, *Z*) ∈ $\mathcal{F}$ (*P a* ⟦*S*⟧ *SKIP res′*)›
by (*simp add*: *F-Sync*) *blast*
with ∗∗(*4*) **show** ‹(*s*, *Z*) ∈ $\mathcal{F}$ (*if res* = *res′ then ?rhs1 else ?rhs2*)›
by (*simp add*: *F-Det F-Ndet* ‹*s* = *ev a* # *s′*› *F-SKIP F-Mprefix*)
qed
qed
next
fix *s Z*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ (*if res* = *res′ then ?rhs1 else ?rhs2*)›
assume ‹(*s*, *Z*) ∈ $\mathcal{F}$ (*if res* = *res′ then ?rhs1 else ?rhs2*)›
then consider ‹*res* = *res′*› ‹(*s*, *Z*) ∈ $\mathcal{F}$ *?rhs1*›
| ‹*res* ≠ *res′*› ‹(*s*, *Z*) ∈ $\mathcal{F}$ *?rhs2*› **by** *presburger*
thus ‹(*s*, *Z*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
assume ‹*res* = *res′*› ‹(*s*, *Z*) ∈ $\mathcal{F}$ *?rhs1*›
then consider ‹*s* = []› | ‹*s* = [✓(*res′*)]› | *a s′* **where** ‹*s* = *ev a* # *s′*›
by (*auto simp add*: *F-Det F-SKIP F-Mprefix*)
thus ‹(*s*, *Z*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
assume ‹*s* = []›
with ‹(*s*, *Z*) ∈ $\mathcal{F}$ *?rhs1*› **have** ‹*tick res′* ∉ *Z*›

by (*auto simp add*: *F-Det F-Mprefix F-SKIP D-SKIP T-SKIP*)
with ‹$s = []$› **show** ‹$(s, Z) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Sync F-Det T-SKIP F-SKIP* ‹*res* = *res'*›)
(*metis Int-Un-eq*(*3*) *si-empty1 Un-Int-eq*(*4*) *Un-commute insertI1*)
next
assume ‹$s = [\checkmark(res')]$›
hence $*$: ‹$([\checkmark(res')], Z) \in \mathcal{F}$ (*Mprefix A P* $\Box$ *SKIP res'*) $\wedge$
s setinterleaves (($[\checkmark(res')]$, $[\checkmark(res')]$), *range tick* $\cup$ *ev* ‘ *S*) $\wedge$
($[\checkmark(res')]$, *Z*) $\in \mathcal{F}$ (*SKIP res'*) $\wedge$ *Z* = (*Z* $\cup$ *Z*) $\cap$ (*range tick* $\cup$ *ev* ‘ *S*) $\cup$ *Z* $\cap$ *Z*›
by (*simp add*: *F-Det F-SKIP*)
show ‹$(s, Z) \in \mathcal{F}$ *?lhs*› **by** (*simp add*: *F-Sync* ‹*res* = *res'*›) (*meson* $*$)
next
fix *a s'* **assume** ‹$s = ev\ a \# s'$›
with ‹$(s, Z) \in \mathcal{F}$ *?rhs1*› **have** $*$: ‹$a \in A$› ‹$a \notin S$› ‹$(s', Z) \in \mathcal{F}$ (*P a* $[\![S]\!]$ *SKIP res'*)›
by (*simp-all add*: *F-Det F-SKIP F-Mprefix image-iff*)
from $*(3)$ **consider** ‹$s' \in \mathcal{D}$ (*P a* $[\![S]\!]$ *SKIP res'*)›
| *t u X Y* **where** ‹$(t, X) \in \mathcal{F}$ (*P a*)› ‹$(u, Y) \in \mathcal{F}$ (*SKIP res'*)›
‹*s'* *setinterleaves* ((*t*, *u*), *range tick* $\cup$ *ev* ‘ *S*)›
‹*Z* = (*X* $\cup$ *Y*) $\cap$ (*range tick* $\cup$ *ev* ‘ *S*) $\cup$ *X* $\cap$ *Y*›
by (*simp add*: *F-Sync D-Sync*) *blast*
thus ‹$(s, Z) \in \mathcal{F}$ *?lhs*›
proof *cases*
assume ‹$s' \in \mathcal{D}$ (*P a* $[\![S]\!]$ *SKIP res'*)›
hence ‹$s \in \mathcal{D}$ *?rhs1*›
by (*simp add*: *D-Det D-Mprefix image-iff* $*(1, 2)$ ‹$s = ev\ a \# s'$›)
with *same-div* **show** ‹$(s, Z) \in \mathcal{F}$ *?lhs*› **by** (*simp add*: ‹*res* = *res'*› *is-processT8*)
next
fix *t u X Y* **assume** $**$: ‹$(t, X) \in \mathcal{F}$ (*P a*)› ‹$(u, Y) \in \mathcal{F}$ (*SKIP res'*)›
‹*s'* *setinterleaves* ((*t*, *u*), *range tick* $\cup$ *ev* ‘ *S*)›
‹*Z* = (*X* $\cup$ *Y*) $\cap$ (*range tick* $\cup$ *ev* ‘ *S*) $\cup$ *X* $\cap$ *Y*›
from $**(2, 3)$ **have** ‹(*ev a* $\#$ *s'*) *setinterleaves* ((*ev a* $\#$ *t*, *u*), *range tick* $\cup$ *ev* ‘ *S*)›
by (*auto simp add*: $*(1, 2)$ *F-SKIP image-iff*)
thus ‹$(s, Z) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Sync F-Det F-Mprefix image-iff*)
(*metis* $*(1)$ $**(1, 2, 4)$ ‹$s = ev\ a \# s'$› *list.distinct*(*1*))
qed
qed
next
assume ‹*res* $\neq$ *res'*› ‹$(s, Z) \in \mathcal{F}$ *?rhs2*›
then consider ‹$s = []$›
| *a s'* **where** ‹$a \in A$› ‹$a \notin S$› ‹$s = ev\ a \# s'$› ‹$(s', Z) \in \mathcal{F}$ (*P a* $[\![S]\!]$ *SKIP res'*)›
by (*auto simp add*: *F-Ndet F-STOP F-Mprefix*)
thus ‹$(s, Z) \in \mathcal{F}$ *?lhs*›
proof *cases*

```
    have ‹([], UNIV) ∈ ℱ ?lhs›
      apply (simp add: F-Sync, rule disjI1)
     apply (rule-tac x = ‹[]› in exI, simp add: F-Det F-Mprefix F-SKIP T-SKIP
D-SKIP)
      apply (rule-tac x = ‹[]› in exI, rule-tac x = ‹UNIV − {✓(res)}› in exI)
      apply (simp, rule-tac x = ‹UNIV − {✓(res')}› in exI)
      using ‹res ≠ res'› by auto
    thus ‹s = [] ⟹ (s, Z) ∈ ℱ ?lhs› by (auto intro: is-processT4)
  next
    fix a s' assume ∗ : ‹a ∈ A› ‹a ∉ S› ‹s = ev a # s'› ‹(s', Z) ∈ ℱ (P a ⟦S⟧
SKIP res')›
    from ∗(4) consider ‹s' ∈ 𝒟 (P a ⟦S⟧ SKIP res')›
      | t u X Y where ‹(t, X) ∈ ℱ (P a)› ‹(u, Y) ∈ ℱ (SKIP res')›
                      ‹s' setinterleaves ((t, u), range tick ∪ ev ‘ S)›
                      ‹Z = (X ∪ Y) ∩ (range tick ∪ ev ‘ S) ∪ X ∩ Y›
      by (auto simp add: F-Sync D-Sync)
    thus ‹(s, Z) ∈ ℱ ?lhs›
    proof cases
      assume ‹s' ∈ 𝒟 (P a ⟦S⟧ SKIP res')›
      hence ‹s ∈ 𝒟 ?rhs2› by (simp add: ∗(1, 2, 3) D-Ndet D-Mprefix)
        with same-div show ‹(s, Z) ∈ ℱ ?lhs› by (simp add: ‹res ≠ res'›
is-processT8)
    next
      fix t u X Y assume ∗∗ : ‹(t, X) ∈ ℱ (P a)› ‹(u, Y) ∈ ℱ (SKIP res')›
                              ‹s' setinterleaves ((t, u), range tick ∪ ev ‘ S)›
                              ‹Z = (X ∪ Y) ∩ (range tick ∪ ev ‘ S) ∪ X ∩ Y›
      show ‹(s, Z) ∈ ℱ ?lhs›
        apply (simp add: F-Sync, rule disjI1)
        apply (rule-tac x = ‹ev a # t› in exI)
        apply (rule-tac x = u in exI)
        apply (rule-tac x = X in exI)
        apply (rule conjI, solves ‹simp add: F-Det F-Mprefix ∗(1) ∗∗(1)›)
        apply (rule-tac x = Y in exI)
        apply (simp add: ∗∗(2, 4))
        using ∗∗(2, 3) by (auto simp add: ∗(1, 2, 3) F-SKIP)
    qed
  qed
 qed
qed


lemma Sliding-def-bis : ‹P ▷ Q = (P ⊓ Q) □ Q›
 by (simp add: Det-distrib-Ndet-right Sliding-def)
```

Chapter 10

Conclusion

We started by defining the operators ($\triangleright$), *Throw* and ($\triangle$) and provided on them several new laws, especially monotony, "step-law" (behaviour with $\Box a \in A \rightarrow P\ a$) and continuity.

We defined the *initials* notion, and described its behaviour with the reference processes and the operators of `HOL-CSP` and `HOL-CSPM` (which is already a minor contribution).

As main contribution, we defined the *After.After* operator which represents a bridge between the denotational and the versions of operational semantics for CSP. We made the construction as generic as possible, by exploiting the locale mechanism. Therefore we derive the correspondence between denotational and operational semantics by construction. Based on failure divergence or trace divergence refinements, the two operational semantics correspond to the versions described in [4, 6].

We have slight variations that can open up for discussion.

Thus, we provided a formal theory of operational behaviour for CSP, which is, to our knowledge, done for the first time for the entire language and the complete FD-Semantics model. Some of the proofs turned out to be extremely complex and out of reach of paper-and-pencil reasoning.

A notable point is that the experimental order ($\sqsubseteq_{DT}$) behaves surprisingly well: initially pushed in `HOL-CSP` for pure curiosity, it looks promising for future applications, since it gives a direct handle for an operational trace semantics for non-diverging processes which is executable.

Another take-away is the development of alternatives with ($\sqsubseteq_F$) and ($\sqsubseteq_T$) orders but this remains a bit disappointing because their monotony w.r.t. to some operators does not allow to recover all the laws of [4, 6].

As a bonus we provided in *HOL−CSP-OpSem.CSP-New-Laws* some powerful laws for CSP. Here, we recall only the most important ones:

$$\frac{bij\ ?f \qquad bij\ ?g}{Renaming\ (?P \setminus ?S)\ ?f\ ?g = Renaming\ ?P\ ?f\ ?g \setminus ?f\ `\ ?S}$$

$$\frac{bij\ ?f \qquad bij\ ?g}{Renaming\ (?P\ [\![?S]\!]\ ?Q)\ ?f\ ?g = Renaming\ ?P\ ?f\ ?g\ [\![?f\ `\ ?S]\!]\ Renaming\ ?Q\ ?f\ ?g}$$

$$\frac{?A \cap ?S \neq \emptyset}{\Box a{\in}?A \rightarrow ?P\ a \setminus ?S = (\Box a{\in}(?A - ?S) \rightarrow (?P\ a \setminus ?S)) \triangleright (\sqcap a{\in}(?A \cap ?S).\ (?P\ a \setminus ?S))}$$

Finally, we discovered that the *After.After* operator and its extensions *AfterExt.After*$_{tick}$ and *AfterExt.After*$_{trace}$ have a real interest even without the construction of operational semantics.

With induction rules based on *AfterExt.After*$_{trace}$, we could for example prove the following theorem:

$$\frac{deadlock\text{-}free\ ?P \qquad deadlock\text{-}free\ ?Q \qquad \alpha(?Q) \cap ?S = \emptyset}{deadlock\text{-}free\ (?P\ [\![?S]\!]\ ?Q)}$$

Bibliography

[1] B. Ballenghien, S. Taha, and B. Wolff. Hol-cspm - architectural operators for hol-csp. *Archive of Formal Proofs*, December 2023. https://isa-afp.org/entries/HOL-CSPM.html, Formal proof development.

[2] B. Ballenghien and B. Wolff. An operational semantics in isabelle/hol-csp. In Y. Bertot, T. Kutsia, and M. Norrish, editors, *15th International Conference on Interactive Theorem Proving, ITP 2024, September 9-14, 2024, Tbilisi, Georgia*, volume 309 of *LIPIcs*, pages 7:1–7:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[3] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 281–305, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[4] A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

[5] A. W. Roscoe. Understanding concurrent systems. In *Texts in Computer Science*, 2010.

[6] A. W. Roscoe. The expressiveness of CSP with priority. In D. R. Ghica, editor, *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 387–401. Elsevier, 2015.

[7] S. Taha, L. Ye, and B. Wolff. Hol-csp version 2.0. *Archive of Formal Proofs*, April 2019. https://isa-afp.org/entries/HOL-CSP.html, Formal proof development.